Entendendo valores e ponteiros em C++

Por: Matías Rodriguez (matias@sumersoft.com)

Este tutorial tenta responder as seguintes perguntas:

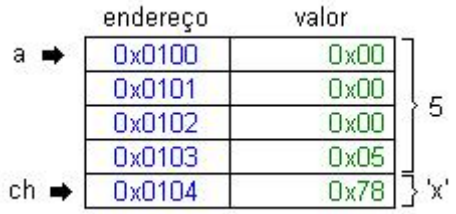
- O que são valores, ponteiros, ponteiros para ponteiros, referências, etc?
- Quando que usa '*' e '&'? Dependendo do contexto querem dizer coisas diferentes.

Valores:

Valores são o mais fácil de se entender. Ver o exemplo:

```
1  void  main()
2  {
3      int a=5;
4      char ch='x';
5  }
```

Para nós é facil de entender que "a" é igual ao valor 5 e "ch" é igual ao valor 'x'. Mas para poder entender o que é um ponteiro mais adiante, esta definição de valor não é suficiente. Temos que "baixar o nível" um pouco e ver da perspectiva do compilador e/ou preocessador. Vocês concordam que para o compilador/processador "a" ou "ch" não faz sentido? Para o compilador/processador "a" e "ch" são endereços de memória. Esta é a chave de tudo! :) São "lugares" da memória, que contém, no caso de "a" um inteiro com valor 5 e no caso de "ch" um caractere com valor 'x'. Isto é muito importante. Então, em quanto nós (humanos) entendemos "a" é igual 5 e ch é igual ao caractere 'x', ele (o computador) entende:



Vamos entender a figura, cada linha representa um byte. A coluna da direita (verde) representa a memória em si, e a coluna da esquerda (azul) representa o endereço de cada byte da memória. A notação utilizada é a hexadecimal, o endereço (azul) foi escolhido aleatoriamente, mas o fato deles serem seqüenciais, não é coincidência.

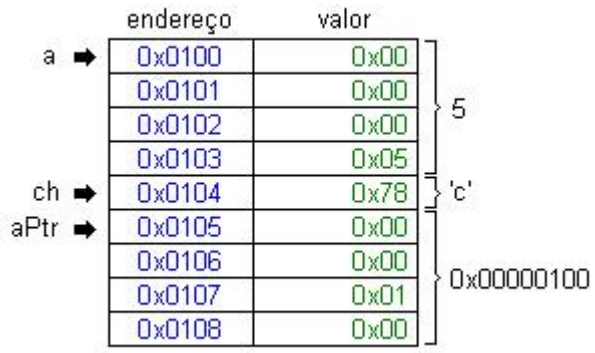
Notar que o que nós enxergamos como "a" o computador (neste exemplo) enxerga como o endereço 0x0100 e pelo fato de "a" ser do tipo int, ele ocupa 4 bytes. O mesmo acontece com "ch", o computador enxerga como o endereço 0x0104 e por ser do tipo char ocupa somente um byte. Resumindo, o endereço 0x0100 ("a") da memória tem o inteiro 0x00000005 (valor 5) e o endereço 0x0104 ("ch") da memória tem 0x78 (valor ASCII hexadecimal do caractere 'x').

Ponteiros:

Vamos ver o porquê de uma definição complexa de um valor. Ao explicar valor, vimos como o computador enxerga uma variável: através de seu endereço na memória. Até então, enxergar uma variável como um endereço de memória era um privilégio do computador, nós tinhamos que nos conformar com ver somente seu valor. Mas por sorte alguêm inventou os ponteiros! Ou seja, quando falamos ponteiro, estamos nos referindo a um endereço de memória e não a um valor. Talvez seja interessante fazer uma analogia entre ponteiro e link. Um ponteiro nos "leva" de uma posição da memória até outra como um link nos leva de um site até outro. Ponteiro é basicamente isto. O resto é entender a sintaxe do C++. Para entender a sintaxe nada melhor que exemplos práticos:

```
1  void main()
2  {
3      int a=5;
4      char ch='x';
5      int* aPtr=&a;
6  }
```

Agora complicou! :) Como foi dito, é so uma questão de entender a sintaxe do C++. Na quinta linha existem duas coisas importantes: A definição da variável "aPtr" (antes do '=') e a inicialização dessa variável (depois do '='). Esta sintaxe nos diz que "aPtr" é do tipo ponteiro para inteiro (int*) e ele recebe o endereço da variável "a" (&a). Mas o que isto quer dizer? Isto quer dizer que "aPtr" **não** é um inteiro, na verdade ele aponta para um inteiro, ou seja, seu valor na memória **não** é o de um inteiro e sim de um endereço e o valor deste endereço é um inteiro. Nossa, complicou denovo! :) Já que estamos falando de ponteiros que apontam para endereços de memória, é interessante ver isso graficamente:



A única diferença desta figura para a anterior é a variável "aPtr". Um ponteiro para inteiro ocupa 4 bytes. Podemos ver que o valor de "aPtr" (coluna verde) é 0x0100 que corresponde ao endereço de "a" que é exatamente o que diz na linha 5 do código fonte: crie um ponteiro para inteiro ("aPtr") e inicialize ele com o endereço de "a". As seguintes expressões C++ são erradas (não compila, erros de tipo pois ponteiro para inteiro **não** é a mesma coisa que um inteiro):

```
1  int* aPtr=a;
2  int* bPtr=10;
3  int* cPtr=0x0100;
4  int* dPtr=&10;
```

Na linha 1, "a" é do tipo inteiro e "aPtr" do tipo ponteiro para inteiro o que faz com que a atribuição "ponteiro para inteiro recebe inteiro" seja ilegal, C++ é uma linguagem fortemente tipada!
Na linha 2 "bPtr" é ponteiro para inteiro e 10 é um inteiro. Neste caso 10 é uma constatnte porque 10 vai ser sempre 10, fazer algo do tipo 10=2; é ilegal. É o mesmo erro da linha 1.
Na linha 3, mmm.... você esta tentando enganar o compilador?! :) A coluna da esquerda (azul) tem valores fictícios que servem somente para este exemplo. Para o compilador, 0x0100 nada mais é do que um inteiro (mesmo problema que na linha 2). Se a intenção é fazer "cPtr" apontar para algum endereço de memória, deve-se fazer usando a sintaxe do C++ que é utilizar o operador '&' antes do nome da variável ou ele receber o valor de um outro ponteiro. A única exceção é inicializar um ponteiro para NULL: `int* cPtr=NULL; //isto é valido`.
A expressão da linha 4 é errada pois 10 é uma constante (não confundir com o modificador const, ver explicação da linha 2) e não ocupa lugar físico na memória que nem uma variável (aPtr, bPtr, etc).
Até agora, aprendemos duas coisas: Como declarar um ponteiro: Adicionar um asterisco após o tipo da variável. E como fazer ele apontar para outra variável: Colocar um 'E' comercial antes da variável.

A pergunta que pode estar surgindo é a seguinte: Como modifico o valor de uma variável se eu tenho somente um ponteiro para ela? fazer algo como:

```
1  int a=10;
2  int* aPtr=&a;
3  aPtr=11; //isto não compila
```

Não funciona pelos mesmos motivos mencionados acima, ou seja, tipos diferentes. O C++ possui um operador onde vc pode dizer: "**Não** modifique meu valor, modifique o valor da variável apontada por mim". Talvez por falta de teclas no teclado, os arquitetos do C++ decidiram utilizar novamente a tecla '*' (asterisco) para representar este operador. Pode parecer meio confuso no começo, mas após um certo tempo e experiência, a distinção passa a ser natural:

```
1  int a=10;
2  int* aPtr=&a;
3  *aPtr=11; //jeito certo, agora o valor de "a" é 11.
```

Aqui vemos a analogia entre ponteiro e link funcionando, ao fazer *aPtr=11 estamos navegando até a variável "a" e mudando seu valor. Vamos ver mais exemplos:

```
1  //declaro dois inteiros ("a" e "b") e dois
2  //ponteiros para inteiro ("ptr1" e "ptr2")
3  int a, b, *ptr1, *ptr2;
4  a=10; //inicializo "a".
5  b=11; //inicializo "b".
6  //"ptr1" recebe o endereço de "a".
7  ptr1=&a;
8  //"ptr2" recebe o valor de "ptr1" que é o endereço de "a".
9  //Tudo bem pois os dois ("ptr1" e "ptr2") são ponteiros para inteiro).
10 ptr2=ptr1;
11 //"ptr1" aponta para "b". Notar que "ptr2" continua apontando para
"a"!
12 ptr1=&b;
13 //o valor da variável apontada por "ptr2" ("a") recebe 20.
14 //é a mesma coisa que a=20;
15 *ptr2=20;
16 //"b" recebe 21. Então *ptr1 também é igual a 21 pois "ptr1" aponta
para "b".
17 b=21;
18 //Nossa! :) Isto é uma expressão burra pois é a mesma coisa que
escrever a=30;
19 //de uma forma mais difícil. É so utilizar os conceitos aprendidos:
20 //"a" é um inteiro e quando fazemos: &a estamos pegando o endereço de
"a".
21 //e ao fazer *(...) estamos navegando para aquele ondereço e
atualizando o
22 //valor da variável com 30.
23 *(&a)=30;
```

Tentar acompanhar um programa como este não é fácil, mesmo com um debugger bom podemos ficar confusos e perdidos. Chama-se nível de indireção ao número de vezes que tenho que "navegar" para chegar na minha varável final. Na linha quinze por exemplo, o número de indireções é um. E na linha 17 é zero (não existe indireção nenhuma). A medida que o número de indireções aumenta, a dificuldade de entender o programa também aumenta.

Ponteiro para Ponteiro:

O quê? isto existe? :) Sim, isto existe e só nossa imaginação pode nos deter. Exemplo, a seguinte linha é válida:

```
int****** ptr=NULL;
```

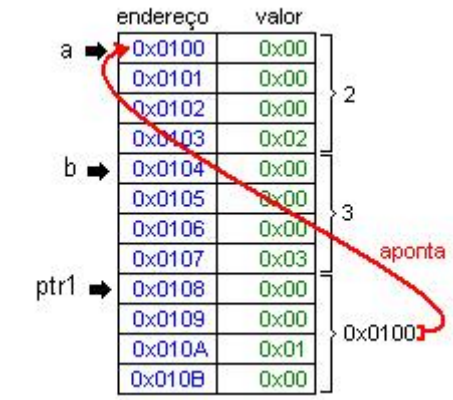
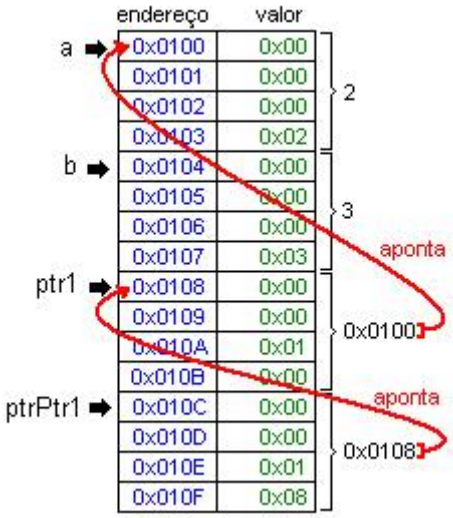
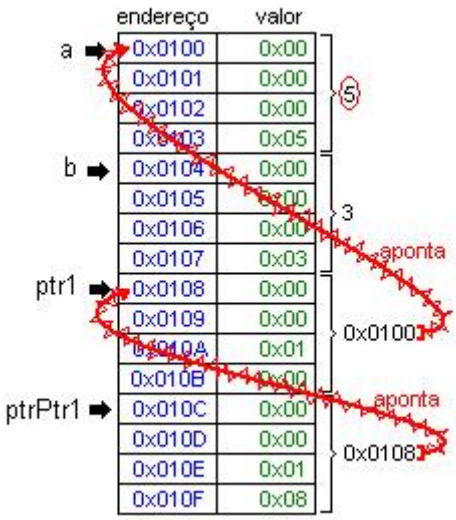
É um ponteiro para ponteiro para ponteiro para ponteiro para ponteiro para ponteiro para inteiro! Para chegar no "destino" final temos que fazer 6 indireções. Um programa que utiliza uma arquitetura assim, não somente fica MUITO difícil de entender mas também pode ficar lento.

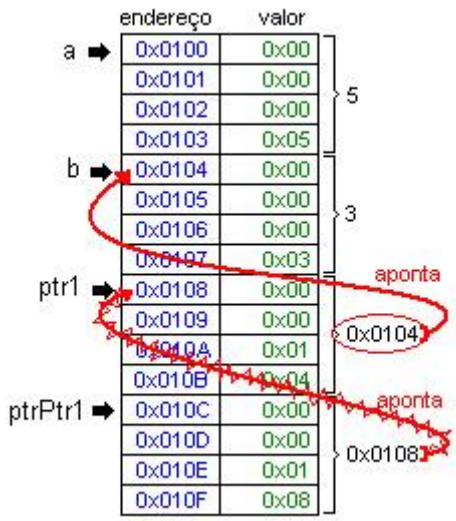
Entender ponteiro para ponteiro requer o conhecimento adquirido até agora, ou seja, o conhecimento de ponteiros. A diferença é que se temos uma variável que é um ponteiro para ponteiro para inteiro, para chegar até o inteiro no final precisamos fazer dois níveis de indireções.

Ex:

```
1  //criar a variável "a" e iniciá-la com o valor 2 e a variável "b" com
3.
2  int a=2, b=3;
3  //criar um ponteiro para inteiro que recebe o endereço de "a".
4  int* ptr1=&a;
5  //criar um ponteiro para um ponteiro para inteiro e inicializá-lo com
o endereço
6  //de "ptr1". Uma expressão como int** ptrPtr1=ptr1; não é valida. O
motivo
7  //já foi discutido: Tipagem, ptrPtr1 é um ponteiro para ponteiro para
inteiro
8  //e ptr1 é ponteiro para inteiro, ou seja, tipos diferentes.
9  int** ptrPtr1=&ptr1;
10 //agora "a" é igual a 5. Estamos fazendo duas indireções.
11 **ptrPtr1=5;
12 //agora "ptr1" aponta para "b" e não mais para "a". Notar que não
13 //referenciamos "ptr1".
14 *ptrPtr1=&b;
15 //agora "b" é igual a 6.
16 **ptrPtr1=6;
17 //Modificar "b" para 6 neste exemplo pode ser feito das seguintes
18 //outras formas:
19 //b=6; ou *ptr1=6;
```

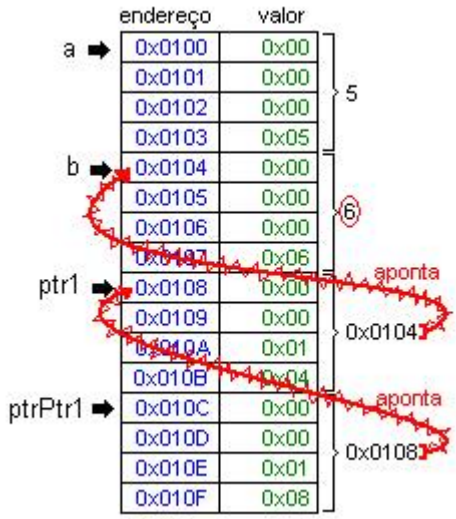
Notar que mesmo que ptrPtr1 seja um ponteiro para ponteiro para inteiro (linha 14), podemos fazer uma indireção somente e modificar aquele valor respeitando o tipo. Fazendo a mesma coisa graficamente acompanhando a situação da memória:

	`4 int* ptr1=&a;` Após a linha 4, a situação da memória é a mostrada na figura à esquerda. Onde "ptr1" aponta para "a", ou seja, seu valor é o endereço de "a". O valor da variável "a" é 2 e de "b" é 3.
	`9 int** ptrPtr1=&ptr1;` Após a linha 9, vemos que a variável "ptrPtr1" foi criada e inicializada com o endereço de "ptr1", ou seja, ela aponta para "ptr1" e "ptr1" aponta para um inteiro ("a"). Aqui vemos graficamente o significado de um ponteiro para um ponteiro para inteiro.
	`11 **ptrPtr1=5;` Na linha 11, o valor da variável "a" foi modificado através de "ptrPtr1". Para fazer isto, foi necessário fazer duas indireções (linhas riscadas na figura e os asteriscos na linha 11 do código) na variável "ptrPtr1". Notar que o valor da variável "a" agora é 5.




```
14 *ptrPtr1=&b;
```

Na linha 14, fazendo uma indireção somente, podemos modificar o valor do ponteiro para inteiro "ptr1". Agora em vez dele apontar para "a", aponta para "b".




```
16 **ptrPtr1=6;
```

Agora com duas indireções a variável modficada é "b". O valor de "b" agora é 6.