Ponteiros, Vetores e Leite em Pó Líquido

Antes de começar a nossa história sobre ponteiros e afins, vamos assumir alguns conceitos que todos já têm nesta altura do campeonato.

Nos nossos programas em C, declaramos variáveis de diferentes tipos: char, int, float, double, etc. Estas variáveis podem ser **globais** (fora da função main) **locais** (dentro da função main) ou mesmo **dentro de algum bloco** de comando (while, if, etc). Uma variável global tem escopo global (visível para o programa inteiro), uma variável dentro da sua função main (pode ser vista somente dentro da main) e a variável dentro de um bloco, finalmente, só tem visibilidade dentro deste bloco. Exemplo:

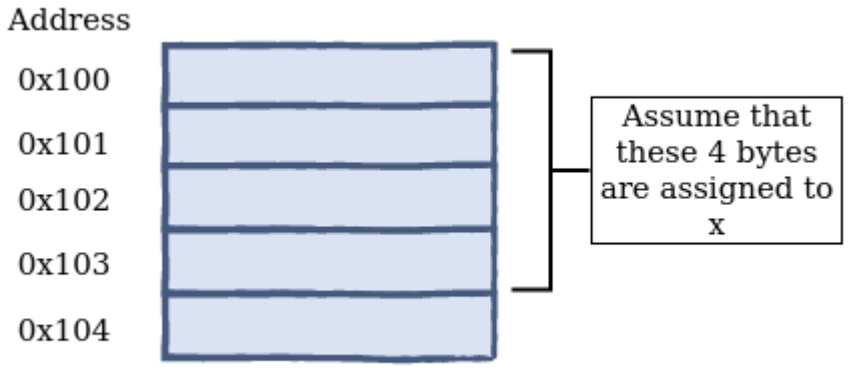
****

```
#include <stdio.h>

int a;


int main(){
   int b;
   char letra;

   // leia a e b
   scanf (“%d”, &a);
   scanf (“%d”, &b);

    if (a>b) {
       int temp;
       // troca os valores de a e b….
       temp = a;
       a = b;
       b = temp;
    }

    print(“soma de a + b = %d\n”, a+b);
}
```

Está claro que variáveis do tipo int (ou char, float, double, etc), quando as manipulamos (lendo, imprimindo, atribuindo, etc) estamos nos referindo aos **valores** destas variáveis. O comando temp = a, atribui o **conteúdo** de a para e coloca este valor no **conteúdo** de b.

Ok. Isso me parece óbvio, assim como também é óbvio que toda variável ocupa um espaço na RAM e que este espaço de alocação é endereçável, ou seja, toda variável que declaro no meu programa tem um endereço. No caso de uma variável **int,** ela ocupa 4 bytes da

RAM (processadores 64 bits). Assuma que a RAM inteira é **endereçada byte a byte**, **contiguamente**, como ilustra a figura abaixo.



Para fins ilustrativos, nossa variável global “a”, foi alocada a partir da posição 0x100 e, portanto, ocupa as posições de memória (0x100, 0x101, 0x102 e 0x103). Já a variável char ‘letra’, cujo tamanho é de apenas um byte, poderia ter sido alocada na posição 0x104 e ocupa somente espaço.

Use sua imaginação para visualizar o seu programa, com inúmeras variáveis alocadas na RAM. Lembre-se de que o compilador tem certa vida própria e vai alocando as variáveis onde bem entender e não necessariamente na ordem em que você declara no programa fonte!

Então, quando fazemos a = b, o **conteúdo** da variável b é copiado para a variável a. Ou seja, 4 bytes de uma certa região de memória (variável b) são transferidos para o endereço de memória de ‘a’, byte a byte, a partir da posição 0x100. Alteramos literalmente o conteúdo de a, mas sem a mínima necessidade de lidar com o endereço dela ou mesmo de b, certo?

Agora retome a chamada da função scanf (“%d”, &a) que lê do terminal um inteiro e armazena em a. Podemos dizer que é algo análogo a fazer a = b, correto?

Correto! A diferença é que o valor da minha variável será alterado por meio de uma função e não por meio de uma atribuição. Aí eu lhe faço 2 perguntas:

a) Quem implementou a função scanf() ? **Resp:** Eu não sei, mas eu sei que faz parte da biblioteca padrão da linguagem C, e está acessível ao incluirmos #include <stdio.h> lá no início do meu programa. Ao compilar o meu programa, em incorporo a função scanf() da biblioteca e ela passa a fazer parte do meu código executável (isso na verdade não necessariamente é assim ** podemos linkar a biblioteca padrão dinamicamente ** mas isso é irrelevante aqui).

b) Como é possível, uma função que eu não escrevi e que não conhece a minha variável, modificar o conteúdo dela??? **Resp:** se a função souber o ENDEREÇO da minha variável, esta função pode modificar o seu conteúdo !

Já usamos scanf dezenas de vezes e o uso dela requer que passemos o símbolo **'&' do lado esquerdo da variável**. Este símbolo '&' significa ENDEREÇO. E a maneira de 'traduzir' a chamada de função scanf ("%d", &a) é a seguinte: leia do terminal uma sequência de caracteres e converta para inteiro ("%d" ** isso já sabemos) e passe para ela o ENDEREÇO da minha variável 'a' ! Ou seja, a função não recebe uma CÓPIA (conteúdo) da variável 'a' (que provavelmente nem tem valor ainda), mas sim o ENDEREÇO dela, que é 0x100. É por este motivo que você tem que necessariamente passar '&a' e não somente 'a'.

Aí vem mais uma pergunta:

a) Como é a implementação da função scanf()? Como a função sabe que está lidando com um endereço de uma variável de tipo int e não o CONTEÚDO dela?

Para responder a esta pergunta, vamos implementar algo que tenha o mesmo comportamento da função scanf(). Seja o programa

```
#define <stdio.h>

int main(){
    int a=10, b=20, soma=0;

    soma1 (a, b, soma);
    printf("%d", soma);
}
```

```
#define <stdio.h>

int main(){
    int a=10, b=20, soma=0;

    soma2 (a, b, &soma);
    printf("%d", soma);
}
```

```
void soma1(int x, int y, int z){
    z = x+1;
}
```

```
int soma2(int x, int y, int *z){
   *z = x + y;
}
```

Perguntas:

a) após execução de soma1(), qual será o valor de soma?
b) após execução de soma2(), qual será o valor de soma?

Respostas:

a) soma = 0
b) soma = 30

Por que a função soma1 está errada? Porque você criou uma função que tem 3 argumentos e TODOS eles são inteiros (x,y e z). Quando a função é chamada no programa principal, ele recebe **CÓPIAS DOS VALORES** a, b e c, apenas. No entanto, esta função não tem a

mínima noção de quem é a variável ‘soma’, que de fato deve ser mudada! A função, localmente, faz o serviço devido: soma x+y e coloca na variável temporária ‘z’, o resultado 30. MAS EM NENHUM MOMENTO a variável soma (do seu programa) foi modificada !!!!
Moral da história: a função soma1() é ecologicamente correta,

Entendeu?????

Agora a função do lado esquerdo: veja que a chamda de soma2() é ligeiramente diferente. Nela, assim como no scanf(), passamos o ENDEREÇO de soma (&soma), pois sei que este é o único jeito de uma função externa ao meu escopo (soma2()) alterar o conteúdo da minha variável local (soma é local a main()).

a) E qual a diferença fundamental entre soma1() e soma2()?

RESP: a declaração do terceiro argumento! Sabemos, pela chamada, que este argumento não é um mero inteiro. É um **ENDEREÇO de um inteiro**.

E como devemos implementar uma função, cujo argumento é um ENDEREÇO (int, char, float, etc)?

RESP: Por meio de PONTEIRO (‘*’).

Veja a implementação de soma2(int x, int y, int *z)(). Devemos ‘ler’ os argumentos da função de trás para frente: soma2 tem três argumentos: 1) x é um inteiro; 2) y é um inteiro e 3) z é **ponteiro para inteiro**.

E o que é um ponteiro?

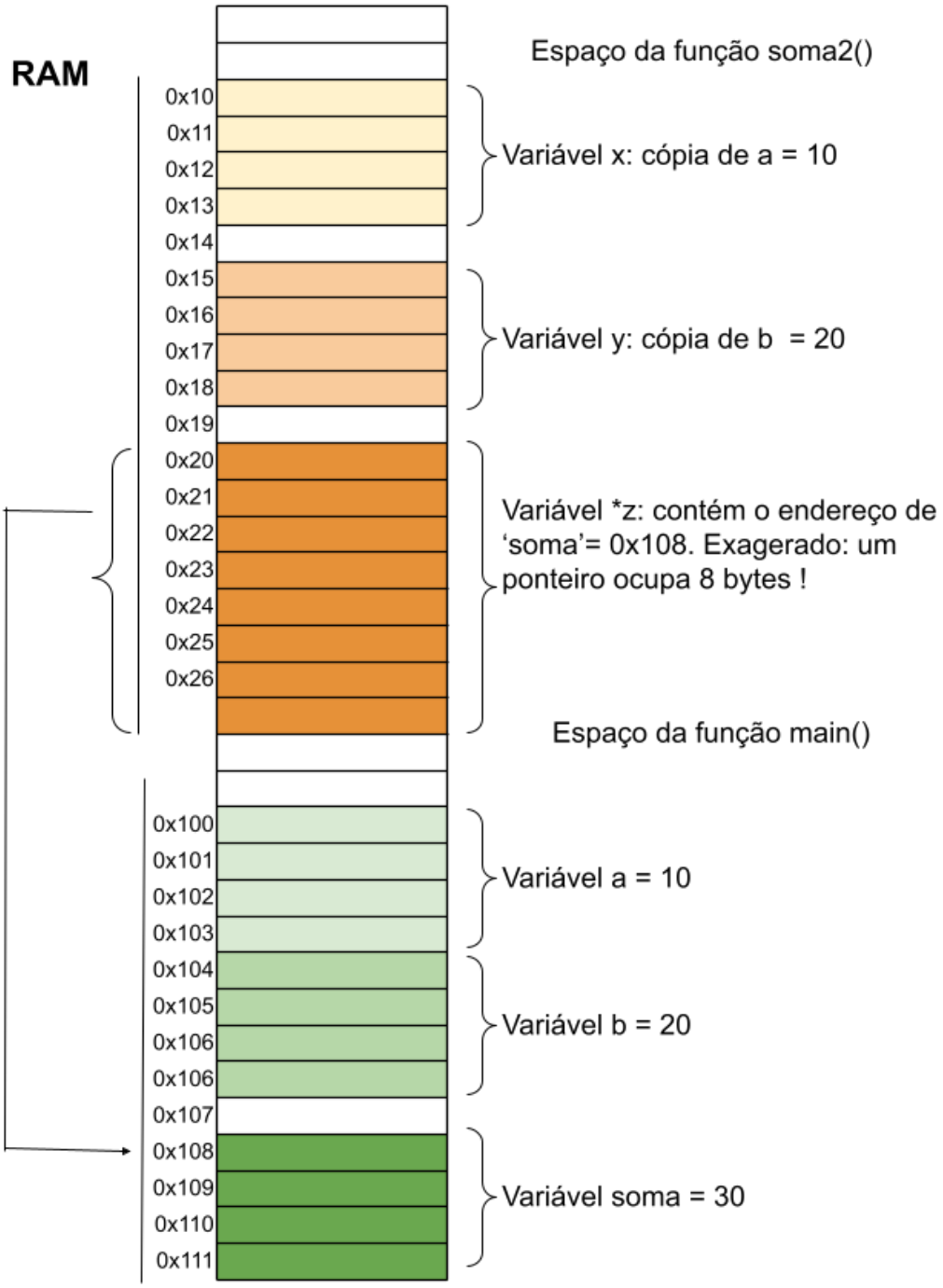
RESP: ponteiro é uma variável que contém o endereço de outra variável !!!

E em C, qualquer ponteiro tem tamanho de 8 bytes. Por que tão grande? Porque a RAM é grande e com 8 bytes dá pra endereçar 2^{64} posições distintas.

Sabendo-se que o programa inteiro está na RAM (neste caso as instruções das funções main() e soma2()), sabendo-se que variáveis locais das funções também ocupam espaço na RAM, podemos representar esquematicamente a execução como ilustrado na figura da página seguinte.

Observe que int *z (z é um ponteiro para inteiro) contém o endereço de soma. Portanto, o conteúdo dos 4 bytes reservados para este ponteiro contém o valor 0x108. Agora repare na sintaxe da atribuição em soma2(): *z = x+y. Como interpretamos isso?

>>> some x+y (10+20) e coloque o resultado em *z (que é um inteiro!!!!!) Mas que inteiro? O inteiro que está no endereço iniciado por 0x108 (a variável ‘soma’). É desta forma que qualquer função pode modificar uma variável squel ela nem sabe que existe. Basta especificar um ponteiro.



Observe ainda que uma variável ponteiro, como qualquer outra variável, também tem o seu próprio endereço. No esquema acima z foi alocada na posição de memória 0x20.

Se vc entendeu esta relação entre ponteiros e endereços, então será capaz de entender o código abaixo:

```
**********************************************
#define <stdio.h>

int main () {
   int a = 10;

   int *p = &a        // (a)
   print(“%p”, p);        // (b)
   print(“%p”, &a);     //  (c)
   print(“%d”, *p);      // (d)
   print(“%d”, a);       // (e)
   print(“%p”, &p);    // (f)
}
**********************************************
```

(a): Note que esta atribuição equivale ao esquema de passagem de parâmetros por endereço, como mostramos na função soma2() ! O endereço de uma variável inteira ‘a’ é passado para uma variável ponteiro p.

Muito importante (leia sempre da direita para a esquerda): 1) o que é p? p é um ponteiro para inteiro (int *p); 2) o que é *p? Ora, *p é um inteiro (int *p) !!!!

Isso é muito importante para você entender a corretude das próximas linhas do programa acima:

(b) Lembre-se de que o modificador “%p” é para imprimir endereços. Então está correto e vai imprimir o endereço da variável ‘a’, pois p (que é um ponteiro para inteiro, portanto, contém um endereço) armazena exatamente o endereço de ‘a’.

(c): Análogo a (b). Vai imprimir o endereço de ‘a’.

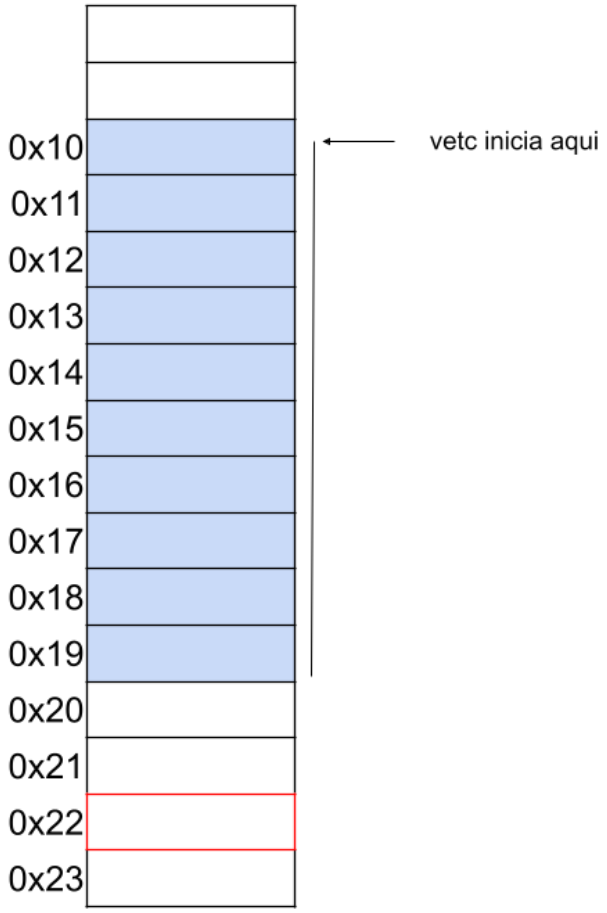
(d): Ora. O que é *p ? Leia da direita para a esquerda: *p é um inteiro! E que inteiro é este? ‘a’ . Portanto, imprime o conteúdo de a !

(e): sem comentários !
(f): lembre-se de que uma variável ponteiro ocupa memória e tb tem o seu próprio endereço. Portanto, este comando imprime o endereço de p, cujo conteúdo é o endereço de ‘a’.

Vetores

Declarar um vetor é bem simples. Exemplos: int vet[10]; char vetc[10]; float vetf[100]; etc.

Vamos tomar como exemplo o vetor de char vetc[10]. Sabemos que um char ocupa apenas um byte. Vamos supor que o compilador inicia o vetor na posição 0x10 de memória. Portanto, este vetor está assim representado na RAM:



Mas vetores e ponteiros compartilham semelhanças.

O primeiro elemento do vetor está alocado na posicão 0x10 de memória (vetc[0]) e o décimo elemento, na posição 0x19 (vetc[9]).

Seja uma variável char única (char c) foi alocada na posição 0x22 (em vermelho), como ilustrado na figura acima. E a ‘c’ atribuímos o conteúdo ‘Z’. (c = ‘Z’).

Se fizermos printf(“%c”, c), teremos ‘Z’ como saída.

Mas e se fizermos printf(“%c”, vetc)?

Isso gerará um ‘warning’ e o compilador dirá a você que “%c” espera um **char**, mas que vc passou como argumento um “*** char**”.

Isso quer dizer que a variável **vetc equivale a** um ponteiro para char (char *). O nome de uma variável declarada como vetor (vetc), quando invocada sem índice algum, representa SEMPRE o endereço do primeiro elemento do vetor e nunca o seu conteúdo.

Portanto, o correto seria sempre fazer printf(“%p”, vetc), que retornaria 0x10.

E o que significa vetc+9 ? Ora, significa que a soma de 0x10 + (9*sizeof(char)) = 0x10+9 = 0x19, o endereço do último elemento do vetor!

Portanto, se quisermos ler todos os elementos do vetor podemos fazer:

```
for (int i = 0, i<10; i++)
   scanf(“%c”, vetc+i);                  // vet+i já é um endereço !! Portanto, nada de ‘&’ aqui ….
```

que é exatamente igual a se fazer:

```
for (int i = 0, i<10; i++)
   scanf(“%c”, &vetc[i]);           // vetc[5] é o conteúdo do sexto elemento. scanf requer ‘&’ …
```

Por que? Porque vetc[i] é o conteúdo de cada elemento do vetor (0<= i < 10). E já que vetc[i] é conteúdo, então, para usar esta notação em scanf, precisamos passar o endereço !

E já que entendemos o que é um ponteiro, então, o que significa *(vet+2)??? Ora, é o conteúdo do terceiro elemento do vetor!

Isso quer dizer que *(vetc+2) = vetc[2] ?? EXATAMENTE. A notação com índices é mais comum, porque é mais próxima da representação matemática de vetores. Internamente, o compilador converte esta notação matemática para notação de ponteiros.

O LEITE EM PÓ LÍQUIDO.

Suponha que eu queira ler o 5 elemento do vetor (vetc[4]).

Podemos fazer: scanf(“%c”, &vetc[4]); OU

scanf(“%c”, vetc+4); OU

scanf(“%c”, &(*(vetc+4)));

sendo que esta última forma eu chamo de LEITE EM PÓ LÍQUIDO.

Por que?!?!?!?!

Como exercício peço que você execute o programa abaixo (que não contém erro algum) e tente entender todas as saídas.

****

```
#include <stdio.h>
int main(int argc, char const *argv[])
{
   int vetor[100];
   int a = 10;
   int *pv = vetor;

   printf("sizeof(a) = %ld\n", sizeof(a));
   printf("sizeof(vetor) = %ld\n", sizeof(vetor));
   printf("conteúdo de vetor = %p\n", vetor);
   printf("conteúdo de pv = %p\n", pv);
   printf("sizeof(pv) = %ld\n", sizeof(pv));
   printf("sizeof(*pv) %ld\n", sizeof(*pv));
   printf("conteúdo de vetor+0 = %p\n", vetor+0);
   printf("conteúdo de &(*(vetor+0)) = %p\n", &(*(vetor+0)));
   scanf("%d", vetor+5);
   scanf(" %d", &vetor[6]);
   printf("conteúdo de vetor[5] = %d\n", vetor[5]);
   printf("conteúdo de vetor[6] =  %d\n", vetor[6]);
   printf("conteúdo de pv[6] =  %d\n", pv[6]);
   printf("conteúdo de  *(vetor+6)  =  %d\n", *(vetor+6));
   printf("contteúdo de *(pv+6) =  %d\n", *(pv+6));
   return 0;
}
```

****

Veja a íntima relação entre ponteiros e vetor (não são sinônimos, mas são muito parecidos). Isso fica claro analisando-se as saídas de printf("sizeof(vetor) = %ld\n", sizeof(vetor)); e printf("sizeof(pv) = %ld\n", sizeof(pv)); A primeira retorna 400 (que é o tamanho do vetor inteiro (cada inteiro ocupa 4 bytes e temos um vetor de 10 inteiros) e o segundo o tamanho de um ponteiro (8 bytes).

Espero que esta pequena apostila tenha lhe ajudado a entender melhor o que são os ponteiros e vetores.

ate mais
João Batista e Marcelo Manzato.