ESTRUTURA DE DADOS

AULA 04 – PONTEIROS (APONTADORES)

PARTE I

PONTEIRO

- **Para uma variável *x*, como declarada a seguir, estão a ela associadas as seguintes informações:**

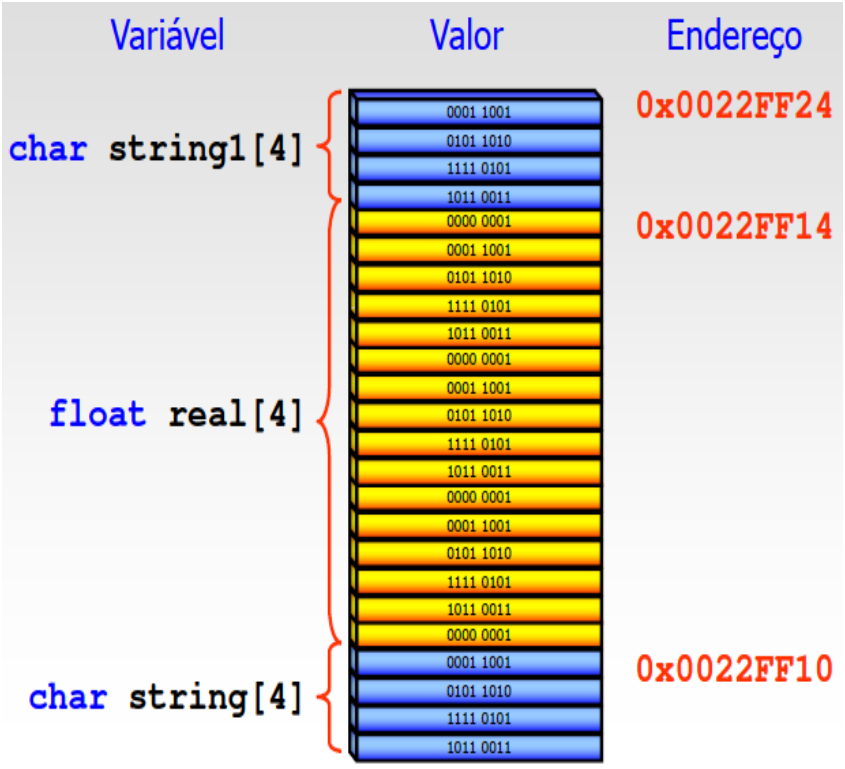
```
int x = 100;
```

- um nome: x ;
- um endereço de memória ou referência: (0xbfd267c4);
- um valor: 100.

Quais são as outras?

PONTEIRO

- A **memória** é **composta** por uma **sequência** de **bytes**, onde cada byte é **identificado** por um **endereço** numérico único.
- Cada objeto (variáveis, strings, vetores etc.) na memória ocupa um certo nº de bytes e possui um endereço.



LOGO, PONTEIRO ...

- **Os *ints* guardam inteiros. Os *floats* guardam números de ponto flutuante. Os *chars* guardam caracteres.**
- **Ponteiros guardam endereços de memória.**
- **A intenção é semelhante a: anotar o endereço de um colega, pois está sendo criado um ponteiro. O ponteiro é este seu pedaço de papel. Ao anotar o endereço de um colega, isto servirá para achar este colega.**
- **É desta mesma forma que o C funciona. Você anota o endereço de algo em uma variável ponteiro para depois usar.**
- **Analogia: como uma agenda, onde são guardados endereços de vários amigos, essa agenda poderia ser vista como sendo uma matriz de ponteiros no C.**
- **Logo, um ponteiro é uma variável que contém o endereço de outra variável.**

PONTEIRO: CARACTERÍSTICAS E RAZÕES

- Proporciona um modo de acesso indireto a uma variável, ou seja, sem referi-la diretamente pelo nome da variável.
- O valor nela armazenado indica em que parte da memória uma outra variável está alocada, mas não indica o valor que está dentro desta outra variável.
- Algumas razões para o uso de ponteiros:
  - Manipular elementos de matrizes;
  - Receber argumentos em funções que necessitem modificar o argumento original;
  - Passar *strings* de uma função para outra;
  - Criar estrutura de dados complexas (listas encadeadas, árvores binárias etc.) em que um item deve conter referência de outro;
  - Alocar e desalocar memória do sistema;
  - Passar para uma função o endereço de outra.

PONTEIRO: + CARACTERÍSTICAS...

- **Por ser uma variável, todo ponteiro:**
  - **Precisa ser declarado;**
  - **tem que ter um tipo de dado a ele associado;**
  - **Armazena valor (endereço);**
  - **A declaração dessa variável aloca espaço;**
  - **Ao nome do ponteiro está associado um endereço de memória que esse ponteiro ocupa, pois este ponteiro é uma variável.**
  - **No C, a declaração do tipo de dado do ponteiro informa ao compilador para que tipo de variável ele apontará e não o tipo de dado que ele armazena. Por exemplo: um ponteiro *int* aponta para um inteiro, isto é, guarda o endereço para chegar a um inteiro.**

Por que isso?

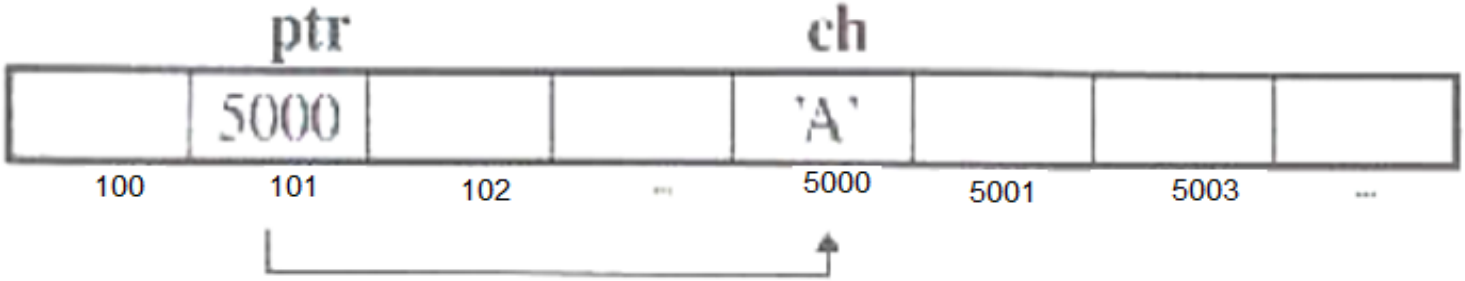
PONTEIRO: SINTAXE

- **Há vários tipos de ponteiro:**
  - **ponteiros para caracteres;**
  - **ponteiros para inteiros;**
  - **ponteiros para ponteiros para inteiros;**
  - **ponteiros para vetores;**
  - **ponteiros para estruturas.**
- **Sintaxe:**

  tipo_do_ponteiro *nome_da_variável;

- **Na declaração de variáveis, o asterisco * é um operador unário que precede o nome da variável e identifica ao compilador que a variável não vai guardar um valor qualquer, mas sim que é um ponteiro que guarda um endereço para aquele tipo especificado.**

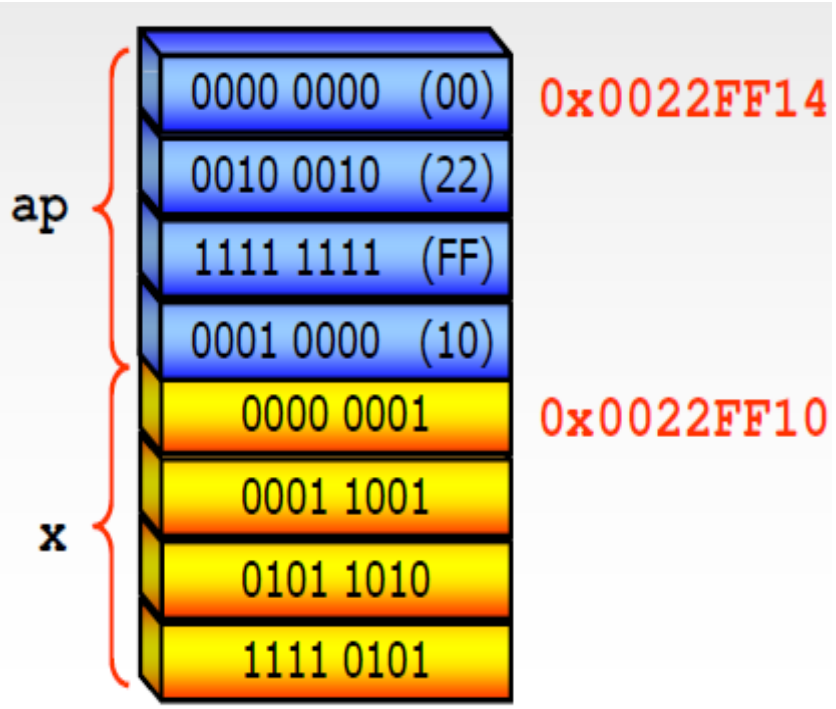
PONTEIRO: EXEMPLOS INICIAIS




```
int     *ap_int
char    *ap_char;
float   *ap_float;
double *ap_double;
```

```
int *ap1, *ap_2, *ap_3;
```

```
int x;
int *ap;     // apontador para inteiros
ap = &x;     // ap aponta para x
```



PONTEIRO: INICIALIZAÇÃO

Como inicializar um ponteiro já que ele guarda um endereço de memória? Como guardar um endereço em um ponteiro?

- **Para inicializar um ponteiro, pode-se realizar diferentes operações de atribuição, com:**
  - **um endereço de uma variável conhecida;**
  - **uma constante *NULL;***
  - **Uma outra variável ponteiro.**

Como saber a posição na memória de uma variável do programa?

PONTEIRO: CONHECENDO O ENDEREÇO DE UMA VARIÁVEL

- **Seria muito difícil saber o endereço de cada variável usada, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho.**
- **Para saber o endereço de uma variável basta usar o operador *&*. Exemplo:**

```
int count = 10;
int *pt;
pt = &count;        //ponteiro pt, recebe o endereço de count
```

- **Foi criado um inteiro *count* de valor 10 e um apontador para um inteiro que é *pt*. A expressão *&count* fornece o endereço de *count*, que é armazenado em *pt*.**
- **O valor de *count* não é alterado, continua 10. Se o valor de *count* for alterado, *pt* continuará apontando para o endereço de *count* que conterá o valor modificado.**

PONTEIRO: EXEMPLO >> INICIALIZAÇÃO

- **Ex.::**

```
int x,*px;
px=&x;  /*a variável px aponta para x */
y=*px;
```

- Para **exibir** o **endereço** de uma **variável** qualquer na tela utiliza-se o **formatador *%p***.
- Um ponteiro pode ser **inicializado** com ***NULL*** (**não** contém **nenhum endereço**), quando **não aponta** para **nenhuma variável**. ***NULL*** é uma **constante** definida na **biblioteca *<stdlib.h>***.
- **Ex.:**

```
char *p;          //declara um ponteiro p para string
p = NULL;    //p não aponta para nenhum endereço
```

PONTEIRO: MANIPULAÇÃO

- **No exemplo:**

```
int count = 10;
int *pt;
pt = &count;        //pt, recebe o endereço de count
```

- **Pode-se alterar o valor de *count* usando *pt*.**
- **Foi usado o operador "inverso" do operador &, que é o operador *.**
- **No exemplo acima, uma vez que *pt=&count;* pode-se utilizar a expressão **pt* como equivalente ao próprio *count*.**
- **Para mudar o valor de *count* para 12, basta acrescentar a seguinte instrução:**

```
*pt =12;        //altera o conteúdo apontado por pt.
```

- **Isto é uma alteração indireta do conteúdo da variável *count*.**

- **Ex.: *main()***

```
{
      int x,*px,*py;
      x=9;
      px=&x;
      py=px;
      printf("x= %d\n",x);      //imprime na tela o 9
      printf("&x= %p\n",&x);    //imprime o endereço de x
      printf("&px= %p\n",&px);          //imprime o endereço de px
      printf("px= %p\n",px);    //imprime o endereço contido em px
      printf("*px= %d\n",*px);          //imprime o valor de px que é 9
      printf("*py= %d\n",*py);          //imprime o valor de py que é 9
      system("pause");
}
```

```
x= 9
&x= 0022FF74
&px= 0022FF70
px= 0022FF74
*px= 9
*py= 9
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
 {
  int i, j;
  int *p, *q;
  j = 3;
  i = *&j;
  p = NULL;          //o ponteiro p não contem nenhum endereço
  printf("Conteudo de i: %d\n", i);
  printf("Conteudo de j: %d\n", *&j);
  printf("Endereço contido em p: %x\n", p);
  p=q;               //p aponta para o mesmo endereço de q
  printf("Endereco de q: %p\n", &q);
  printf("Endereco de p: %p\n", &p);
  printf("Endereco contido em q: %p\n", q);
  printf("Endereco contido em p: %p\n", p);

  system("pause");
  return 0;
}
```

```
Conteudo de i: 3
Conteudo de j: 3
Endereço contido em p: 0
Endereco de q: 0022FF68
Endereco de p: 0022FF6C
Endereco contido em q: 7C800000
Endereco contido em p: 7C800000
```

PONTEIRO PERDIDO

- **Ex.:**

  int *pt, *pt2; //declara 2 ponteiros para um inteiro
  char *temp; //declara um ponteiro para caracteres.

- **Com estes ponteiros foram declarados, mas não foram inicializados, logo eles apontam para um lugar indefinido.**
- **Manipular ponteiros nessa situação pode causar instabilidade, pois pode significar que se está manipulando, inclusive, porção da memória reservada ao SO ou estão sendo referenciados endereços que excedem o limite da memória.**

PONTEIRO PERDIDO

- **Consequências:**
  - **Este erro é chamado de ponteiro perdido;**
  - **É um dos erros mais difíceis de se encontrar, pois a cada vez que a operação com o ponteiro é utilizada, poderá estar sendo lido ou gravado em posições desconhecidas da memória;**
  - **Pode acarretar em sobreposições sobre áreas de dados ou mesmo área do programa na memória;**
  - **Usar um ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior. Isto é perigoso e não desejável.**

ATENÇÃO!
O ponteiro deve ser inicializado (para algum lugar conhecido) antes de ser usado ou para *NULL*.

```c
#include <stdio.h>
int main ()
{
      int num,valor;
      int *p;
      num=55;
      p=&num;
      /* Pega o endereco de num */
      valor=*p;
      /* Valor e igualado a num de uma maneira indireta */
      printf ("\n\n%d\n",valor);
      printf ("Endereco para onde o ponteiro aponta: %p\n",p);
      printf ("Valor da variavel apontada: %d\n",*p);
      return(0);
}
```

```c
#include <stdio.h>
int main ()
{
      int num,*p;
      num=55;
      p=&num;      /* Pega o endereco de num */
      printf ("\nValor inicial: %d\n",num);
      *p=100; /* Muda o valor de num de uma maneira indireta */
      printf ("\nValor final: %d\n",num);
      return(0);
}
```

PONTEIRO: OPERAÇÕES>> IGUALAR

- **Igualar dois ponteiros:**
  - **se há dois ponteiros p1 e p2, de mesmo tipo, pode-se fazer uma atribuição para ele, ou seja, fazer *p1=p2;*. Isto significa que *p1* aponta para o mesmo endereço que *p2*.**
  - **Cuidado pois os ponteiros devem ser iguais em tipo.**
- **Para que uma variável apontada por *p1* tenha o mesmo conteúdo de uma a variável apontada por *p2* deve-se fazer:**

**p1=*p2;*

ATENÇÃO! Isto não significa que *p1* e *p2* estejam apontando para o mesmo endereço! Mas que o conteúdo apontado por **p2* foi atribuído ao endereço para o qual *p1* aponta.

PONTEIRO: OPERAÇÕES >> INCREMENTO/DECREMENTO

- **quando um ponteiro é incrementado ele passa a apontar para o próximo endereço do mesmo tipo para o qual o ponteiro aponta.**
- **Se há um ponteiro para um inteiro e este ponteiro é incrementado ele passa a apontar para o próximo inteiro.**
- **Se você incrementa um ponteiro *char** ele é deslocado 1 byte para frente na memória;**
- **Se você incrementa um ponteiro *double** ele é deslocado 8 bytes para frentes na memória. O decremento funciona semelhantemente.**
- **Neste tipo de operação o conteúdo do ponteiro é alterado, pois sofre uma auto atribuição (auto modificação).**

ATENÇÃO! Operações de incremento e decremento modificam o conteúdo do ponteiro que passa a apontar para uma nova posição.

PONTEIRO: OPERAÇÕES >> INCREMENTO/DECREMENTO

- **Se p é um ponteiro, as operações de incremento e decremento são escritas como:**

p++;
p--;

- **As operações acima estão sendo realizadas sobre os endereços referenciados pelos ponteiros e não sobre os conteúdos das variáveis para as quais os ponteiros apontam.**
- **Os incrementos e decrementos dos endereços possuem procedência sobre o * e sobre as operações matemáticas e são avaliados da direita para a esquerda.**
- ***Ex.:***

ptr++;

//Supondo que ptr foi declarado como um ponteiro para double, em ptr tem o endereço 2112, passará a apontar para endereço 2120 (8 bytes a frente) e não para 2113.

PONTEIRO: OPERAÇÕES >> INCREMENTO/DECREMENTO

- **O incremento ou decremento de ponteiros está relacionado ao tipo de dado do ponteiro. No exemplo acima, se p é um ponteiro para inteiro (ou seja, *int *p;*), e p armazena o endereço 100, então p++ faz p armazenar o endereço 104 (pois um inteiro ocupa 4 bytes).**
- **Pode-se utilizar parênteses para "quebrar" a precedência do operador *.**
- **Com a utilização de parênteses o conteúdo que é apontado é incrementado, porque os operadores * e ++ são avaliados da direita para esquerda, sem eles o *px* (seu endereço) seria incrementado.**

```
*px++;        // incrementa uma posição na memória e depois mostra o conteúdo.

(*px)++;//conteúdo de px é incrementado de +1, equivale a: *px+=1

*(px--);      // mesma coisa de *px--
```

PONTEIRO: OPERAÇÕES

- **O operador * tem maior precedência que as operações aritméticas.**
- **Na instrução, abaixo, pega o conteúdo que está no endereço que *px* aponta e soma 1 ao seu conteúdo. Desta forma, se *px* apontar para um endereço que guarda o valor 10, este 10 será somado com uma unidade e o seu resultado será armazenado em *y*.**

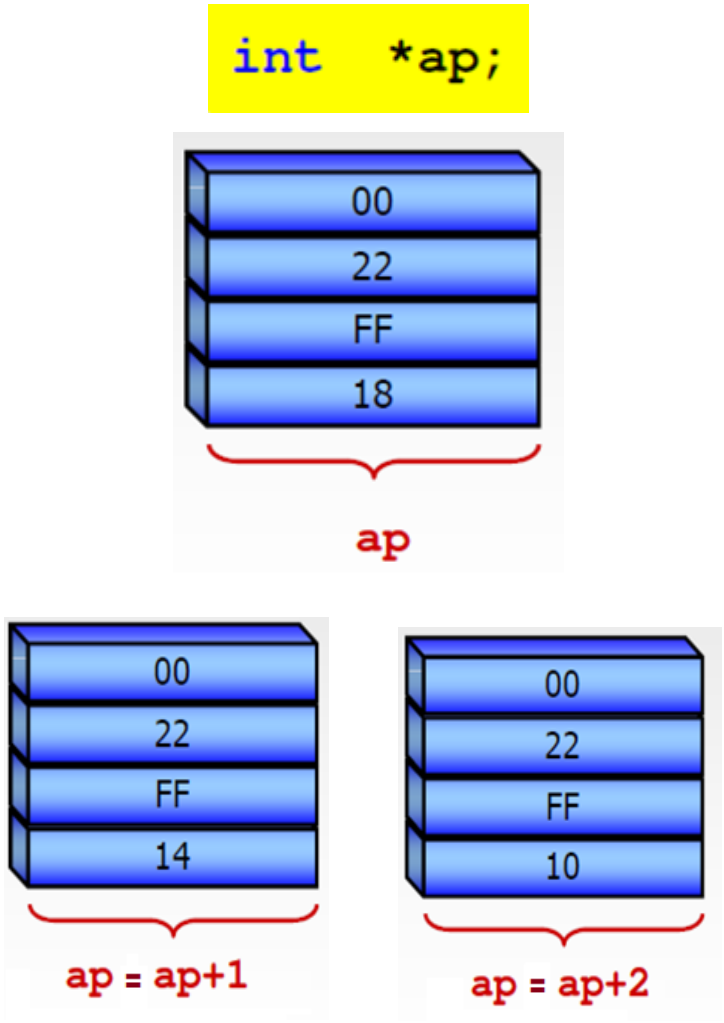
  *y = *px+1;*

- **Na instrução: *y=*(px+1);* o endereço que está dentro do ponteiro será incrementado em uma unidade e, após este deslocamento, o conteúdo da dessa próxima posição da memória será atribuído a *y*.**
- **Soma e subtração de inteiros com ponteiros também podem ser feitas com ponteiros. Assim, pode-se fazer:**

  p=p+15; ***ou*** *p+=15;*

PONTEIRO: OPERAÇÕES >> SOMA E SUBTRAÇÃO

- Há um conjunto limitado de operações aritméticas que pode ser realizado com ponteiros.
- No caso de uma operação de adição ou subtração com ponteiro estaremos manipulando para endereços mais acima ou mais abaixo que o endereço atual (deslocamentos).
- Nessas operações deve-se tomar cuidado para não invadir outras áreas de memória de outras variáveis ou de outros programas.
- Este tipo de operação não modifica o endereço para qual o ponteiro aponta, mas apenas calcula a nova posição apontada. Ou seja, não modifica o endereço que está dentro do ponteiro. Não ser que tenha sido feita uma atribuição



PONTEIRO: OPERAÇÕES >> SOMA E SUBTRAÇÃO

- **Exemplos:**

ptr + 1; // endereço de onde *ptr* apontava + *sizeof(tipo),* deslocado de uma unidade para frente (endereço acima, em relação ao endereço atual).

ptr + 2; //endereço de onde *ptr* apontava, deslocado de duas unidades para frente (endereços acima, em relação ao endereço atual).

ptr – 3; //endereço de onde *ptr* apontava, deslocado de duas unidades para trás (endereços abaixo, em relação ao endereço atual).

- **Nessas operações não foram realizadas atribuições sobre os ponteiro, apenas foram calculados os deslocamentos, sempre em relação ao tipo de dado associado ao ponteiro.**

PONTEIRO: OPERAÇÕES >> SOMA E SUBTRAÇÃO

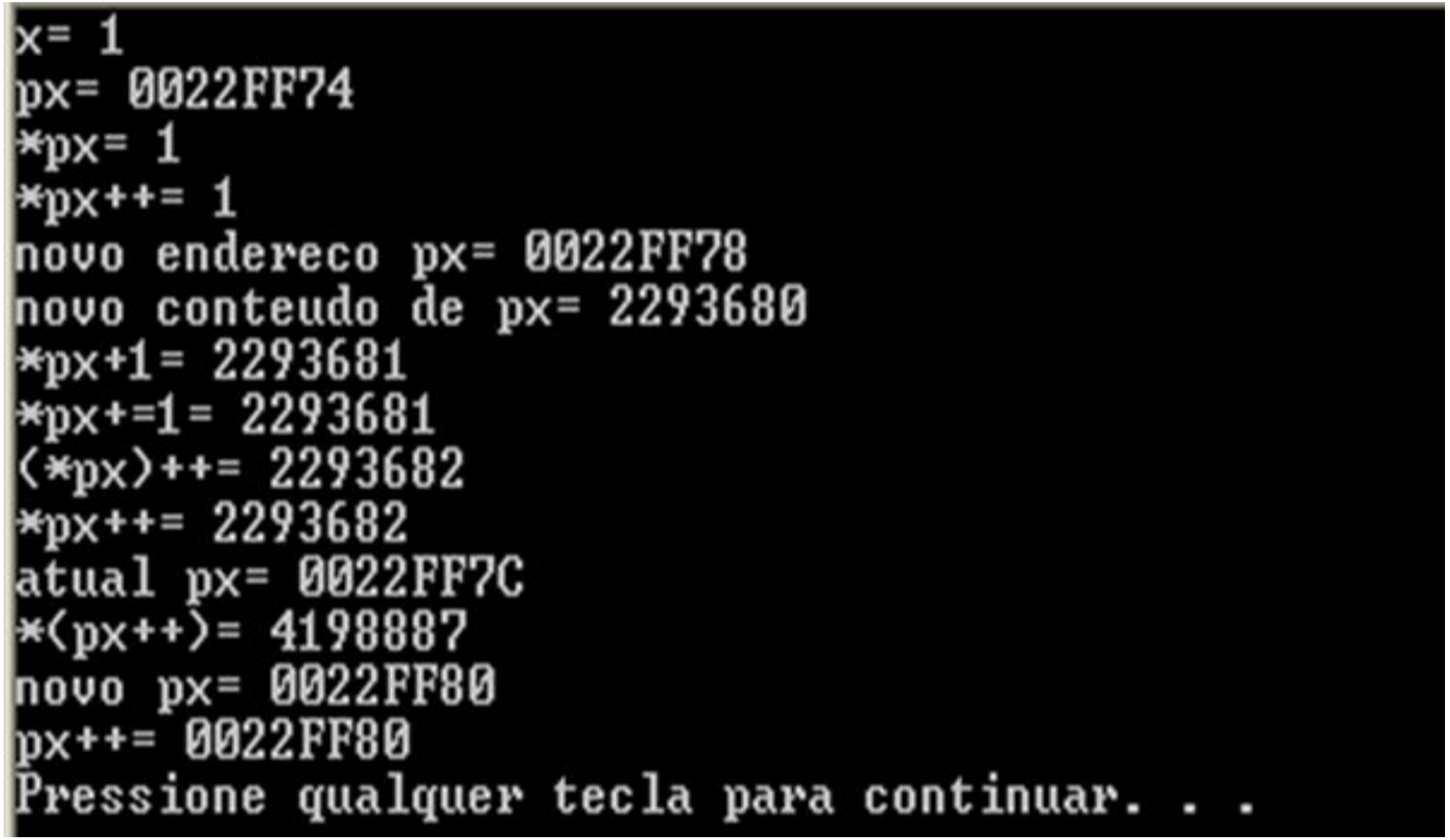
```
main()
{
      int v[] = {10, 20, 30};
      printf("%p\n", v);
      printf("%p\n", v+2);
      printf("%d\n", *(v+2));
}
```

- **Para usar o conteúdo apontado pelo ponteiro 15 posições adiante, basta fazer: *(p+15). A subtração funciona da de modo similar.**

PONTEIRO: OPERAÇÕES >> COMPARAÇÃO

- **Para saber se dois ponteiros são iguais ou diferentes utiliza-se os operadores == e !=.**
- **Esta comparação está relacionada aos endereços que esses ponteiros referenciam, ou seja, se apontam ou não para a mesma posição da memória.**
- **No caso dos operadores relacionais (>, <, >= e <=) compara-se qual ponteiro aponta para uma posição mais alta/baixa na memória.**
- **Então, uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória.**
- **A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:** `p1>p2`
- **Há entretanto operações que você não pode efetuar em um ponteiro:**
  - **dividir ou multiplicar ponteiros;**
  - **adicionar dois ponteiros;**
  - **adicionar ou subtrair *floats* ou *doubles* de ponteiros.**

```c
main(){
    int x,*px;
    x = 1;
    px = &x;
    printf("x= %d\n",x);              //imprime o valor de x --> 1
    printf("px= %p\n",px);            //endereço de px
    printf("*px= %d\n",*px);          //conteúdo apontado pelo endereço de px
    printf("*px++= %d\n",*px++);      //usa o endereço de px e depois incrementa
    printf("novo endereco px= %p\n",px);
    printf("novo conteudo de px= %d\n",*px);
    printf("*px+1= %d\n",*px+1);      //incrementa o valor referenciado por px
    printf("*px+=1= %d\n",*px+=1);    //incrementa o valor referenciado por px
    printf("(*px)++= %d\n",*px=*px+1);  //incrementa o valor referenciado por px
    printf("*px++= %d\n",*px++);          //usa o conteudo de px e depois incrementa
    printf("atual px= %p\n",px);
    printf("*(px++)= %d\n",*(px++));      //usa o novo conteudo de px, mostra seu valor e depois incrementa
    printf("novo px= %p\n",px);
    printf("px++= %p\n",px++);        //usa px e depois incrementa o endereco
    system("pause");
}
```


x= 1
px= 0022FF74
*px= 1
*px++= 1
novo endereco px= 0022FF78
novo conteudo de px= 2293680
*px+1= 2293681
*px+=1= 2293681
(*px)++= 2293682
*px++= 2293682
atual px= 0022FF7C
*(px++)= 4198887
novo px= 0022FF80
px++= 0022FF80
Pressione qualquer tecla para continuar. . .

PRIMEIROS PASSOS

Quais das seguintes instruções são corretas para declarar um ponteiro?

a) int _ptr x;
b) int *ptr;
c) *int ptr;
d) *x;

Qual é a maneira correta de referenciar **ch**, assumindo que o endereço de **ch** foi atribuído ao ponteiro **indica**?

a) *indica;
b) int *indica;
c) *indic;
d) ch
e) *ch;

Na expressão **float *pont**; o que é do tipo float?

a) a variável pont.
b) o endereço de pont.
c) a variável apontada por pont.
d) nenhuma das anteriores.

Assumindo que o endereço de **num** foi atribuído a um ponteiro **pnum**, quais das seguintes expressões são verdadeiras?

a) num == &pnum
b) num == *pnum
c) pnum == *num
d) pnum == &num

Assumindo que queremos ler o valor de **x**, e o endereço de **x** foi atribuído a **px**, a instrução seguinte é correta? Por que?

```
scanf ( "%d", *px );
```

PONTEIRO PARA PONTEIRO

- **Um ponteiro para um ponteiro é uma forma de indicação múltipla.**
- **Em um ponteiro "normal", o conteúdo do ponteiro é o endereço que aponta para o endereço que contém o valor desejado.**
- **Quando se tem ponteiro para ponteiro, o primeiro ponteiro contém o endereço do segundo ponteiro, e este segundo ponteiro é que aponta para o endereço que contém o valor desejado.**
- **Exemplo:**
- *float **b; //b é um ponteiro para um ponteiro float.*

```
// apontador para apontador
int     **ap_ap_int;
```

```
10
Endereþo de x: 0022FF74
Endereþo de p: 0022FF70
Endereþo de q: 0022FF6C
Endereþo contido em p: 0022FF74
Endereþo contido em q: 0022FF70
```

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
    int x,*p,**q;
    x=10;   // x recebe o valor inteiro 10
    p=&x;   //p recebe o endereço de x
    q=&p;   // endereço de p é recebido por q
    printf("%d",**q);   //imprime o valor 10, sendo que q tem o endereço de p e p tem o endereço de x.
    printf("\nEndereço de x: %p",&x);
    printf("\nEndereço de p: %p",&p);
    printf("\nEndereço de q: %p",&q);
    printf("\nEndereço contido em p: %p",p);
    printf("\nEndereço contido em q: %p\n",q);
    system("PAUSE");
}
```

PONTEIRO PARA PONTEIRO...

- No C pode-se declarar ponteiros para ponteiros para ponteiros e assim por diante. Para fazer isto basta aumentar a quantidade de * (asteriscos) na declaração.
- Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas ou mais vezes, de acordo com a quantidade de ponteiros utilizada na declaração. Ex.:

```
#include <stdio.h>

int main()
{
    float fpi = 3.1415, *pf, **ppf;
    pf = &fpi;                    /* pf armazena o endereco de fpi */
    ppf = &pf;                    /* ppf armazena o endereco de pf */
    printf("%f", **ppf);         /* Imprime o valor de fpi */
    printf("%f", *pf);            /* Tambem imprime o valor de fpi */
    return(0);
}
```

PRIMEIROS PASSOS

Qual é a instrução que deve ser adicionada ao programa seguinte para que ele trabalhe corretamente?

```
main ( ) {
        int  j,  *pj;
        *pj = 3;
}
```

Assumindo que o endereço da variável **x** foi atribuído a um ponteiro **px**, escreva uma expressão que não usa **x** e divida **x** por 5.

Qual o valor das seguintes expressões:

```
int  i = 3,  j = 5;
int  *p = &i,  *q = &j;
```

a) p == &i

b) *p - *q

c) **&p