Estruturas de Dados 1

05 – Ponteiros

Antonio Angelo de Souza Tartaglia

angelot@ifsp.edu.br

Conceito de Ponteiros - Apresentação

- Toda informação que manipulamos dentro de um programa (esteja ela armazenada em uma variável, vetor, estrutura, etc.), obrigatoriamente está armazenada na memória do computador.

- Quando criamos uma variável, o computador reserva um espaço de memória onde podemos guardar o valor associado a ela. Ao nome que damos a ela o computador associa o endereço do espaço que ele reservou na memória para guarda-la.

- De modo geral, interessa ao programador saber o nome das variáveis. Já o computador precisa saber onde elas estão na memória, ou seja, precisa de seus **endereços**.

Estrutura de Dados 1

Conceito de Ponteiros – Variáveis tipos primitivos

- Uma variável é uma posição de memória reservada e nomeada pelo compilador;
- É usada para guardar um valor que pode ser modificado pelo programa;
- Exemplo de declaração:

```
int a = -5;
char b = 'x';
float c = 12.23;
```

Conceito de Ponteiros – Variáveis tipos primitivos

- Cada variável possui
  - Um nome;
  - Um endereço de memória ou referência;
  - Um valor;
- Exemplo:

```
int x = 10;
```

- Nome: x;
- Endereço de memória: 0xABF236;
- Valor: 10.

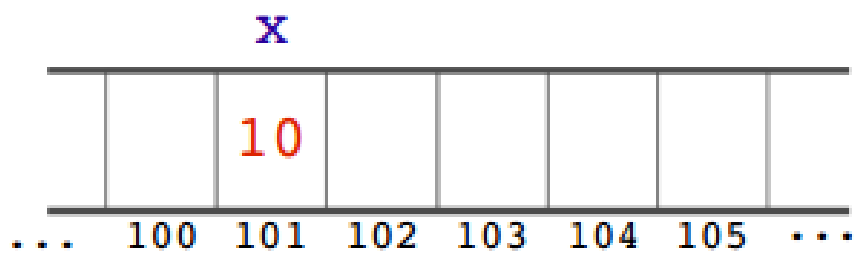
Posição de memória representada em notação Hexadecimal

Estrutura de Dados 1

Conceito de Ponteiros – Variáveis alocadas em Memória

- Imagine a memória do computador como sendo um grande vetor de células consecutivamente numeradas;
- Para acessar o endereço de uma variável utilizamos o operador &.
- Exemplo:

```
int x = 10;          &x = 101
```



Estrutura de Dados 1

Conceito de Ponteiros – O que são:

- Ponteiros são um tipo especial de variável que permite armazenar endereços de memória ao invés de dados numéricos, como os tipos primitivos `int`, `float` e `double`, ou caracteres como o tipo `char`.

- Por meio de ponteiros, podemos acessar o endereço de uma variável e manipular o valor que está armazenado lá dentro. Eles são uma ferramenta extremamente útil dentro da Linguagem C.

Conceito de Ponteiros – O que são:

- Apesar de suas vantagens muitos programadores têm medo de usá-los porque isso trás muitos perigos:
  - Ponteiros permitem que um programa acesse objetos que não foram explicitamente declarados com antecedência e, consequentemente permitem grande variedades de erros de programação;
  - Outro grande problema, é que eles podem *ser apontados para endereços* não utilizados ou para dados dentro da memória que estão sendo usados para outro propósito.
  - Apesar dos perigos no uso dos ponteiros, seu poder é tão grande que existem tarefas que são difíceis de ser implementadas sem sua utilização.

Estrutura de Dados 1

Conceito de Ponteiros – O que são:

- O correto entendimento e uso de ponteiros é crítico para uma programação bem sucedida em C. Há três razões para isso:
  - Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
  - Eles são usados para suportar as rotinas de alocação dinâmica de memória em C;
  - O uso de ponteiros pode aumentar a eficiência de certas rotinas.
- Ponteiros são um dos aspectos **mais fortes e mais perigosos** de C:
  - Por exemplo, ponteiros não inicializados, podem provocar uma quebra do sistema.
- Talvez pior: é fácil usar ponteiros incorretamente, ocasionando erros que são muito difíceis de encontrar.

Conceito de Ponteiros – O que são:

- Um ponteiro é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de uma outra variável na memória. Se a variável contém o endereço de outra, então dizemos que esta variável (a que contém o endereço), aponta para a outra.

- Se uma variável irá conter um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste no tipo de base (`int`, char, `float`, tipo criado pelo programador, etc), um `*` e o nome da variável.

- O tipo base do ponteiro define que tipo de variáveis o ponteiro pode apontar:
  - Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória. No entanto toda a aritmética de ponteiros é feita por meio do tipo base, assim é importante declarar o ponteiro corretamente.

Estrutura de Dados 1

Tipos de dados definidos no padrão ANSI

Tipo	Tamanho	Intervalo
Char	8 bits	-127 a 128
Unsigned char	8 bits	0 a 255
signed char	8 bits	-127 a 128
short int	16 bits	-32768 a 32767
unsigned shor int	16 bits	0 a 65535
signed short int	16 bits	-32768 a 32767
int	32 bits	-2.147.483.648 a 2.147.483.647
signed int	32 bits	-2.147.483.648 a 2.147.483.647
unsigned int	32 bits	0 a 4.294.967.295
long int	32 bits	-2.147.483.648 a 2.147.483.647
signed long int	32 bits	-2.147.483.648 a 2.147.483.647
unsigned long int	32 bits	0 a 4.294.967.295
float	32 bits	3,4x10-38 a 3,4x10+38
Double	64 bits	1,7 x10-308 a 1,7 x10+308
Long Double	80 bits	3,4 x104932 a 1,1 x104932

Estrutura de Dados 1

Tipos de dados – mapeamento de ocupação em memória

100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125
126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151
152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177
178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203
204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229
230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281
282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307
308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333
334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359
360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385
386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411
412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437
438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463
464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489
490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515
516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541
542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567
568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593
594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619

int - 32 bits

char - 8 bits

float - 32 bits

double - 64 bits

struct - char+int

Declarando ponteiros

Na Linguagem C, um ponteiro pode ser declarado para qualquer tipo de variável (`int`, `char`, `double`, etc), inclusive para aquelas criadas pelo programador (`struct`, etc).

Sintaxe:

<TIPO> *<NOME_DA_VARIÁVEL_PONTEIRO>;

Exemplos de declaração:

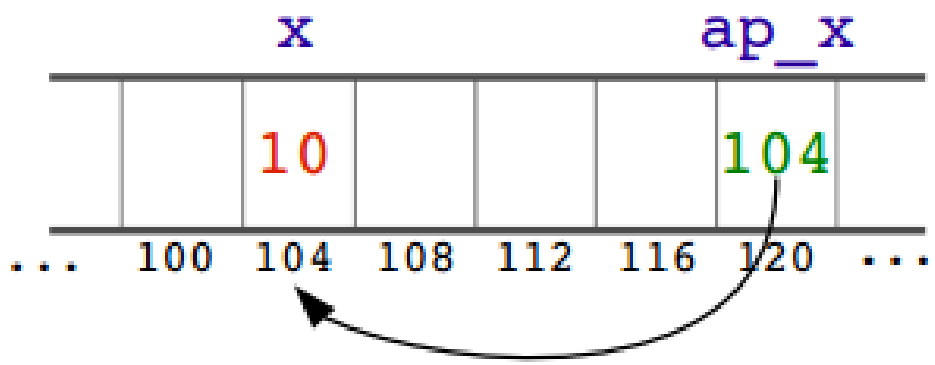
```
int     *ap_int;
char    *ap_char;
float   *ap_float;
double *ap_double;
```

Estrutura de Dados 1

Ponteiros

- Ação do operador &, capturando o endereço de uma variável:

```
int x = 10;
int *ap_x;
ap_x = &x; // ap_x aponta para x
```

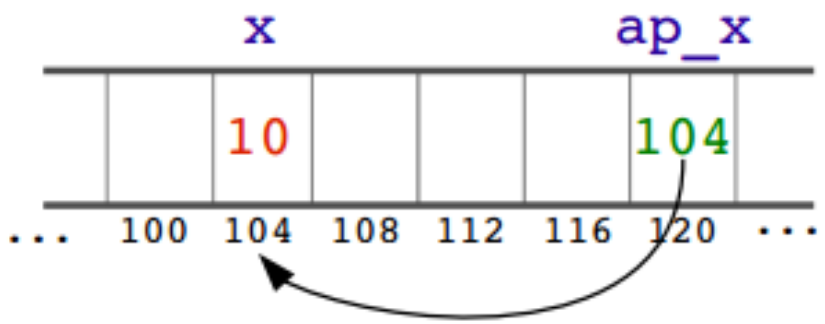


Estrutura de Dados 1

Ponteiros

- O operador * acessa o conteúdo do endereço apontado pelo ponteiro:

```c
int x = 10;
int *ap_x;
ap_x = &x;                  //ap_x aponta para x

printf("%d\n", *ap_x);      // 10
```



Estrutura de Dados 1

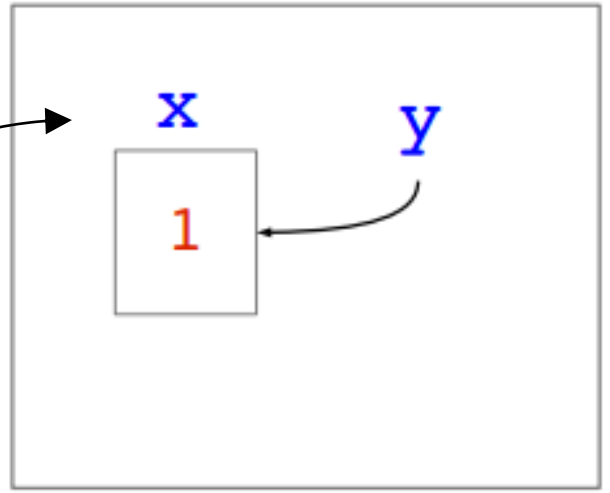
Exemplo 1

```c
#include <stdio.h>

void main(){
int x;    //declaração da variável x
int *y;   //declaração da variável de ponteiro y

x = 1;    //atribui o valor 1 á variável x
y = &x;   //atribui o endereço da variável x ao ponteiro y
*y= 2;    //o conteúdo que está no endereço em y, recebe 2

printf("\n %d \n",x);
}
```

Estrutura de Dados 1

Exemplo 1

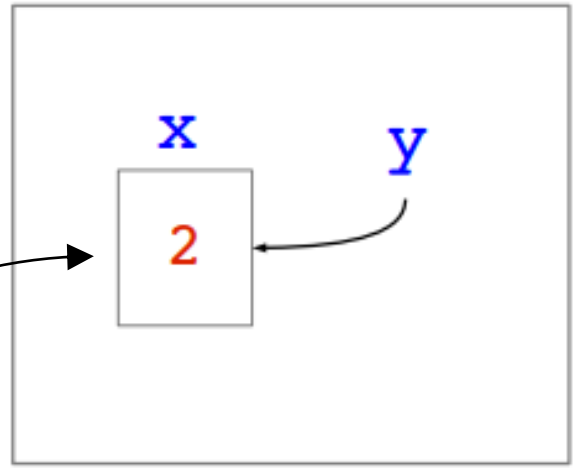
```
1  #include <stdio.h>
2
3  void main(){
4  int x;
5  int *y;
6
7  x = 1;
8  y = &x;
9  *y= 2;
10
11 printf("\n %d \n",x);
12 }
```



Estrutura de Dados 1

Exemplo 1

```
1   #include <stdio.h>
2
3   void main(){
4   int x;
5   int *y;
6
7   x = 1;
8   y = &x;
9   *y= 2;
10
11  printf("\n %d \n",x);
12  }
```



Estrutura de Dados 1

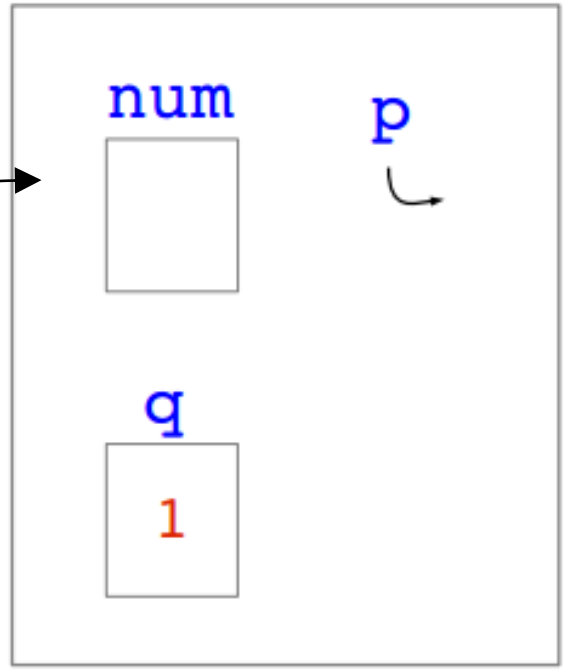
Exemplo 2

```c
#include <stdio.h>
void main(){
    int num, q=1; //declara num e q como int
    int *p;       //declara ponteiro p

    num = 50;     // atribui 50 a num
    p = &num;     // atribui o endereço de num a p
    q = *p;       // atribui o conteúdo do endereço que está em p a q

    printf("%d \n",q);
}
```

Estrutura de Dados 1

Exemplo 2

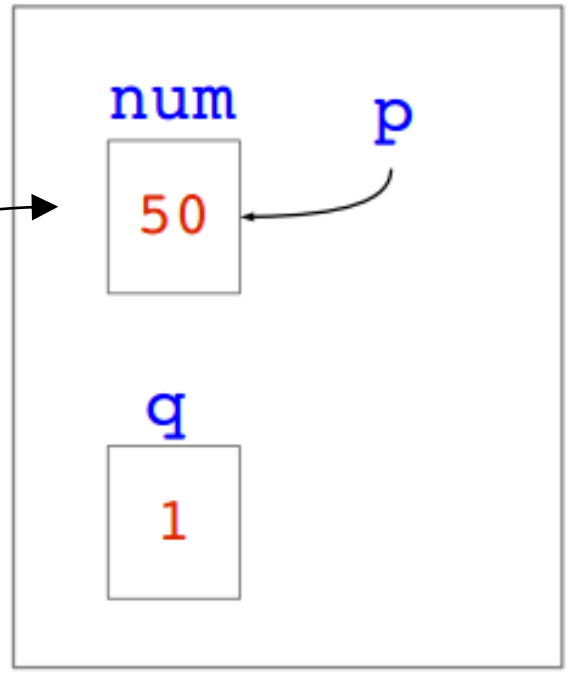
```c
#include <stdio.h>
void main(){
    int num, q=1;
    int *p;
    num = 50;
    p = &num;
    q = *p;
    printf("%d \n",q);
}
```



Estrutura de Dados 1

Exemplo 2

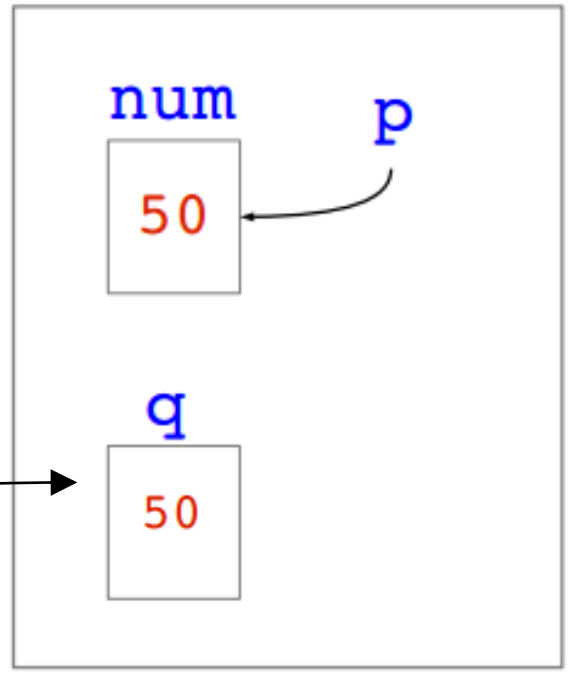
```c
#include <stdio.h>
void main(){
    int num, q=1;
    int *p;
    num = 50;
    p = &num;
    q = *p;
    printf("%d \n",q);
}
```



Estrutura de Dados 1

Exemplo 2

```c
#include <stdio.h>
void main(){
    int num, q=1;
    int *p;
    num = 50;
    p = &num;
    q = *p;
    printf("%d \n",q);
}
```



Estrutura de Dados 1

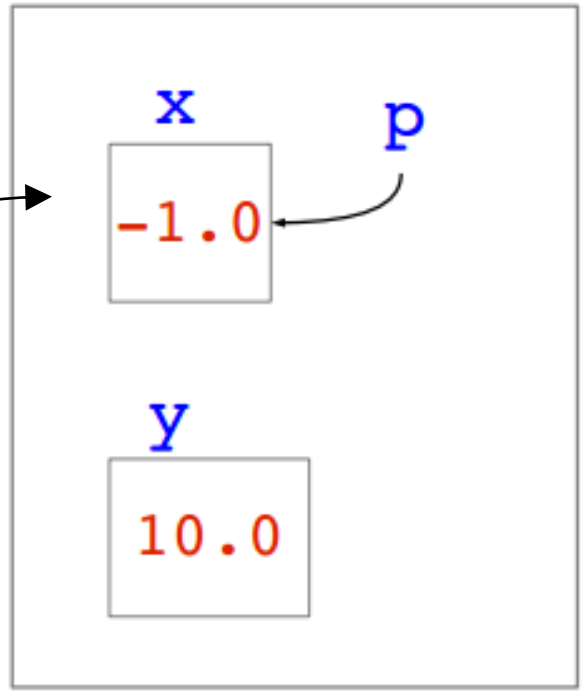
Exemplo 3

```c
#include <stdio.h>
void main(){
    float x = -1.0,  y =10.0; //declara 2 floats
    float *p = &x;       //declara ponteiro p e atribui endereço de x
    y = (*p) + 5.0;      //atribui conteúdo do endereço p + 5, à y
    printf("%f \n",y);
}
```

Estrutura de Dados 1

Exemplo 3

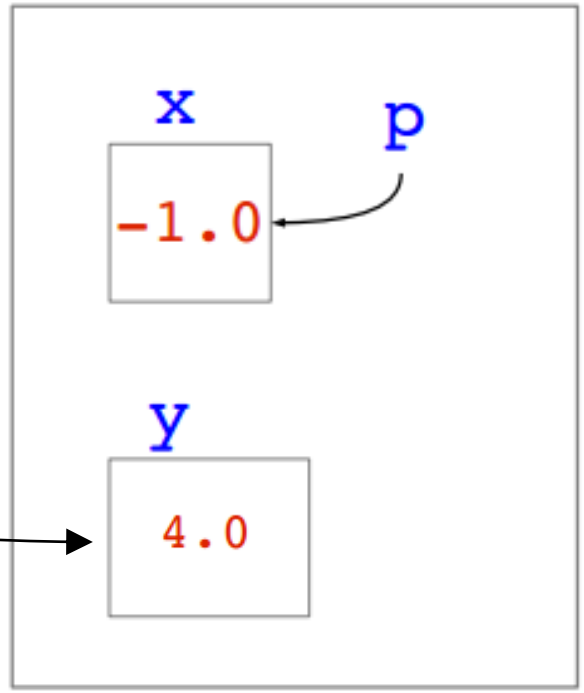
```
1  #include <stdio.h>
2  void main(){
3      float x = -1.0,  y =10.0;
4      float *p = &x;
5
6     y = (*p) + 5.0;
7
8     printf("%f \n",y);
9  }
```



Estrutura de Dados 1

Exemplo 3

```c
#include <stdio.h>
void main(){
    float x = -1.0,  y =10.0;
    float *p = &x;

   y = (*p) + 5.0;

   printf("%f \n",y);
}
```



Inicializando o Ponteiro

- Ponteiros não inicializados apontam para um lugar indefinido;

```
int *p;
```

Declarando o ponteiro sem inicializa-lo, a posição em que ele ocupará na memória pode conter informações (lixo), que talvez causem problemas...

- Um ponteiro pode ter um valor especial NULL, que é o endereço de “lugar nenhum”;

```
int *p = NULL;
```

Dessa forma, garantimos que não haverá conteúdos indesejados no ponteiro declarado

- Com isso o ponteiro “p”, aponta para uma região de memória que não é válida, ou seja que não existe, não está na memória.
- A constante NULL está definida na biblioteca `stdlib.h`. Trata-se de um valor reservado que indica que aquele ponteiro aponta para uma posição de memória inexistente. O valor da constante NULL é **ZERO** na maioria dos computadores.

Estrutura de Dados 1

Atribuições com ponteiros

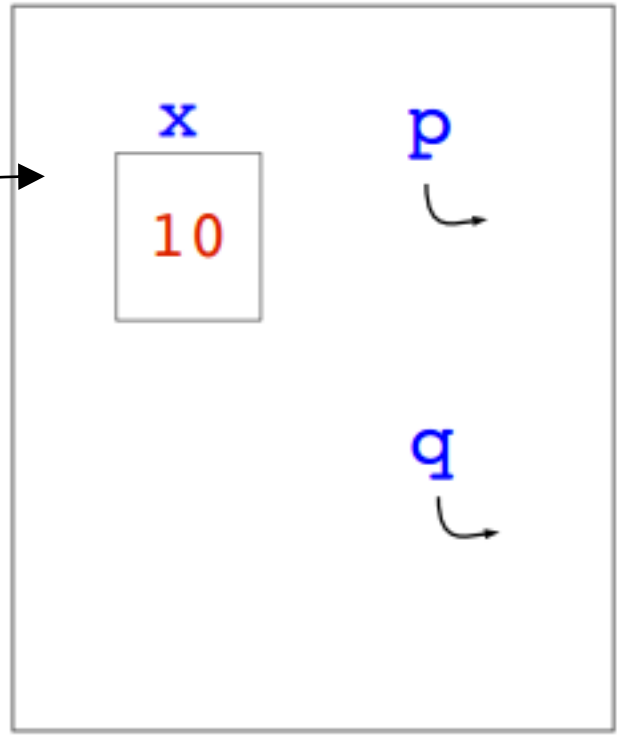
- Como qualquer variável, um ponteiro pode passar seu valor para outro ponteiro. Porém esta operação só pode ser executada com ponteiros do mesmo tipo:

```c
#include <stdio.h>
    void main(){
    int x = 10;//Declara x
    int *p,*q; //Declara dois ponteiros p e q
    p = &x;    //Atribui o endereço de x à p
    q = p;     //Atribui o valor de p (que é um endereço) à q
    printf("%d \n",*q); //imprime o conteúdo do endereço que está em q
}
```

Estrutura de Dados 1

Atribuições com ponteiros

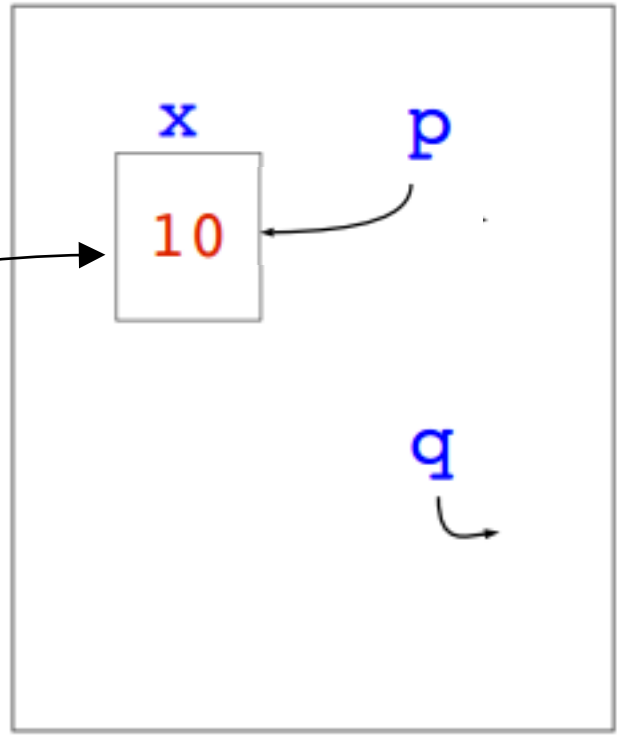
```c
#include <stdio.h>
    void main(){
    int x = 10;
    int *p,*q;

    p = &x;
    q = p;
    printf("%d \n",*q);
}
```



Estrutura de Dados 1

Atribuições com ponteiros

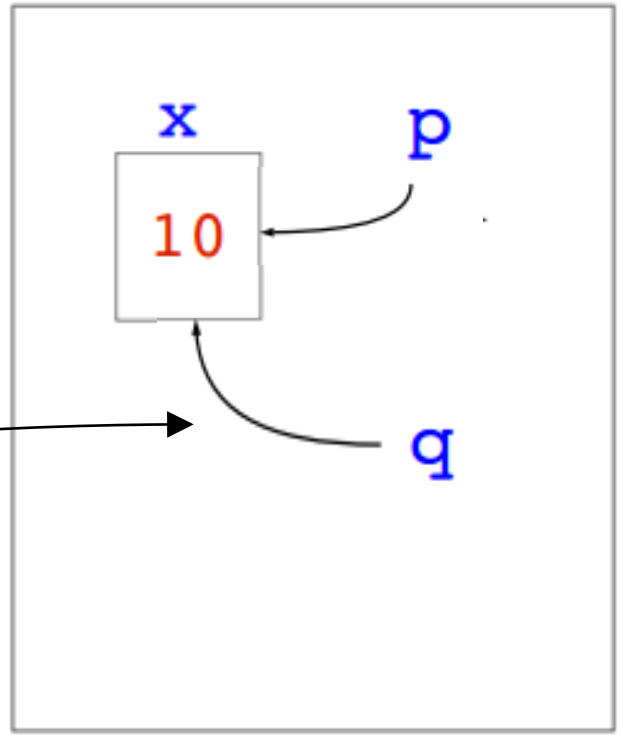
```c
#include <stdio.h>
    void main(){
    int x = 10;
    int *p,*q;

    p = &x;
    q = p;
    printf("%d \n",*q);
}
```



Estrutura de Dados 1

Atribuições com ponteiros

```c
#include <stdio.h>
    void main(){
    int x = 10;
    int *p,*q;

    p = &x;
    q = p;
    printf("%d \n",*q);
}
```



Estrutura de Dados 1

Atribuições com ponteiros

- Erro de tipo na atribuição de ponteiros:

```c
#include <stdio.h>
void main(){
    int x = 10;             //Declara inteiro x e atribui conteúdo
    char y = "B";           //Declara char y e atribui conteúdo
    int *p;                 //Declara o ponteiro p
    p = &x;                 //Atribui o endereço de x à p
    printf("%d \n",*p);     //imprime o conteúdo do endereço que está em p
    p = &y;                 //ERRO!! O compilador até executa, mas haverá erro
}                           //quando o programa acessar a região de memória
```

Expressões com ponteiros

- Operador de Incremento:
  - Quando um ponteiro é incrementado, ele aponta para a posição de memória do próximo elemento do seu tipo base.
- Exemplo:
  - Considere `ap_int` um ponteiro para o tipo `int` com o valor atual 200;
  - De acordo com a tabela do padrão ANSI, o tipo `int` possui 4 bytes;

```
ap_int++;
```

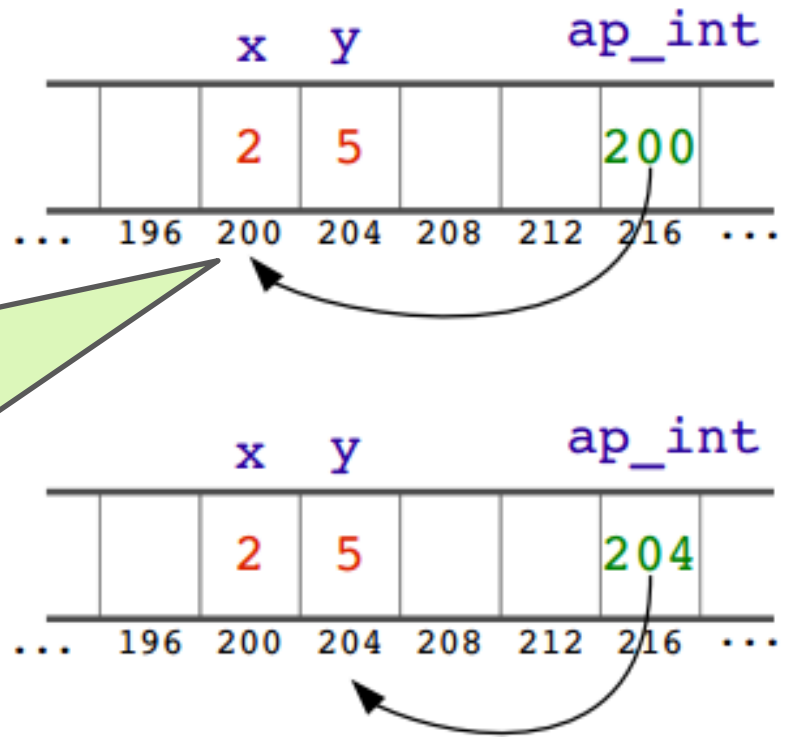
- Após a expressão, `ap_int` terá o valor de 204.

Sobre o valor de endereço armazenado por um ponteiro, podemos apenas somar e subtrair valores **inteiros**.
São as únicas operações que podem ser realizadas, não sendo possível a multiplicação e a divisão.
Da mesma forma, também não é possível a soma ou subtração de um endereço à outro, isso não faria sentido...

Expressões com ponteiros

- Operador de incremento:
- Exemplo:

```
ap_int++;
```



Quando declaramos os inteiros x e y, o compilador reserva 4 bytes para cada variável. Assim nas operações de soma e subtração são adicionados/subtraídos 4 bytes por incremento/decremento, pois esse é o tamanho de um inteiro na memória.

As operações de adição e subtração no endereço permitem avançar ou retroceder nas posições de memória do computador. Esse tipo de operação é bastante útil quando trabalhamos com vetores. Além disso todas as operações de adição e subtração no endereço devem ser inteiras, afinal não é possível andar “meia” posição na memória, por exemplo...

Expressões com ponteiros

- Operador de decremento:
  - Cada vez que é decrementado, o ponteiro irá apontar para a posição do elemento anterior;

- Exemplo:

- Considere `ap_int` um ponteiro para o tipo int com o valor atual 200;

```
ap_int--;
```

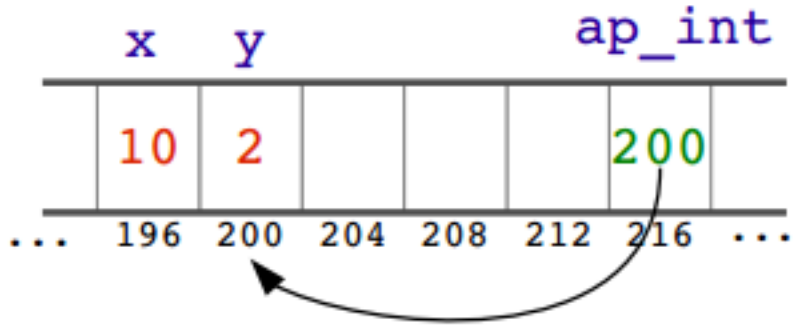
- Após, a expressão `ap_int` terá o valor 196

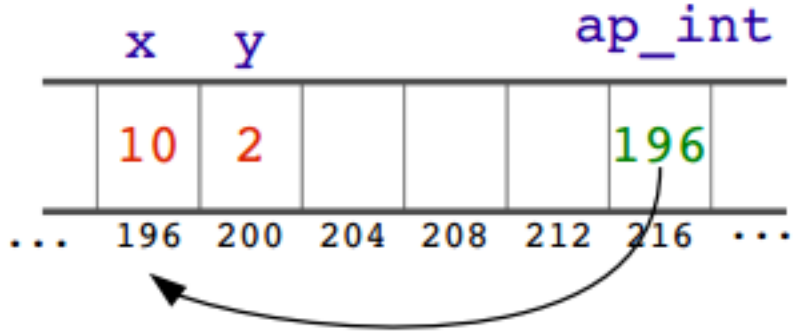
Expressões com ponteiros

- Operador de decremento:

- Exemplo:

```
ap_int--;
```





As operações de adição e subtração no endereço dependem do tipo de dado para o qual o ponteiro aponta. Em outras palavras, o resultado sempre dependerá do tamanho do tipo de dado apontado

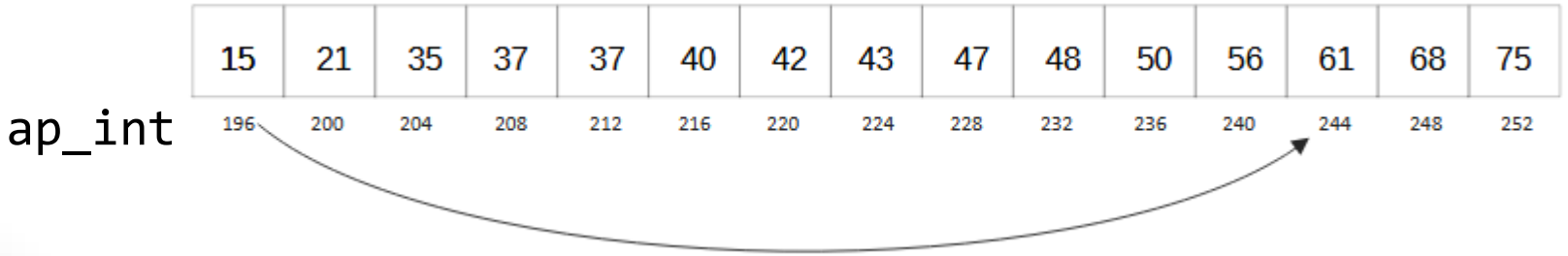
Expressões com ponteiros

- Operador de adição e subtração:
- Além das operações de incremento e decremento é possível somar ou subtrair inteiros de ponteiros. Considere que `ap_int` aponta para o início dos valores inteiros, no endereço 196;
- Exemplo:

```
ap_int = ap_int + 12;
```

ap_int = 196
ap_int = 196 + 12 * 4
ap_int agora aponta para 244

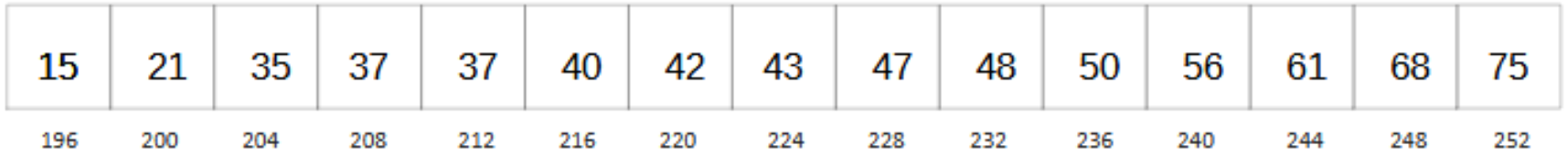
- Faz `ap_int` apontar para o décimo segundo elemento do tipo de `ap_int` , que tem 4 bytes pois `ap_int` aponta para tipos inteiros, adiante do elemento que ele está atualmente apontando.



Comparação com ponteiros

- A Linguagem C permite comparar os endereços de memória armazenados por dois ponteiros utilizando uma expressão relacional. Por exemplo, os operadores “==“ e “!=“ são usados para saber se dois ponteiros são iguais ou diferentes;

- Já os operadores “>”, “<“, “>=“, “<=“ são usados para saber se um ponteiro aponta para uma posição mais adiante na memória do que outro. Novamente, esse tipo de operação é bastante útil quando trabalhamos com vetores.



- Lembre-se um vetor nada mais é do que um conjunto de elementos adjacentes na memória.

Estrutura de Dados 1

Comparação com ponteiros

```
void main(){
    int *a, *b, c=5, d=5;
    b=&c; a=&d;
    if(a > b){
        printf("\nO end apontado por a(%p)e maior que o end apontado por b(%p)",a,b);
    }else{
        if(a < b){
            printf("\nO end apontado por a(%p)eh menor que o end apontado por b(%p)",a,b);
        }else{ //(a==b)
            printf("\n a e b têm armazenado o mesmo endereço: %p",a);
        }
        if(*a == *b){
            printf("\nOs end. apontados por a e b possuem o mesmo valor: %d", *a);
        }
    }
}
```

Quando utilizamos o operador asterisco (*) na frente do nome do ponteiro, estamos acessando o conteúdo da posição de memória para qual o ponteiro aponta. Resumindo: acessamos o valor guardado no endereço para o qual o ponteiro aponta.

- %p formata endereços para exibição. Os endereços são dependentes do tipo de computador e do compilador. Também dependem do lugar em que o programa está carregado na memória. Assim um mesmo programa pode ocupar posições diferentes de memória em duas execuções distintas. Dessa forma, o programa pode ter suas variáveis alocadas em lugares diferentes, se for compilado com diferentes compiladores.

Estrutura de Dados 1

Comparação com ponteiros

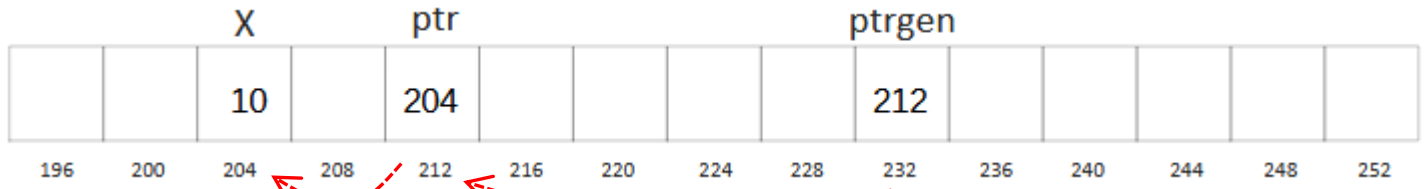
- É possível verificar se um ponteiro está apontando para uma posição válida:

```
1  #include <stdio.h>
2  void main(void){
3      int *a = NULL, *b = NULL, c=5;
4      a=&c;
5      if(a != NULL){
6      b = a;
7      printf("Numero : %d", *b);
8      }
9  }
```

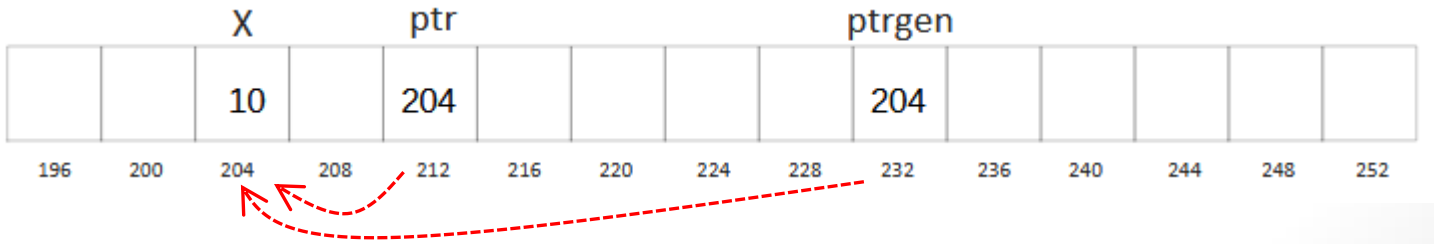
Ponteiros genéricos

- Normalmente, um ponteiro aponta para um tipo específico de dado. Porém, pode-se criar um ponteiro *genérico*. Esse tipo de ponteiro pode apontar para todos os tipos de dados existentes ou que ainda serão criados. Sua sintaxe é:

```
void *<nome_do_ponteiro>;
```



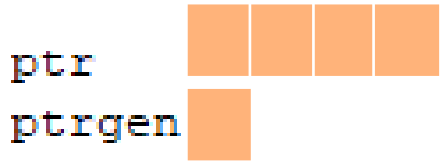

```
void *ptrgen;
int *ptr, x = 10;
ptr = &x;  //ptr recebe o endereço de x - 204
ptrgen = &ptr; //ptrgen recebe o endereço de ptr 212
printf("Endereço que ptrgen aponta: %p"; ptrgen); //imprime 212
```




```
ptrgen = ptr; //ptrgen recebe o conteúdo que está em ptr
printf("Endereço que ptrgen aponta: %p"; ptrgen); //imprime 204
```

Ponteiros genéricos

- Um ponteiro genérico permite armazenar um endereço de qualquer tipo de dado. Essa vantagem também carrega uma desvantagem: sempre que tivermos de acessar o conteúdo de um ponteiro genérico, será necessário utilizar o operador de ***typecast***, ou seja, antes de acessar seu conteúdo é necessário converter o ponteiro genérico para o tipo de ponteiro com o qual se deseja trabalhar, para que se possa acessar seu conteúdo;




```c
void *ptrgen;
int *ptr, x = 10;
ptr = &x;
ptrgen = ptr; //endereço de int
printf("Conteúdo do endereço em ptrgen: %d \n", *ptrgen);
//Isso gera erro!! Não é possível acessar o conteúdo.
printf("Conteúdo do endereço em ptrgen: %d \n", *(int*)ptrgen);
```

Acessa o conteúdo do endereço guardado em ptrgen. Porém não se sabe qual é o tamanho do tipo armazenado neste endereço...

Operador de *typecast*

Estrutura de Dados 1

Ponteiros genéricos - convertendo

É necessário converter o endereço para algum tipo conhecido, assim: “Eu quero que este ponteiro genérico seja tratado como um ponteiro para inteiro”.

Após isso é possível acessar o seu conteúdo.

```
*(int*)ptrgen
```

- As operações aritméticas como ponteiros genéricos são sempre realizadas com base em uma unidade de memória. Como genéricos não têm tipo estas operações funcionam com a menor unidade de memória, 1 *byte* por vez.

Ponteiro para Ponteiro

- Toda a informação que manipulamos dentro de um programa está obrigatoriamente armazenada na memória do computador e, portanto, possui um endereço de memória associado a ela.
- Ponteiros, como qualquer outra variável, também ocupam espaço na memória do computador e possuem endereço desse espaço de memória associado ao seu nome.
- Não existem diferenças entre a maneira como uma variável e um ponteiro são guardados na memória, assim é possível criar um ponteiro que aponte para outro ponteiro. Sua sintaxe:

```
<TIPO>  **<NOME_DA_VARIÁVEL>;
```

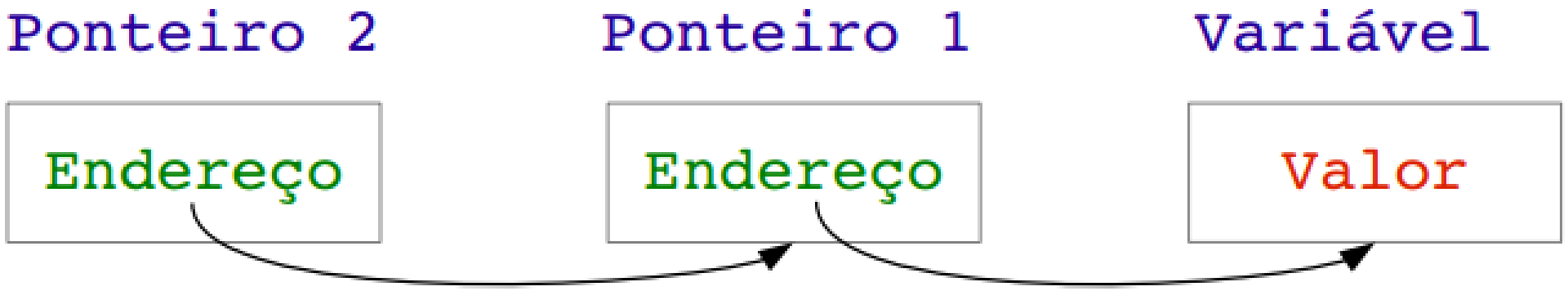
- Exemplo:

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

A linguagem C permite criar ponteiros com diferentes níveis de apontamento, isto é, ponteiros que apontam para outros ponteiros

Estrutura de Dados 1

Ponteiro para Ponteiro



- Em um ponteiro para ponteiro, o segundo ponteiro contém o endereço do primeiro ponteiro, que por sua vez aponta para a variável com o valor desejado

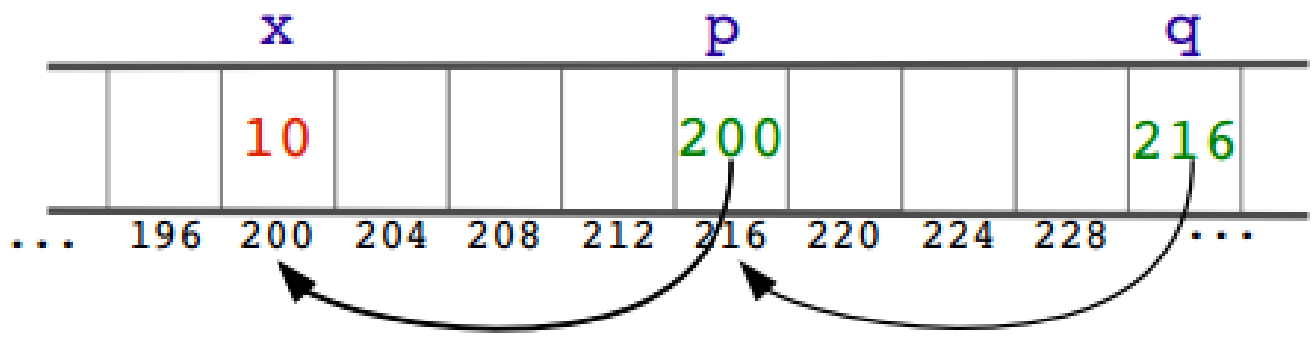
Estrutura de Dados 1

Ponteiro para Ponteiro

- Exemplo:

```c
void main(){
    int x, *p, **q;
    x = 10;
    p = &x; //atribui o endereço de x à p
    q = &p; //atribui o endereço de p à q
    printf("%d", **q); // imprime o valor de x
}
```

A linguagem C permite que se crie um ponteiro com um número infinito de níveis. Na prática, devemos evitar trabalhar com muitos níveis de apontamento

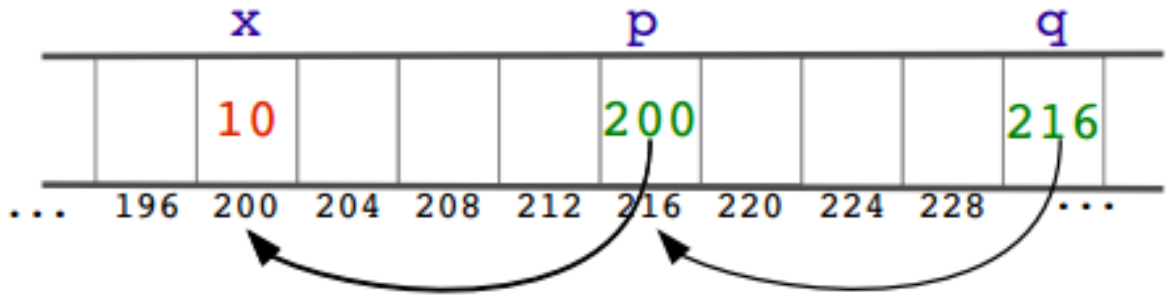


Estrutura de Dados 1

Ponteiro para Ponteiro

Acessa parcialmente o conteúdo

```c
int x = 10;
int *p = &x;
int **q = &p;
printf("q: %d \n", q);      //imprime o endereço de p
printf("*q: %d \n", *q);    //imprime o conteúdo de p - &x
printf("**q: %d \n", **q); //imprime o conteúdo do endereço do endereço - x
                            // ou seja 10
```



Estrutura de Dados 1

Ponteiro para Ponteiro

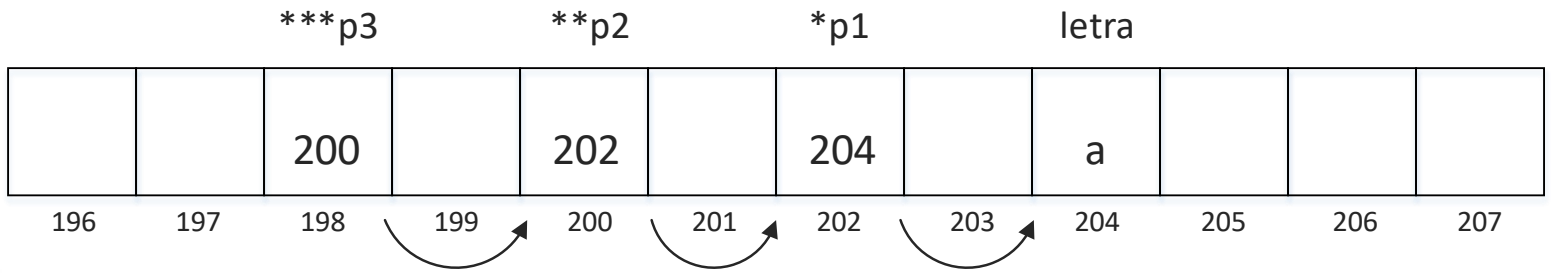
- É a quantidade de asteriscos (*) na declaração do ponteiro que indica o número de níveis do ponteiro:

```
int x;       //variável int
int *p1      //ponteiro para int (1 nível)
int **p2     //ponteiro para ponteiro de int (2 níveis)
int ***p3    //ponteiro para ponteiro para ponteiro de int (3 níveis)
```

Estrutura de Dados 1

Ponteiro para Ponteiro

```c
char letra = "a";
char *p1 = &letra;
char **p2 = &p1;
char ***p3 = &p2;
printf("*p1: %d \n", *p1);  //imprime "a"
printf("*p2: %d \n", *p2);  //imprime #204
printf("*p3: %d \n", *p3);  //imprime #202

printf("**p2: %d \n", **p2);    //imprime "a"
printf("***p3: %d \n", ***p3);  //imprime "a"
```



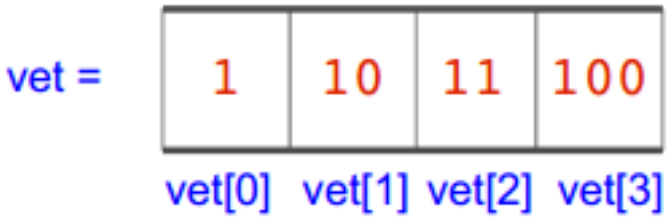
Vetores e Ponteiros

- Uma variável do tipo vetor, assim como um ponteiro, armazena um endereço na memória – o endereço do início do vetor;

- Ao passarmos um vetor como um argumento de uma função, na verdade:
  - Estamos passando um ponteiro para o início do espaço de memória alocado para o vetor.

- Exemplo:
  - A variável `vet` contém o endereço de memória do início do vetor:

```
int vet[4] = {1,10,11,100}
```



Estrutura de Dados 1

Vetores e Ponteiros

- Exemplo:

Note que este é um procedimento e NÃO RETORNA VALOR!!

```c
#include <stdio.h>
#include <stdlib.h>

void zera_vetor(int vetor[], int n){
    int i;
    for(i = 0; i < 5; i++){
        vetor[i] = 0;
    }
}

int main(){
    int i, vetor[5] = {1, 2, 3, 4, 5};
    printf("Vetor antes da funcao: \n");
    for(i = 0; i < 5; i++){
        printf(" %d", vetor[i]);
    }
    zera_vetor(vetor, 5);

    printf("\nVetor depois da funcao: \n");
    for(i = 0; i < 5; i++){
        printf(" %d", vetor[i]);
    }
    return 0;
}
```

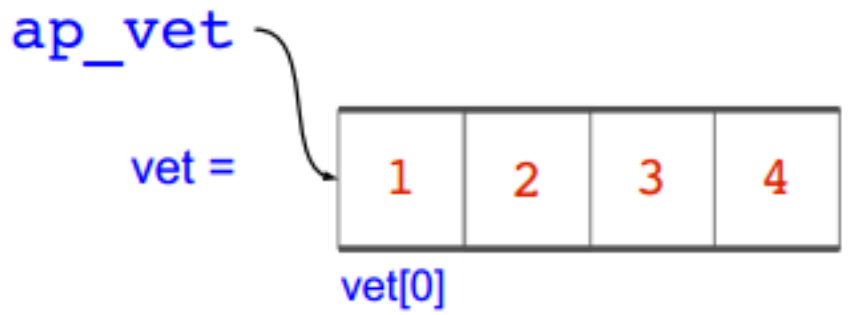
```
"C:\Users\angelot\Documents\Aulas\GRUEDA1\Aula 05 - P
Vetor antes da funcao:
 1 2 3 4 5
Vetor depois da funcao:
 0 0 0 0 0
Process returned 0 (0x0)   exec
Press any key to continue.
```

Vetores e Ponteiros

- A variável vet pode ser atribuída para um ponteiro como:

```
int vet[] = {1,2,3,4};
int *ap_vet;
ap_vet = vet;
```

Um vetor ou matriz é na realidade um ponteiro para uma região de memória onde estão armazenados dados



Estrutura de Dados 1

Vetores e Ponteiros

- Um ponteiro também pode ser usado como um vetor:

```c
int vet[] = {1,2,3,4};
int *ap_vet, i;

ap_vet = vet; //ap_vet = &vet[0];

for(i=0; i<4; i++){
    printf("%d , ", *(ap_vet + i)); // normalmente usamos a
                                    // forma simplificada ap_vet[i]
}
```

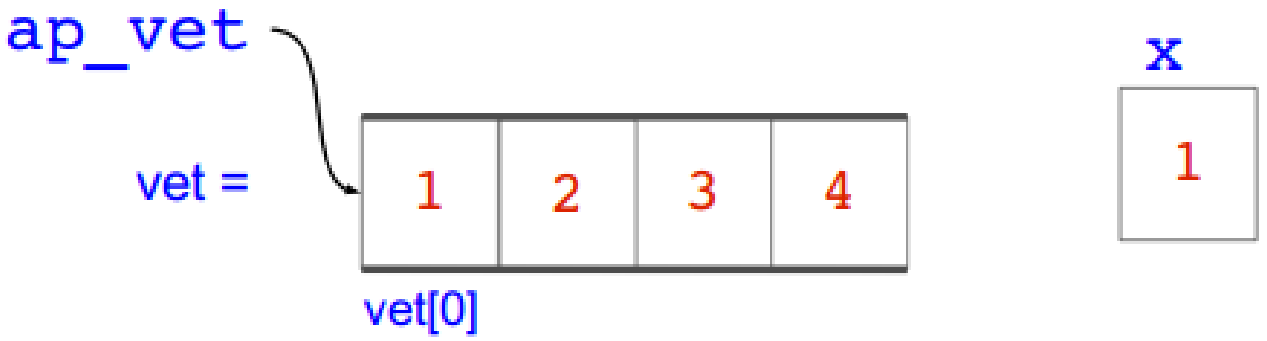
Estrutura de Dados 1

Vetores e Ponteiros

- Atribuição:

```
int vet[] = {1,2,3,4};
int *ap_vet, x;

ap_vet = vet;

x = *ap_vet; // x = vet[0]
```

ap_vet recebe o endereço inicial do vetor vet



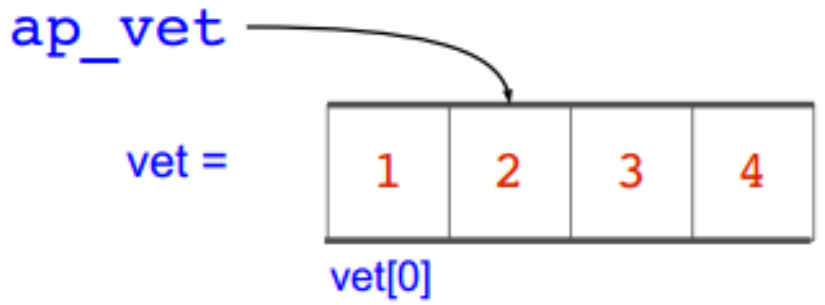
Estrutura de Dados 1

Vetores e Ponteiros

- Atribuição:

```
int vet[] = {1,2,3,4};
int *ap_vet;

ap_vet = vet;
```

- Se ap_vet aponta para vet[0], então ap_vet + 1 aponta para vet[1]
  - *(ap_vet + 1) aponta para vet[1],



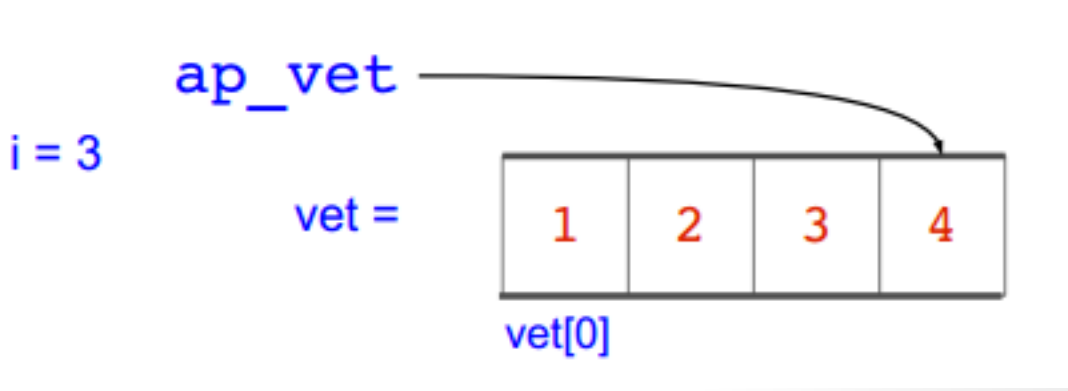
Estrutura de Dados 1

Vetores e Ponteiros

- Atribuição:

```
int vet[] = {1,2,3,4};
int *ap_vet;

ap_vet = vet;
```

- Se ap_vet aponta para vet[0], então
  - *(ap_vet + i) aponta para vet[i],

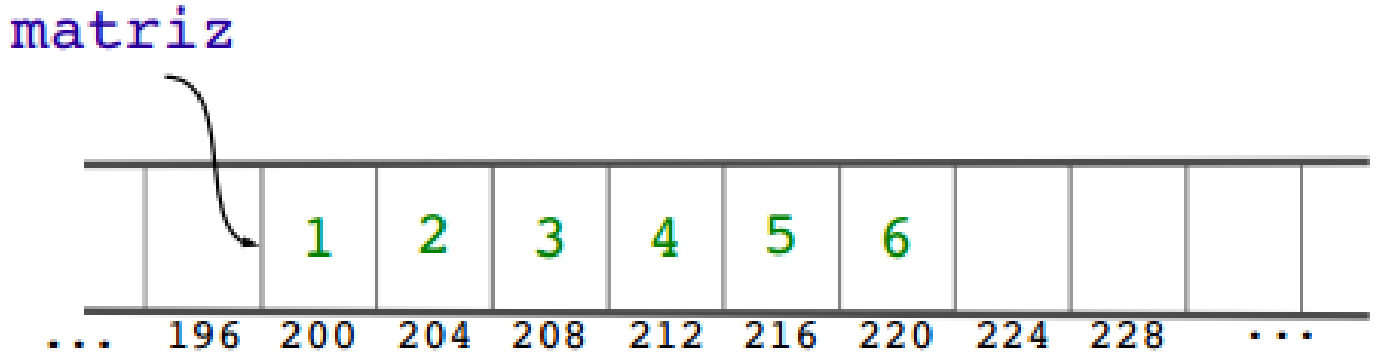


Estrutura de Dados 1

Representação de uma matriz na memória

matriz[3][2]

	0	1
0	1	2
1	3	4
2	5	6



Estrutura de Dados 1

Matrizes e ponteiros

- Acessando o conteúdo de uma posição na matriz:

```
matriz[i][j] == *(*(matriz + i)+j)
```

- Exemplo:

```
matriz[2][1] = 6

*(*(matriz+2)+1) = 6
```

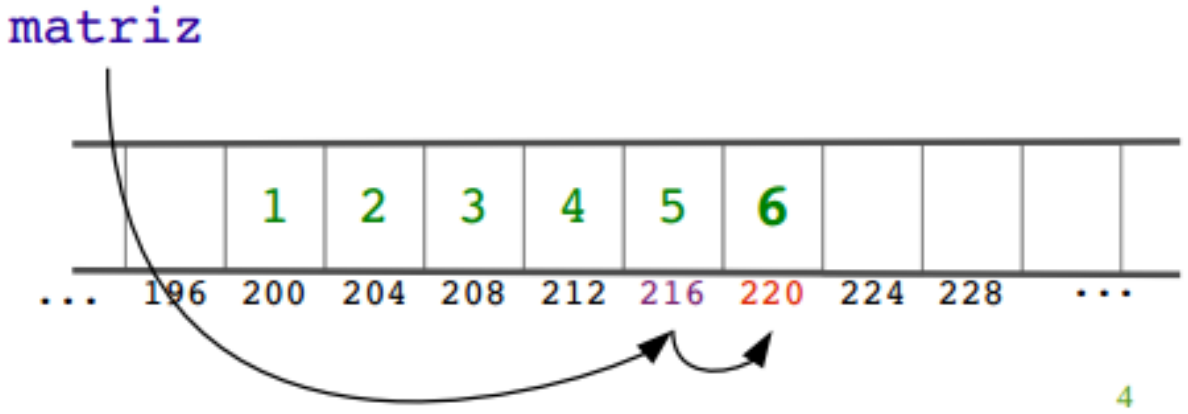
matriz

	0	1
0	1	2
1	3	4
2	5	6

Estrutura de Dados 1

Matrizes e ponteiros

```
*(matriz + 2) = 216        // acessa 1º endereço da 3ª linha
*(matriz + 2) + 1 = 220    // acessa 2º endereço da 3ª linha
*(*(matriz + 2) + 1) = 6 //acessa conteúdo do 2º endereço da
                           // 3ª linha
```



matriz

	0	1
0	1	2
1	3	4
2	5	6

Estrutura de Dados 1

Matrizes e ponteiros

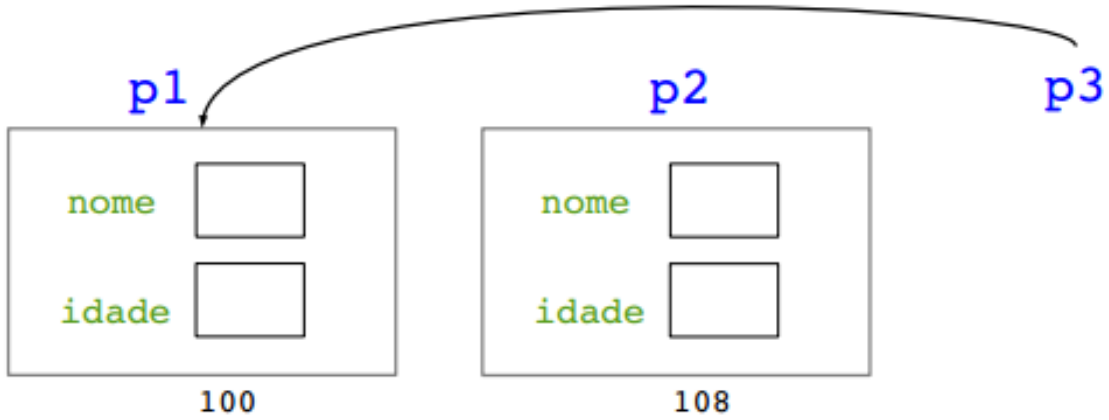
```c
#include <stdio.h>
#include <stdlib.h>
void main() {
  int l=3, c=3, i, j;
  int matriz [3][3]; // = {{1,2,3},{4,5,6},{7,8,9}};

  for(i=0; i<l; i++)
    for(j=0; j<c; j++)
      *(*(matriz + i)+j) = (i+1)*(j+1);

  for(i=0; i<l; i++) {
    for(j=0; j<c; j++)
      printf("%d\t", *(*(matriz + i)+j));
    printf("\n");
  }
}
```

Estrutura de Dados 1

Registros e ponteiros

- É possível criar um ponteiro para uma estrutura;

```c
#include <stdio.h>
struct pessoa{
    char nome[30];
    int idade;
};

int main(){
    struct pessoa p1, p2, *p3;
    p3 = &p1;
…
}
```



Registros e ponteiros

- Para acessar os campos de uma estrutura via ponteiro utilize o operador * juntamente com o operador . ;

```
(*PONTEIRO).CAMPO
```

Estrutura de Dados 1

Registros e ponteiros

- Exemplo:

```c
#include <stdio.h>
#include <string.h>
struct pessoa{
    char nome[30];
    int idade;
};
void main(){
    struct pessoa p1,*p3;
    p3 = &p1;
    strcpy((*p3).nome , "Ana");
    (*p3).idade = 18;
    printf("Pessoa 1) nome: %s, idade: %d",
    p1.nome,p1.idade);
}
```

O operador -> também pode ser usado para acessar os campos do registro quando o acesso se dá em **funções**:

PONTEIRO->CAMPO

```c
strcpy(p3->nome, “Ana”);
*p3->idade = 18;
```

Estrutura de Dados 1

Atividade 1

- Elabore um programa com 3 variáveis do tipo int e 3 ponteiros para o tipo int;
- Receba via teclado os valores de cada variável do tipo int;
- Faça com que os ponteiros apontem para as variáveis recebidas;
- Modifique os valores contidos nos endereços variáveis recebidas através dos ponteiros, somando 100 ao seu valor inicial. Faça isso passando os ponteiros como argumento para a função que será do tipo `void`;
- Imprima os valores finais de cada variável;
- Entregue no Moodle como atividade 1.

Estrutura de Dados 1

Atividade 2

- Escreva um programa que contenha duas variáveis inteiras. Leia essas variáveis do teclado. Em seguida, compare seus endereços e exiba o maior endereço e o conteúdo da variável que este endereço aponta.

- Exemplo:
- Endereço da variável *xyz*: 0x....
- Conteúdo da variável *xyz:* xxx...

- Entregue no Moodle como atividade 2

Estrutura de Dados 1

Atividade 3

- Crie um programa que contenha um vetor de `float` contendo dez elementos. Imprima na tela o endereço de cada posição desse vetor. Depois faça o mesmo com um vetor do tipo `double` de mesmo tamanho.

- Exemplo:
- Posição 0: 0x....
- Posição 1: 0x....
- Posição 2: 0x....
- E assim por diante.
- Entregue no Moodle como atividade 3