INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ESTRUTURA DE DADOS APLICADA

CONTINUAÇÃO DE PONTEIROS

ALOCAÇÃO DINÂMICA DE MEMÓRIA

PROFESSORA RESPONSÁVEL:
José Arnaldo Mascagni de Holanda
CONTATOS: arnaldomh@ifsp.edu.br

Modificadores, sizeof e Ponteiros

MODIFICADORES DE TIPOS EM C

Há **4 modificadores** de tipos em C:

- **signed –** *default;*
- **unsigned** – especifica variáveis sem sinal;
- **long** - aumenta o tamanho natural; e
- **short** - reduz o tamanho natural.

Aplicações dos modificadores em tipos pré-existentes:

- **int**: aplicam-se todos os modificadores
  - **signed** int; **unsigned** int; **long** int; **short** int;
- **float**: não se aplicam os modificadores;
- **double**: apenas o modificador **long**.

Tipo e Tipos Modificados	Num de bits	Formato para leitura com scanf	Intervalo	
			Inicio	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	32	%i	-2.147.483.648	2.147.483.647
unsigned int	32	%u	0	4.294.967.295
signed int	32	%i	-2.147.483.648	2.147.483.647
short int	16	%hi	-32.768	32.767
unsigned short int	16	%hu	0	65.535
signed short int	16	%hi	-32.768	32.768
long int	32	%li	-2.147.483.648	2.147.483.647
signed long int	32	%li	-2.147.483.648	2.147.483.647
unsigned long int	32	%u	0	4.294.967.295
float	32	%f	3,4E-38	3.4E+38
double	64	%lf	1,7E-308	1,7E+308
long double	80	%Lf	3,4E-4932	3,4E+4932

hardware de 16 bits

COMANDO SIZEOF

Características do comando sizeof:

- fornece o tamanho (em bytes) de variáveis ou de tipos;
- é substituído pelo tamanho do tipo ou variável *no momento da compilação*;
- é usado para garantir portabilidade;
- não é uma função, mas sim um operador.

sizeof var

sizeof (tipo)

EXEMPLO USANDO SIZEOF

```c
main()
{
    printf("short int : %d bytes\n\n", sizeof(short int) );

    printf("int : %d bytes\n\n", sizeof( int ) ); /*dependendo da máquina,
                                                    reserva 2 ou 4 bytes (long int).*/

    printf("long long int : %d bytes\n\n", sizeof(long long int) );

    printf("short: %d bytes\n\n", sizeof( short ) );

    printf("long: %d bytes\n\n", sizeof( long ) );

    printf("char : %d bytes\n\n", sizeof(char) );
    printf("float : %d bytes\n\n", sizeof(float) );
    printf("double : %d bytes\n\n", sizeof(double) );
}
```

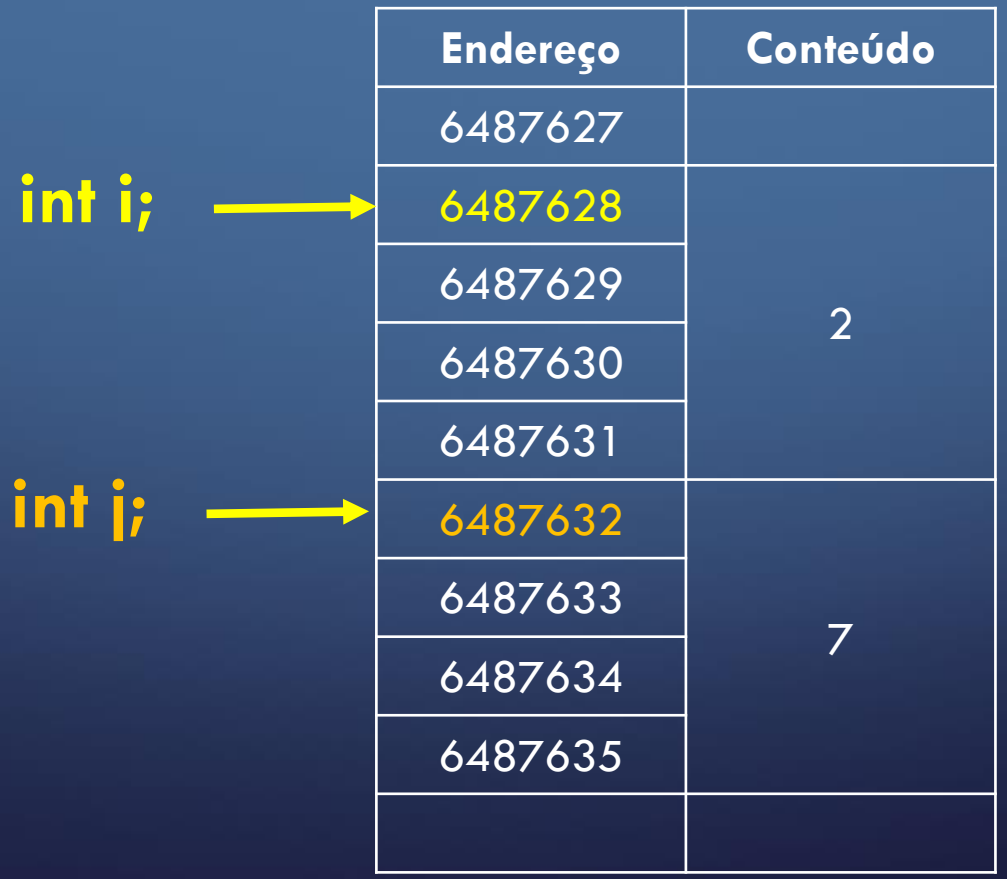
```
short int : 2 bytes

int : 4 bytes

long long int : 8 bytes

short: 2 bytes

long: 4 bytes

char : 1 bytes

float : 4 bytes

double : 8 bytes
```

INFORMAÇÕES ADICIONAIS SOBRE PONTEIROS: OPERAÇÕES

- **Operações básicas possíveis:**
  - ✓ Aritméticas: Soma → px++; Subtração → px--;
  - ✓ Operações Lógicas → <, >, ==, !=
  - ✓ Operações de referência (ou indireto) → *
  - ✓ Operações de endereço → &

INFORMAÇÕES ADICIONAIS SOBRE PONTEIROS: MEMÓRIA

- Ponteiros são inicializados com 0 ou NULL. Ex: int *px = NULL;
- O endereço de memória de uma variável, é o primeiro byte ocupado por ela.



```c
main() {
    int x = 1, y = 2;
    int *px, *py;

    px = &x;  py = &y;

    printf("x = %d, y = %d \n", x, y); getchar();

    (*px)++; (*py)++;
    printf("x = %d, y = %d \n", x, y); getchar();

    printf("px = %u, *px = %d, &px = %u\n", px, *px, &px); getchar();
    printf("py = %u, *py = %d, &py = %u\n", py, *py, &py); getchar();

    if ( px < py )
        printf("py - px = %u \n", py-px);
    else
        printf("px - py = %u \n", px-py);

    px++;
    printf("\npx++ .........\n");
    printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
    printf("x = %d \n", x); getchar();

    py = px + 3;
    printf("py = px + 3 .........\n"); printf("px = %u \n", px);
    printf("py = %u, *py = %d, &py = %u\n", py, *py, &py); getchar();
    printf("y = %d \n", y); getchar();
    printf("*py = %d \n", *py); getchar();

    printf("py - px  .........\n");
    printf("py - px = %u\n", py - px); getchar();
    printf("x = %d, y = %d \n", x, y); getchar();
}
```

```
x = 1, y = 2

x = 2, y = 3

px = 6487628, *px = 2, &px = 6487616

py = 6487624, *py = 3, &py = 6487608

px - py = 1

px++ .........
px = 6487632, *px = 136144, &px = 6487616
x = 2

py = px + 3 .........
px = 6487632
py = 6487644, *py = 0, &py = 6487608

y = 3

*py = 0

py - px  .........
py - px = 3

x = 2, y = 3
```

VOCÊ RESPONDE....

Qual a saída?

```
main()
{
        int y, *p, x;
        y = 7;
        p = &y;
        x = *p;
        x = 5;
        (*p)++;
        x--;
        (*p) += x;
        printf ("y = %d\n", y);
}
```

Relação entre Array e Ponteiros...

RELAÇÃO ENTRE ARRAY E PONTEIROS

Um **ponteiro** (declarado pelo operador indireto *) é um tipo especial de variável que armazena o endereço de outra variável. Este ponteiro é chamado de **ponteiro variável**.

Um **array** é um **ponteiro constante** (será sempre do mesmo tamanho) que não pode ter seu valor alterado.

→ Sendo assim, tudo que pode-se fazer com índices de matrizes, pode-se fazer com ponteiros.

```
*(matriz + índice)   ==   matriz[índice]
```

PONTEIROS E ARRAY: VERSÃO USANDO NOTAÇÃO DE MATRIZ

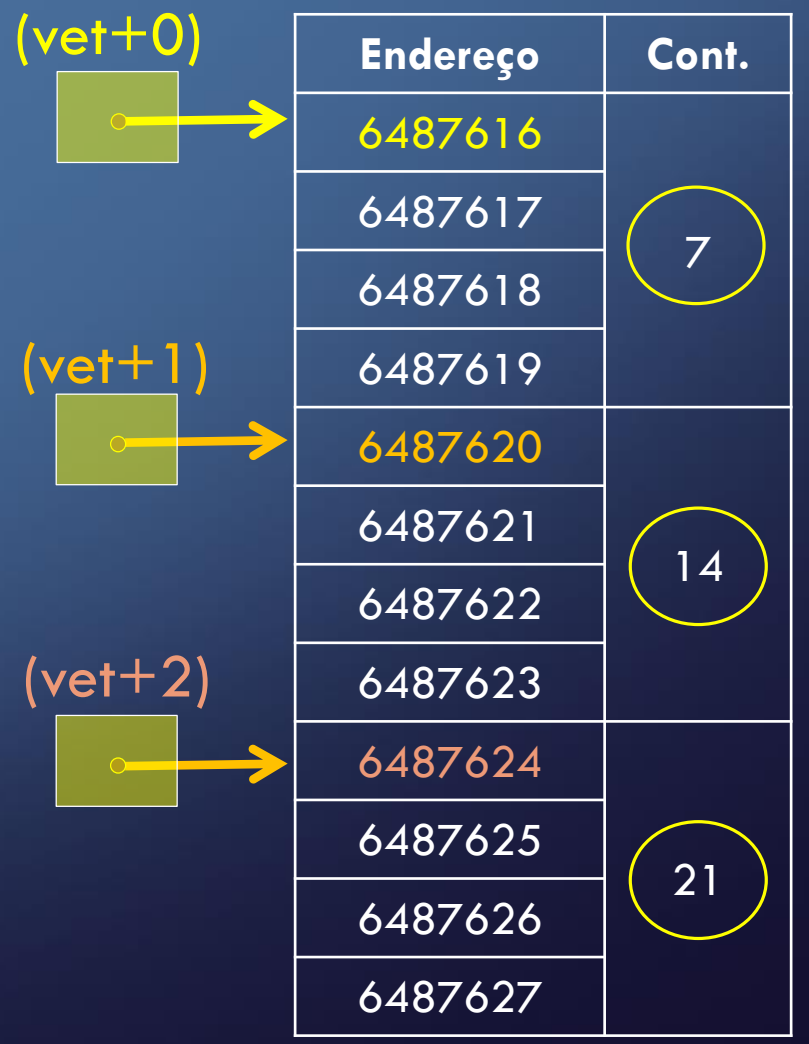
```
main()
{
        int i, vet[] = {7,14,21};

        for(i = 0; i < 3; i++)
                printf("%d\n", vet[i]);

        printf("\n\n Tamanho de vet %d", sizeof(vet));
}
```

```
7
14
21


 Tamanho de vet 12
```

PONTEIROS E ARRAY: VERSÃO USANDO NOTAÇÃO DE PONTEIRO

```
main()
{
        int i, vet[] = {7,14,21};

        for(i = 0; i < 3; i++)
                printf("%d\n", *(vet+i));

        printf("\n\n Tamanho de vet %d", sizeof(vet));
}
```

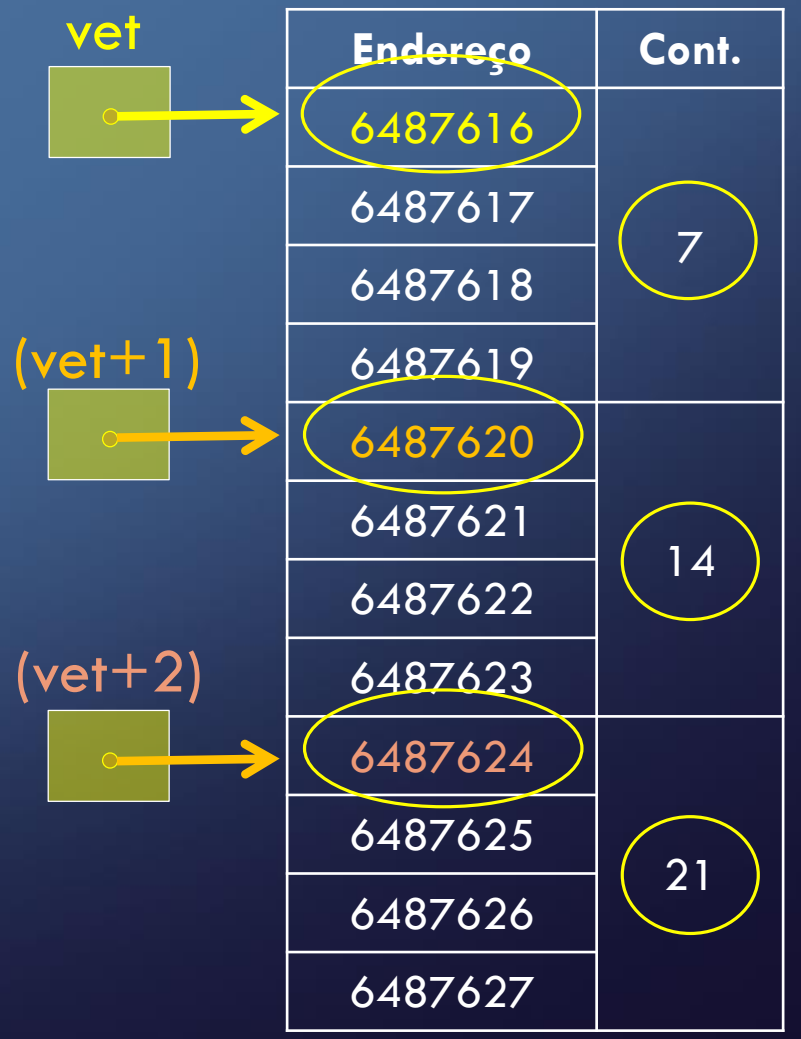
```
7
14
21


 Tamanho de vet 12
```



PONTEIROS E ARRAY: DESCOBRINDO ENDEREÇOS DE ELEMENTOS DA MATRIZ

```
main()
{
        int i, vet[] = {7,14,21};

        printf ("\n\nNotação de Ponteiro: \n");
        for(i = 0; i < 3; i++)
                printf("%u - %d\n", (vet+i), *(vet+i));

        printf ("\n\nNotação de Matriz: \n");
        for(i = 0; i < 3; i++)
                printf("%u - %d\n", &vet[i], vet[i]);
}
```

```
Notacao de Ponteiro:
6487616 - 7
6487620 - 14
6487624 - 21


Notacao de Matriz:
6487616 - 7
6487620 - 14
6487624 - 21
```



ARRAY É UM PONTEIRO!

O nome de um array é, na verdade, um ponteiro que aponta para o 1° elemento do array.

```
main()
{
   int i, vet[5] = {1,2,3,4,5};
   int *p;

   //p = &vet;  //ERRO
   p = vet; //CORRETO
```

ponteiro

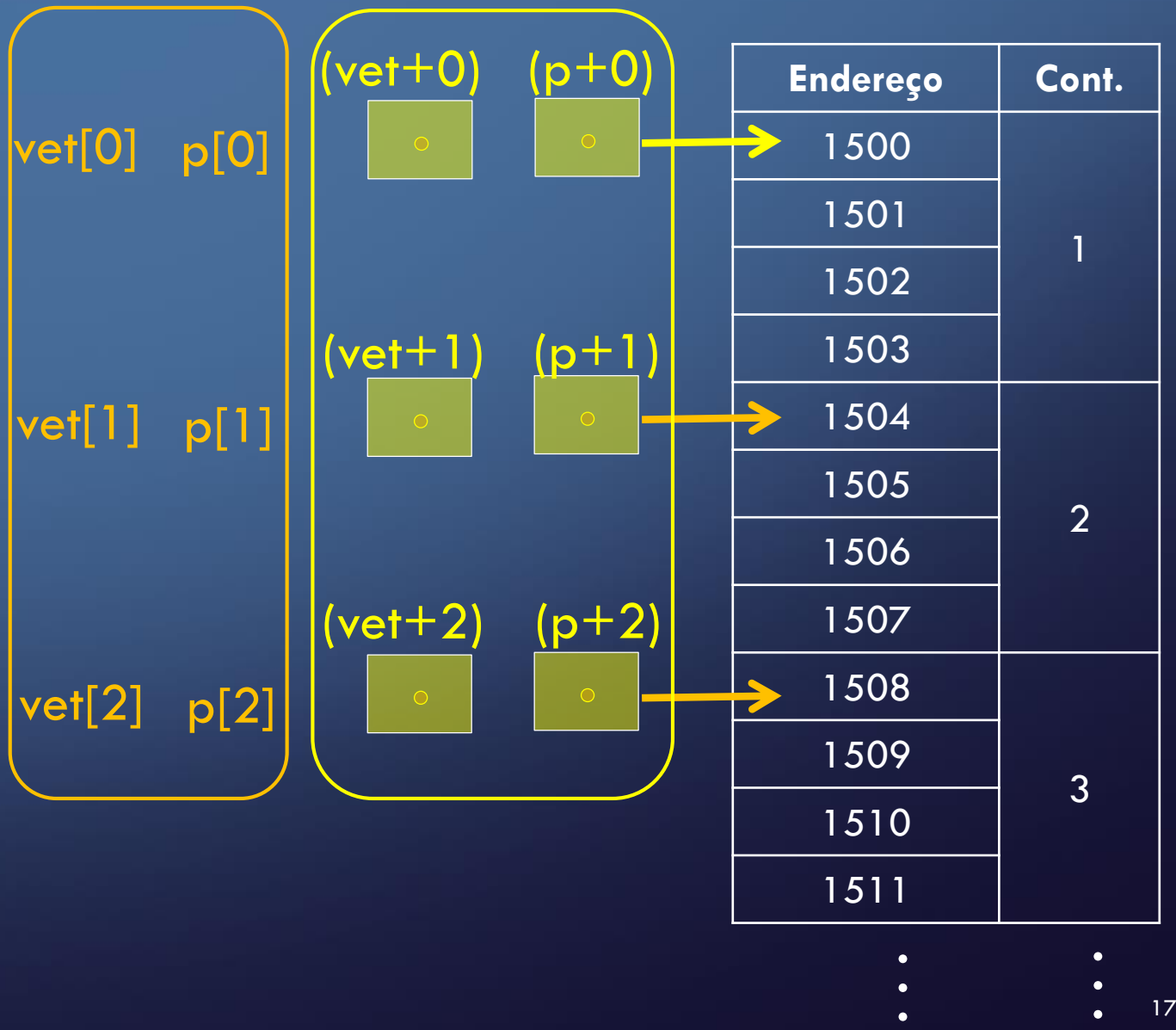
```
   for (i=0; i<5; i++)
        printf("*(p+%d) = %d\n", i, *(p+i));
   printf("\n\n");
   for (i=0; i<5; i++)
        printf("*(vet+%d) = %d\n", i, *(vet+i));
   printf("\n\n\n");
```

array

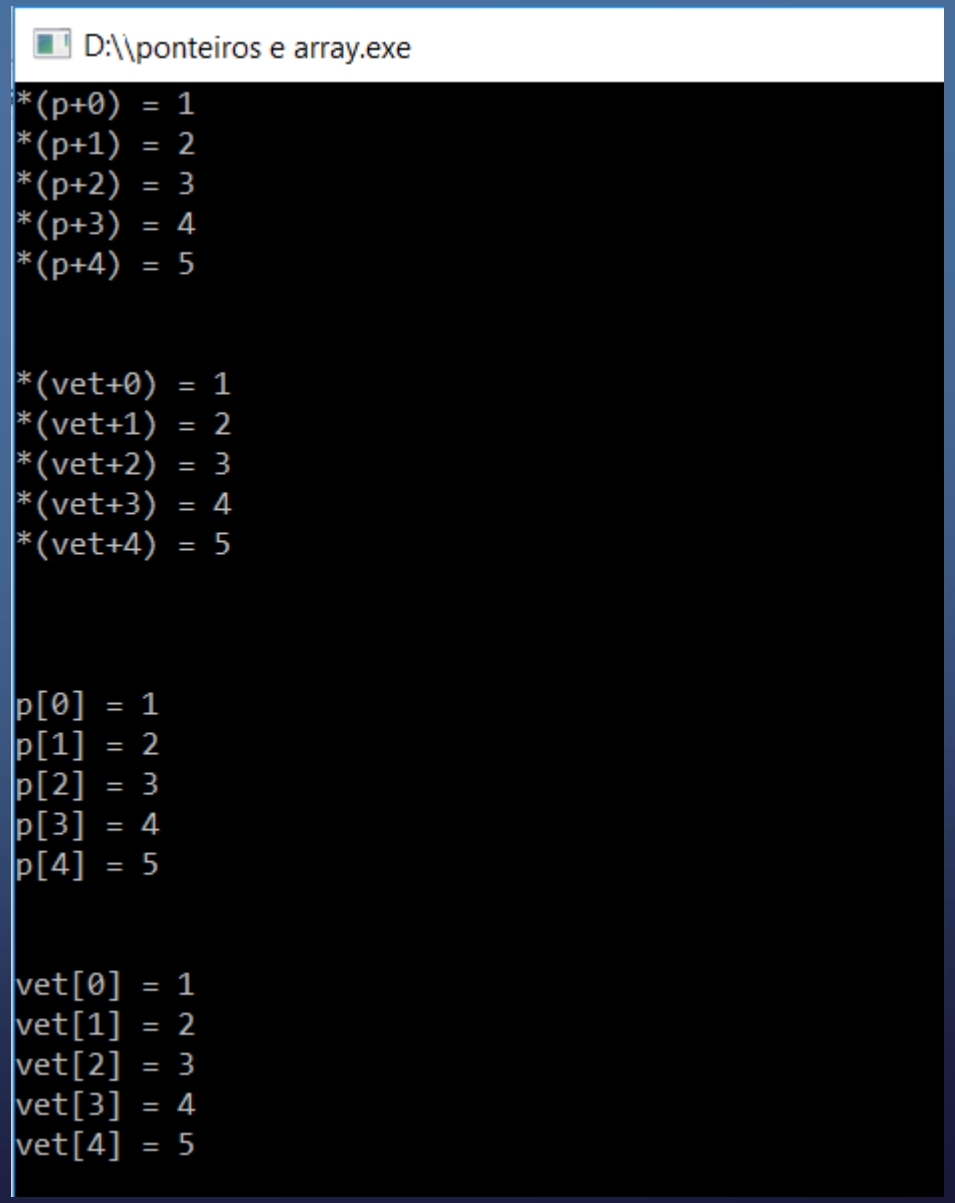
```
   for (i=0; i<5; i++)
        printf("p[%d] = %d\n", i, p[i]);
   printf("\n\n");
   for (i=0; i<5; i++)
        printf("vet[%d] = %d\n", i, vet[i]);
}
```

FORMAS DE ACESSO AO ARRAY (EXEMPLO)

```
for (i=0; i<5; i++)
        printf("*(p+%d) = %d\n", i, *(p+i));
printf("\n\n");
for (i=0; i<5; i++)
        printf("*(vet+%d) = %d\n", i, *(vet+i));

printf("\n\n\n");

for (i=0; i<5; i++)
        printf("p[%d] = %d\n", i, p[i]);
printf("\n\n");
for (i=0; i<5; i++)
        printf("vet[%d] = %d\n", i, vet[i]);
}
```



SAÍDA PARA O EXEMPLO ANTERIOR



INTERESSANTE!

Apesar de não fazer muito sentido, é possível ter índices negativos para um array.

Neste caso, estaríamos pegando posições de memória antes do vetor.

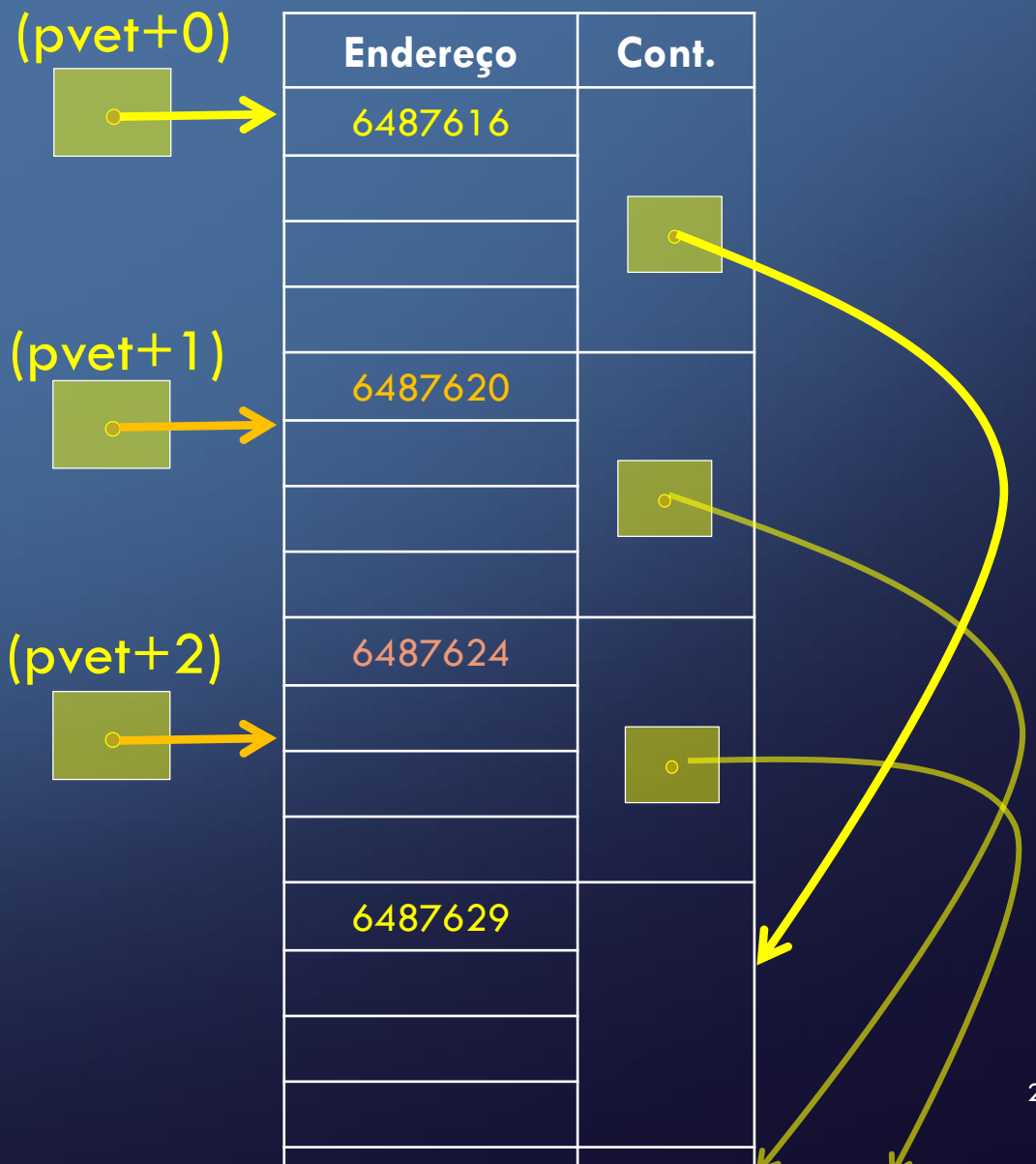
Isto explica também porque o C não verifica a validade dos índices. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

‘

Array de ponteiros...

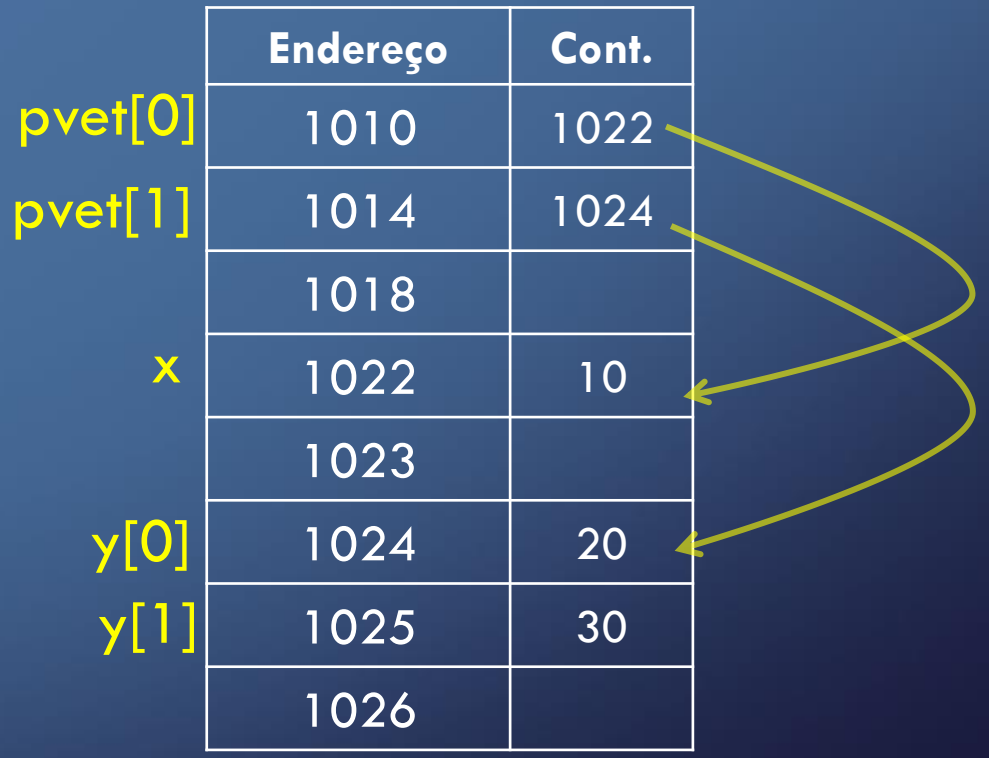
ARRAY DE PONTEIROS

```
main()
{
        int *pvet[3]; //array com 3 ponteiros


}
```



UTILIDADE DE ARRAY DE PONTEIROS

```
main()
{
        int *pvet[2];    //array com 2 ponteiros

        int x=10, y[2]={20,30};

        pvet[0] = &x;
        pvet[1] = y;    //endereço de y

       printf("\npvet[0]: %d\n", pvet[0]);    //&x
        printf("\npvet[1]: %d\n", pvet[1]);  //&y[0]

        printf("\n*pvet[0]: %d\n", *pvet[0]);   //x
        printf("\npvet[1][1]: %d\n", pvet[1][1]); //y[1]

}
```



Comandos typedef e -> (seta)...

COMANDO TYPEDEF

O comando **typedef** possibilita criar sinônimos para nomes de tipos de dados existentes.

```
typedef tipo_existente novo_nome;
```

Exemplo:

```
typedef int inteiro;
```

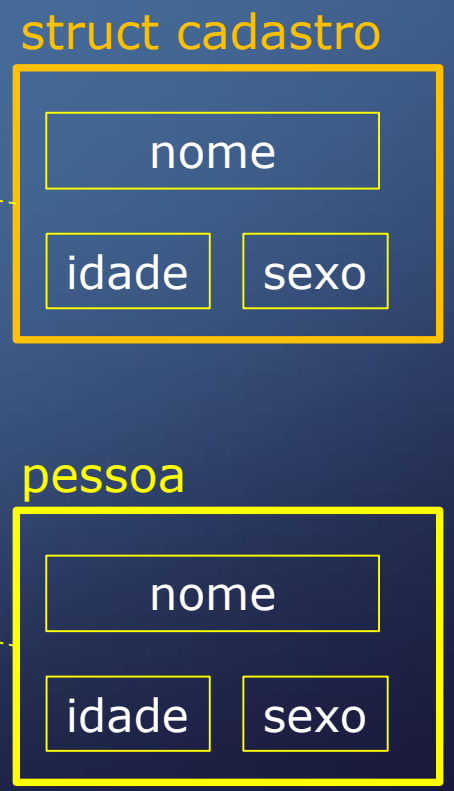
Exemplo de uso

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

typedef int inteiro;

int main(){
    int x = 10;
    inteiro y = 20;
    y = y + x;
    printf("Soma = %d\n",y);

    return 0;
}
```

USO DE TYPEDEF COM STRUCT

É possível definir nomes mais simples para estruturas. Isso evita o uso da palavra ***struct*** a cada referência de determinada estrutura.

```
struct cadastro{
     char nome[300];
     int idade;
     char sexo[1];
};

typedef struct cadastro pessoa;

main(){
    struct cadastro funcionario1;
    pessoa funcionario2;
}
```



COMANDO -> (SETA) PARA MANIPULAÇÃO INDIRETA DE CAMPOS DE *STRUCT*

Usando o operador *

```
struct ponto {
   int x, y;
}

main()
{
        struct ponto p1;
        atribui (&p1);
        printf("p1.x = %d e p1.y = %d",p1.x, p1.y);
}

void atribui(struct ponto *p)
{
   (*p).x = 10;
   (*p).y = 20;
}
```

```
p1.x = 10 e p1.y = 20
--------------------------------
```

Usando o operador ->

```
struct ponto {
   int x, y;
}

main()
{
        struct ponto p1;
        atribui (&p1);
        printf("p1.x = %d e p1.y = %d",p1.x, p1.y);
}

void atribui(struct ponto *p)
{
   p ->x= 10;
   p -> y = 20;
}
```

```
p1.x = 10 e p1.y = 20
--------------------------------
```

‘

ALOCAÇÃO DINÂMICA DE MEMÓRIA

PROBLEMA

Nem sempre é possível saber, em tempo de execução, o quanto de memória uma variável ou um programa irá usar.

```
int n, i;
double produtos[n];
```

```
int n, i;
scanf("%d", &n);
double produtos[n];
```

OK,
mas não indicado.

SOLUÇÃO: ALOCAÇÃO DINÂMICA

A *alocação dinâmica* permite ao programador criar "variáveis" em tempo de execução.

Características e vantagens da alocação dinâmica:

- Alocação de memória é feita sob demanda (quando o programa precisa);
- Menor desperdício de memória;
- Espaço é reservado até liberação explícita;
- Após um espaço ser liberado, estará disponibilizado para outros usos;
- Espaço alocado e não liberado explicitamente é automaticamente liberado ao final da execução.

FUNÇÕES UTILIZADAS NA ALOCAÇÃO DINÂMICA

A linguagem C usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na <stdlib.h>:

- **malloc** – usada para alocar memória. Passa-se o número de bytes a serem alocados;

MALLOC (*MEMORY ALLOCATION*)

```
void * malloc(unsigned int n);
```

Malloc retorna um ponteiro genérico e espera um inteiro sem sinal.

Exemplo1: Alocar 1000 bytes de memória livre.

```
char *c = malloc(1000);
```

```
char *c = (char *) malloc(1000);
```

Exemplo2: Alocar espaço para 50 inteiros.

```
int *v = malloc(50*sizeof(int);
```

200 bytes

```
int *v = (int *) malloc(50*sizeof(int);
```

MALLOC – SEM ESPAÇO SUFICIENTE EM MEMÓRIA PARA ATENDER À REQUISIÇÃO

Caso não haja memória suficiente para atender à requisição, a função **malloc()** retorna um ponteiro nulo.

```c
#include <stdio.h>
#include <stdlib.h>
main(){
    int i, *p;

    printf("ALOCACAO COM MALLOC\n");
    int *p1;
    p1 = (int *) malloc(5*sizeof(int));

    if (p1 == NULL){
        printf("ERRO: Overflow!\n");
        exit(1);
    }

    for (i = 0; i<5; i++){
        printf("Digite o valor da posicao %d: ", i);
        scanf("%d", &p1[i]); //p pode ser tratado como vetor
    }
}
```

RELEMBRANDO: PONTEIROS GENÉRICOS

void *nome_ponteiro;

Ponteiro Genérico aponta para qualquer TIPO DE DADO (EXISTENTE [inclusive para outro ponteiro] OU QUE SERÁ CRIADO). Ex:

```
#include <stdio.h>
#include <stdlib.h>
main (){
    int x = 10;
    void *px;

    px = &x;

     printf("Conteúdo: %d", *px ); //ERRO
     printf("Conteudo: %d", (int *)px ); //CORRETO
}
```

FUNÇÕES UTILIZADAS NA ALOCAÇÃO DINÂMICA

A linguagem C usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na <stdlib.h>:

- **malloc** – usada para alocar memória. Passa-se o número de bytes a serem alocados;
- **free** – usada para liberar memória previamente alocada;

FREE

```
void free(void *p);
```

```c
#include <stdio.h>
#include <stdlib.h>
main(){

    printf("ALOCACAO COM MALLOC\n");
    int i, *p1;
    p1 = (int *) malloc(5*sizeof(int));

    if (p1 == NULL){
        printf("ERRO: Overflow!\n");
        exit(1);
    }

    for (i = 0; i<5; i++){
        printf("Digite o valor da posicao %d: ", i);
        scanf("%d", &p1[i]);
    }

    free(p1);
}
```

FUNÇÕES UTILIZADAS NA ALOCAÇÃO DINÂMICA

A linguagem C usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na <stdlib.h>:

- **malloc** – usada para alocar memória. Passa-se o número de bytes a serem alocados;
- **free** – usada para liberar memória previamente alocada;
- **calloc** – também aloca memória. Passam-se dois parâmetros: quantidade de posições e o tamanho de tipo de dado alocado.

CALLOC

void * **calloc(unsigned** int n, **unsigned** int size);

```c
#include <stdio.h>
#include <stdlib.h>
main(){

    printf("\n\nALOCACAO COM CALLOC\n");
    int i, *p2;
    p2 = (int *) calloc(50, sizeof(int));

    if (p2==NULL){
        printf("ERRO: Overflow!\n");
        exit(1);
    }
    for (i = 0; i<5; i++){
        printf("Digite o valor da posicao %d: ", i);
        scanf("%d", &p2[i]); //p pode ser tratado como vetor
    }
    free(p2);
}
```

MALLOC X CALLOC

MALLOC - apenas faz alocação de memória e te devolve um ponteiro para o primeiro byte.

CALLOC – faz a mesma coisa, mas inicializa todos os bits com 0.

Se estiver trabalhando com **inteiro**, você terá todos os bits com 0, portanto terá o valor 0 mesmo em cada posição.

Enquanto com MALLOC, você não terá a inicialização com 0, portanto, terá lixo de memória em cada posição.

FUNÇÕES UTILIZADAS NA ALOCAÇÃO DINÂMICA

A linguagem C usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na <stdlib.h>:

- **malloc** – usada para alocar memória. Passa-se o número de bytes a serem alocados;
- **free** – usada para liberar memória previamente alocada;
- **calloc** – também aloca memória. Passam-se dois parâmetros: quantidade deposições e o tamanho de tipo de dado alocado.
- **realloc** – usada para realocar (para mais ou para menos) memória previamente alocada.

REALLOC

void * **realloc(void** *p, **unsigned** int n);

```c
main(){
    int i;

    printf("ALOCACAO COM MALLOC\n");
    int *p = malloc(5*sizeof(int));

    for (i = 0; i<5; i++){
        p[i] = i+1;
    }

    for (i = 0; i<5; i++){
        printf("%d\n", p[i]);
    }

    printf("\n");
    //diminui tamanho do array
    p = realloc(p, 3*sizeof(int));
    for (i = 0; i<3; i++){
        printf("%d\n", p[i]);
    }

    printf("\n");
    //aumenta tamanho do array
    p = realloc(p, 10*sizeof(int));
    for (i = 0; i<10; i++){
        printf("%d\n", p[i]);
    }
}
```

```
ALOCACAO COM MALLOC
1
2
3
4
5

1
2
3

1
2
3
4
5
0
1174405190
36262
1643616
0
```

Espaços consecutivos – contém valores anteriores.

OBSERVAÇÕES (USANDO REALLOC PARA ALOCAÇÃO E LIBERAÇÃO)

- Se não há vetor definido na realocação, aloca **n** bytes e devolve um ponteiro (igual malloc);

```
p = (int *) realloc(NULL, 50 * sizeof(int));
```

equivale

```
p = (int *) malloc(50 * sizeof(int));
```

- se **n** é zero, a memória apontada por ***p** é liberada, equivalente ao *free*().

```
p = (int *) realloc(p, 0);
```

- Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

ALOCAÇÃO DINÂMICA DE STRUCT

Assim como os tipos primitivos, também é possível fazer alocação de tipos estruturados:

- Um ponteiro para **struct** receberá o **malloc()**
- Utilizamos o **operador seta** para acessar o conteúdo
- Usamos **free()** para liberar a memória alocada

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

struct cadastro {
    char nome[50];
    int idade;
};

main(){

    printf("MALLOC de STRUCT\n");
    struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));

    strcpy(cad->nome,"Maria");
    cad->idade  = 27;

    printf("\nNome = %s, Idade = %d", cad->nome, cad->idade);

    free(cad);
}
```

ALOCAÇÃO DINÂMICA DE STRUCT

Assim como os tipos primitivos, também é possível fazer alocação de tipos estruturados:

- Um ponteiro para **struct** receberá o **malloc()**
- Utilizamos o **operador seta** para acessar o conteúdo
- Usamos **free()** para liberar a memória alocada

```
printf("\n\nALTERNATIVAMENTE....\n");
struct cadastro *cad = (struct cadastro*) malloc(sizeof(struct cadastro));

strcpy((*cad).nome,"Maria");
(*cad).idade    = 27;

printf("\nNome = %s, Idade = %d", (*cad).nome, (*cad).idade);

free(cad);
```

DÚVIDAS?

EXERCÍCIOS (ALOCAÇÃO DINÂMICA)

1. Crie um programa que:
(a) Aloque dinamicamente um array de 5 números inteiros,
(b) Peça para o usuário digitar os 5 números no espaço alocado,
(c) Mostre na tela os 5 números,
(d) Libere a memória alocada.

2. Faça um programa que leia um número N e:
(a) Crie dinamicamente e leia um vetor de inteiro de N posições;
(b) Leia um número inteiro X e conte e mostre os múltiplos desse número que existem no vetor.

3. Escreva um programa que leia primeiro os 6 números gerados pela loteria e depois os 6 números do seu bilhete. O programa então compara quantos números o jogador acertou. Em seguida, ele aloca espaço para um vetor de tamanho igual a quantidade de números corretos e guarda os números corretos nesse vetor. Finalmente, o programa exibe os números sorteados e os seus números corretos.

EXERCÍCIOS (ALOCAÇÃO DINÂMICA)

4. Considere um cadastro de produtos de um estoque, com as seguintes informações para cada produto:

Código de identificação do produto: representado por um valor inteiro
Nome do produto: com até 50 caracteres
Quantidade disponível no estoque: representado por um número inteiro
Preço de venda: representado por um valor real

(a) Defina uma estrutura, denominada produto, que tenha os campos apropriados para guardar as informações de um produto
(b) Crie um conjunto de N produtos (N é um valor fornecido pelo usuário) e peça ao usuário para entrar com as informações de cada produto
(c) Encontre o produto com o maior preço de venda
(d) Encontre o produto com a maior quantidade disponível no estoque