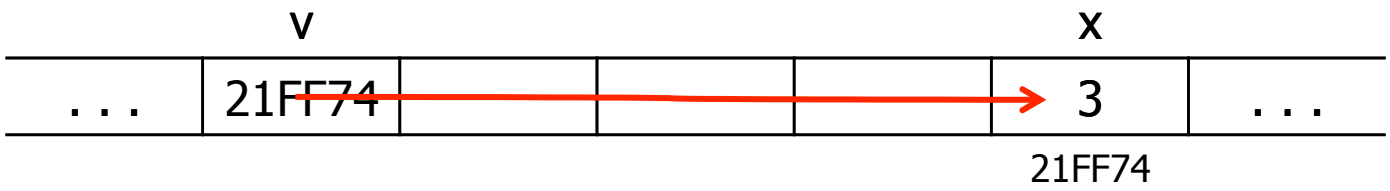
Alocação Dinâmica de Memória

- Para relembrar: o que é mesmo um ponteiro?
- Um ponteiro (também conhecido como variável de referência) é uma variável que armazena um endereço de memória (onde está amazenado um valor).
- Para declarar um ponteiro é preciso saber para qual tipo de valor o ponteiro faz referência.
- Exemplo: A variável x armazena o valor inteiro 3. O endereço da variável x é 21FF74. A variável v armazena o endereço 21FF74. Portanto, a variável v faz referência ao valor inteiro 3. Ou seja, v é um ponteiro para int. Logo, a variável v deve ser declarada como: int * v;



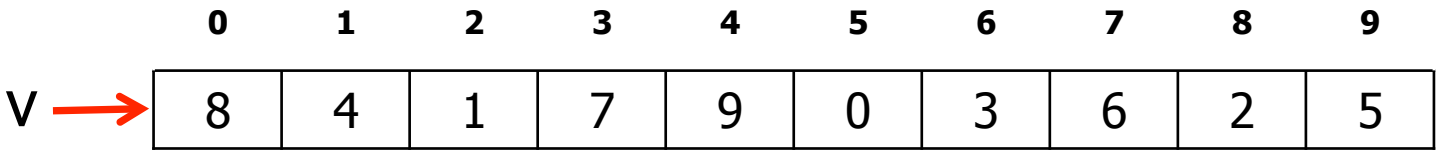
Importante saber:

```
void alterar(int x)
{
  x = 10;
  return;
}
int main()
{
  int x = 5;
  alterar(x);
  printf("x = %d\n",x);
  return 0;
}
```

```
void alterar(int *x)
{
  *x = 10;
  return;
}
int main()
{
  int x = 5;
  alterar(&x);
  printf("x = %d\n",x);
  return 0;
}
```

```
void alterar(int n, int v[])
{
  int i;
  for (i = 0; i < n; i++)
  {
    v[i] = 10;
  }
  return;
}
int main()
{
  int i,n
  n = 5;
  int v[5] = {1,2,3,4,5};
  alterar(n,v);
  printf("v = [");
  for (i = 0; i < n; i++)
  {
    printf("%d ",v[i]);
  }
  printf("]\n");
  return 0;
}
```

- O nome de um vetor é um ponteiro para o seu primeiro elemento, ou seja, o nome de um vetor armazena o endereço deste vetor.
- Exemplo: int v[10];



- Como o nome de um vetor é um endereço, se um vetor é usado como parâmetro de uma função, a passagem do parâmetro é por referência (e não por valor).
- Assim, as modificações que a função fizer no vetor serão preservadas e o vetor pode ser visto, além de um parâmetro de entrada, também como um parâmetro de saída da função.
- Como v é um ponteiro para int, uma outra forma de declarar este vetor é: int * v;

- Qual é a diferença entre declarar o vetor v como int v[10] e declarar este vetor como int * v?
- A diferença está na alocação de memória.
- Ao declarar int v[10], o compilador aloca automaticamente o espaço de memória necessário, ou seja, aloca 10*4 = 40 bytes de memória para o vetor v (lembrar que cada valor do tipo int consome 4 bytes de memória).
- Neste caso, como a memória é alocada na compilação (e não na execução do programa), tem-se uma alocação estática de memória.
- Ao declarar int * v, nenhuma memória é alocada pelo compilador porque a declaração não especifica quantos elementos tem o vetor.
- Na execução do programa, o número de elementos do vetor deverá ser especificado e, então, poderá haver a alocação dinâmica de memória para o vetor.

- O espaço de memória para um determinado tipo pode variar entre compiladores. No compilador GCC cada valor do tipo int consome 4 bytes de memória. Para um outro compilador, esse número pode ser diferente.
- Assim, para determinar o número de bytes reservado pelo compilador para um determinado tipo deve-se utilizar a função sizeof.
- Para fazer a alocação dinâmica de memória, podemos usar a função malloc.
- A função malloc requer, como parâmetro: o tamanho do bloco de memória a ser alocado (em bytes), que pode ser dado por: número de posições de memória * sizeof(tipo de dado desejado).
- A função malloc retorna um ponteiro do tipo void * para o início do espaço de memória alocado. Este ponteiro deve ser convertido para o tipo de dado desejado.

- Resumindo:

```
int *v;
v = (int *)malloc(10*sizeof(int));
```

Converte para o **tipo de dado** correto.

Aloca um bloco de **10 * 4 bytes** de memória.

Retorna o **tamanho** (em bytes) do tipo **int**. Para o GCC, retorna 4.

- Qual é a vantagem de declarar um vetor como ponteiro?
- A vantagem é não ser necessário definir, a priori, o tamanho do vetor.
- O tamanho do vetor pode ser estabelecido na execução do programa, quando a memória para o vetor for alocada pela função malloc.

Pode ser uma variável.

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

- Outra forma de fazer a alocação dinâmica de memória é usando a função calloc.
- A função calloc requer 2 parâmetros: o número de posições de memória e o tamanho de cada posição de memória a ser alocada (em bytes), dado pela função sizeof.
- A função calloc também retorna um ponteiro void * para o início do espaço de memória alocado. Este ponteiro deve ser convertido para o tipo de dado desejado.
- Portanto, são equivalentes:

```
int *v;
v = (int *)malloc(10*sizeof(int));
```

```
int *v;
v = (int *)calloc(10,sizeof(int));
```

- Ao declarar um vetor como:

```
int v[MAX];
```

  o tamanho da constante MAX precisa ser um valor estimado. Isso pode levar a duas situações possíveis:
  - Desperdício de memória (se MAX for maior do que o necessário;
  - Falta de memória (se MAX for menor do que o necessário, o que será muito mais grave).

- Exemplo: um vetor para armazenar os coeficientes de um polinômio. Qual deve ser o tamanho do vetor?
- Declarando o vetor como ponteiro, pode-se alocar o espaço de memória necessário e suficiente depois de conhecido o grau do polinômio. Dessa forma, não haverá desperdício nem falta de memória.

Exercícios

1. Escrever um programa que usa vetor para armazenar os coeficientes de um polinômio. Escrever uma função void mostrar(int n, int *p), que recebe como parâmetros o grau de um polinômio (n) e seus coeficientes (p) e mostra o polinômio da forma usual.

2. Escrever a função int procura(int *V, int n, int k) que retorna 1 se o valor k aparece no vetor V (de n elementos) e retorna 0, caso contrário.

3. Escrever a função int uniao(int *A, int n, int *B, int m, int *C), que recebe como parâmetros os vetores A (de tamanho n) e B (de tamanho m) e constrói o vetor C, que deve conter todos os elementos de A e B, sem repetição. A função deve retornar o tamanho do vetor C.

Strings como ponteiros

- Considere o seguinte programa:

```
int main()
{
  char *m;
  m = "Sao Paulo FC";
  printf("%s\n",m);
  return 1;
}
```

- Neste caso, m é um ponteiro para char, ou seja, um vetor de caracteres (ou seja, um string).
- Note que não há alocação de memória explícita (nem estática, nem dinâmica) para m.
- Isto é possível porque, para strings, o espaço de memória necessário e suficiente é alocado, implicitamente (e de forma dinâmica), no momento da atribuição.

- A instrução: m = "Sao Paulo FC" aloca 13 posições de memória para m:

S	a	o		P	a	u	l	o		F	C	\0

- Se em seguida, o programa faz: m = "SP Campeao", o espaço alocado para m será diminuído, automaticamente, para 11 posições.

S	P		C	a	m	p	e	a	o	\0

- Este tipo de alocação dinâmica ocorre apenas no caso de atribuições para strings.
- Se o valor do string não for atribuído explicitamente (por exemplo, se o valor for lido), a alocação deve ser feita de forma estática ou usando malloc.

- Uma vantagem da alocação dinâmica é a possibilidade de reduzir ou aumentar a quantidade de memória alocada anteriormente.
- Isto pode ser feito com a função realloc, cujos parâmetros são: um ponteiro para o início do bloco de memória e a quantidade de bytes a ser alocada.
- Essa função retorna um ponteiro para o início do novo bloco de memória. Este ponteiro pode ser o mesmo ponteiro usado para o bloco de memória original.
- Caso não seja o mesmo, a função realloc copia os dados armazenados no bloco de memória original para o novo bloco de memória.

```
v = (int *)malloc(5*sizeof(int));
...
v = (int *)realloc(v,10*sizeof(int));
...
v = (int *)realloc(v,3*sizeof(int));
```

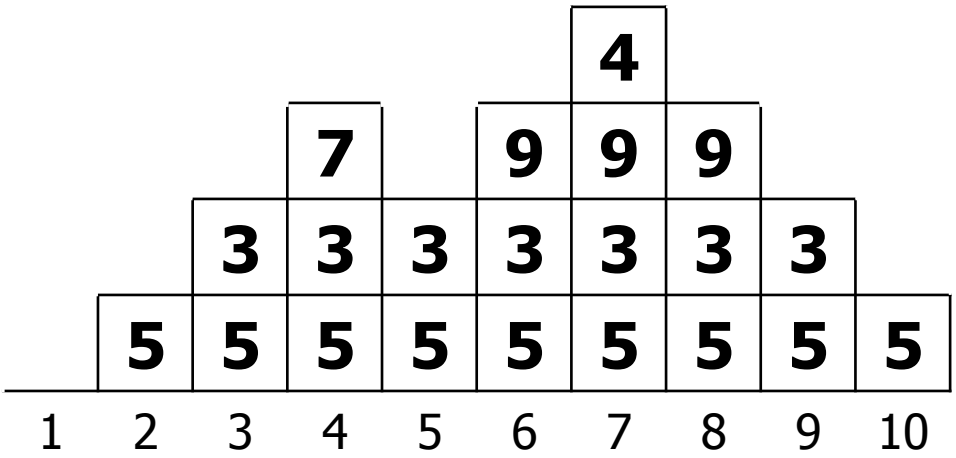
```
int v[5];

int v[10];

int v[3];
```

Exercício

Considere uma pilha de valores inteiros. Em uma pilha, as operações (empilhar e desempilhar) são realizadas no topo.



Lembrar que só é possível **realocar**, se antes, a memória foi **alocada dinamicamente**.

Escrever um programa contendo as funções void empilhar(int *n, int *p, int x), que empilha o valor x, e int desempilhar(int *n, int *p), que retorna o valor desempilhado. Utilizar as funções malloc e realloc para alocar memória para a pilha, de modo que o vetor p tenha sempre o tamanho necessário e suficiente.

Solução:

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

void empilhar(int *n, int *p, int x)
{
  (*n)++;
  p = (int *)realloc(p,(*n)*sizeof(int));
  p[(*n)-1] = x;
  return;
}

int desempilhar(int *n, int *p)
{
  int x = p[(*n)-1];
  (*n)--;
  p = (int *)realloc(p,(*n)*sizeof(int));
  return x;
}
```

```
void mostrarPilha(int n, int *p)
{
  int i;
  printf("Pilha: [");
  for (i = 1; i < n; i++)
  {
    printf(" %d",p[i]);
  }
  printf("]\n");
}

int menuOpcoes()
{
  int opcao;

  printf("Selecione uma opcao:\n");
  printf("  1. Empilhar\n");
  printf("  2. Desempilhar\n");
  printf("  3. Fim\n");
  printf("> ");
  scanf("%d",&opcao);
  return opcao;
}
```

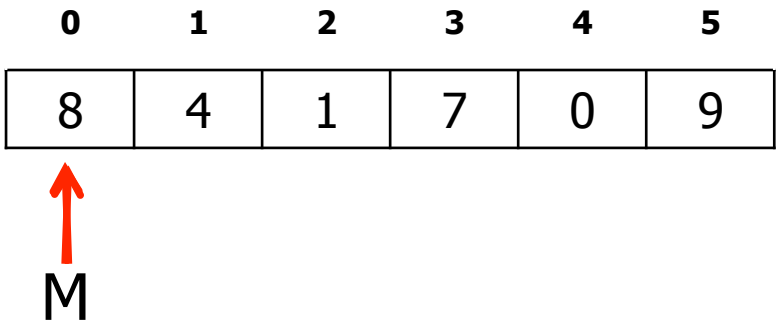
```
int main()
{
  int i,n,opcao,x,*p;
  n = 1;
  p = (int *)malloc(n*sizeof(int));
  while (1)
  {
    mostrarPilha(n,p);
    opcao = menuOpcoes();
    switch (opcao)
    {
      case 1:
        printf("Valor: ");
        scanf("%d",&x);
        empilhar(&n,p,x);
        break;
      case 2:
        x = desempilhar(&n,p);
        printf("Valor desempilhado: %d\n",x);
        break;
      case 3:
        return 0;
    }
  }
}
```

Alocação dinâmica de memória para matrizes

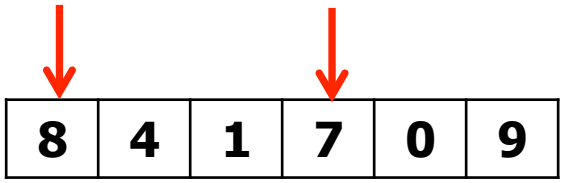
- Como no caso de vetores, o nome de uma matriz é também um ponteiro para a primeira posição de memória alocada para esta matriz.
- Exemplo: M[2][3]

M =

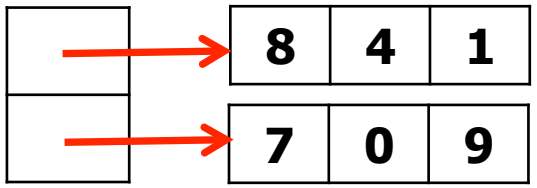
8	**4**	**1**
7	**0**	**9**



- Portanto, uma matriz também pode ser declarada como um ponteiro. Embora os elementos de M estejam em posições consecutivas de memória (ou seja, M poderia ser pensado como um vetor), podemos imaginar M como:



ou, de uma forma mais abstrata, como:

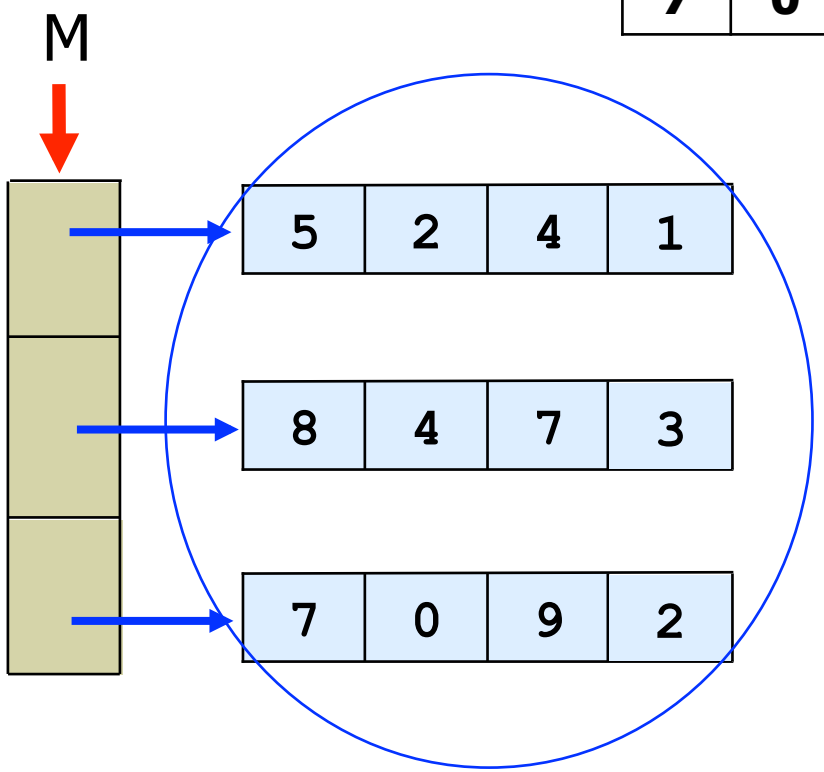


- Em geral, uma matriz m x n pode ser imaginada como um vetor de m elementos, em que cada M[i] é um ponteiro para o início de um vetor de n elementos.

- Exemplo: int M[3][4];

M =

5	**2**	**4**	**1**
8	**4**	**7**	**3**
7	**0**	**9**	**2**



M aponta para um vetor de **m** ponteiros.

Cada elemento do vetor aponta para um vetor de **n** valores.

Cada vetor armazena uma **linha** da matriz

- Portanto, para alocar memória para a matriz M precisamos:
  1. Alocar um vetor de tamanho 3 para armazenar os endereços das linhas da matriz.
  2. Para cada elemento deste vetor, alocar um vetor de tamanho 4 para armazenar os valores (int) de uma linha da matriz.
- O passo 2 já sabemos fazer:

```
for (i = 0; i < 3; i++)
   M[i] = (int *)malloc(4*sizeof(int));
```

- Mas, e o passo 1?

```
M = (int **)malloc(3*sizeof(int *));
```

M é um **ponteiro** para um **ponteiro** para **int**.

Cada elemento é um **ponteiro** para **int**.

- Portanto, a alocação dinâmica de memória para uma matriz M de m linhas e n colunas será feita como:

```
int **M;
M = (int **)malloc(m*sizeof(int *));
for (i = 0; i < m; i++)
{
  M[i] = (int *)malloc(n*sizeof(int));
}
```

- A alocação de memória para matrizes com mais dimensões pode ser feita de forma análoga. Por exemplo: como seria para uma matriz tridimensional A[p][q][r]?

```
int ***A;
A = (int ***)malloc(p*sizeof(int **));
for (i = 0; i < p; i++)
{
  A[i] = (int **)malloc(q*sizeof(int *));
  for (j = 0; j < q; j++)
    A[i][j] = (int *)malloc(r*sizeof(int));
}
```

Exercícios

1. Escrever a função float * vetor(int n), que aloca dinamicamente um vetor de tamanho n, prenche o vetor com valores float entre 0 e 1, gerados aleatoriamente, e retorna um ponteiro para esse vetor.
2. Escrever a função float * maximo(float *V, int n), que retorna um ponteiro para o maior valor presente no vetor V (de tamanho n).
3. Escrever a função float ** matriz(float *V, int n, int r, int *c), que aloca dinamicamente uma matriz com **r** linhas e copia os elementos do vetor V (de tamanho n) para essa matriz. O número de colunas da matriz (c) deve ser o menor possível, em função dos valores de n e r. Os elementos adicionais da matriz devem ser preenchidos com zero. A função retorna um ponteiro para a matriz e retorna no parâmetro c, o número de colunas.
4. Escrever a função main que passa um vetor criado pela função vetor para as outras duas funções e mostra os resultados.

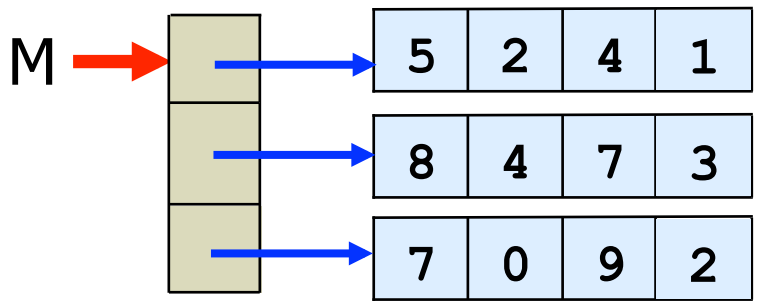
Liberação de memória

- A linguagem C, além de funções para alocar memória, dispõe da função free para liberar memória que foi alocada anteriormente e que não é mais necessária.
- Exemplo:

```
int *v;
...
v = (int *)malloc(n*sizeof(int));
...
free(v);
```

Observar que basta **liberar o ponteiro** que aponta para a área de memória alocada.

- Como liberar a memória alocada para uma matriz?



Primeiro: liberar os ponteiros que apontam para as linhas.

Depois: liberar o ponteiro que aponta para o vetor de ponteiros.

```
for (i = 0; i < n; i++)
{
  free(M[i]);
}
free(M);
```

- Exercício: Escrever um programa que lê um vetor de inteiros, remove valores consecutivos repetidos e mostra o vetor atualizado. O programa deve alocar memória para o vetor inicial e usar a função realloc para ajustar o tamanho do vetor sempre que necessário. O tamanho do vetor deve ser sempre o mínimo necessário para armazenar os valores. O programa deve mostrar o tamanho final do vetor.

Tamanho inicial = 10

0	0	1	1	1	2	2	3	0	3

Tamanho final = 6

0	1	2	3	0	3

Solução:

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

int main()
{
  int i,j,n,x;
  int *v;

  printf("Tamanho: ");
  scanf("%d",&n);
  v = (int *)malloc(n*sizeof(int));
  printf("Vetor: ");
  for (i = 0; i < n; i++)
  {
    scanf("%d",&v[i]);
  }
```

```
  i = 0;
  while (i < n-1)
  {
    if (v[i] == v[i+1])
    {
      for (j = i; j < n-1; j++)
      {
        v[j] = v[j+1];
      }
      n--;
      v = (int *)realloc(v,n*sizeof(int));
    }
    else
    {
      i++;
    }
  }
  printf("Vetor: [");
  for (i = 0; i < n; i++)
    printf(" %d",v[i]);
  printf("]\n");

  return 0;
}
```