



Algoritmos e Programação de Computadores

Disciplina 113476

Prof. Alexandre Zaghetto
<http://alexandre.zaghetto.com>
zaghetto@unb.com

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

O presente conjunto de *slides* não pode ser reutilizado ou republicado sem a permissão do instrutor.

Módulo 13

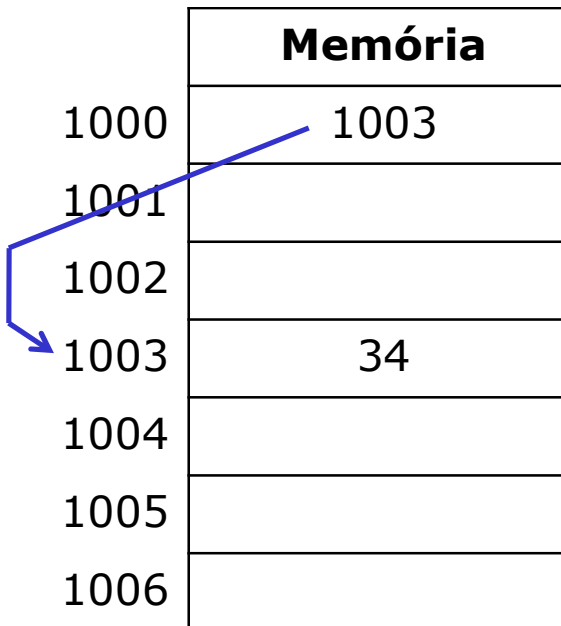
Ponteiros

1. Ponteiros

- **Ponteiro** é uma variável que contém o endereço de uma outra variável.
- Daí o nome, pois ele **aponta** para outra variável.

Memória	
1000	1003
1001	
1002	
1003	34
1004	
1005	
1006	

Endereços de memória



The diagram illustrates a pointer variable. A blue arrow originates from the value '1003' stored at memory address 1001 and points to the memory address 1003. Another blue arrow points from the label 'Endereços de memória' to the address 1003. The value '34' is stored at memory address 1003.

1. Ponteiros

- Alguns usos:
 - ✓ Manipular vetores e matrizes, incluindo strings.
 - ✓ Modificar os argumentos (variáveis, vetores, matrizes e *structs*) de funções (passagem por referência).
 - ✓ Alocar e desalocar memória dinamicamente.
 - ✓ Passar para uma função o endereço de outra função.
 - ✓ Criar estruturas de dados complexas.

1. Ponteiros

- Declaração de variáveis ponteiros:

`tipo *nome`

- Operadores de Ponteiros

- ✓ Existem dois operadores especiais para ponteiros:

- ✓ &

- ✓ *



1. Ponteiros

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

```
int main (int argc, char *argv[])
{
```

```
    int x;
```

```
    x = 15;
```

```
    printf("CONTEUDO de X = %d \n", x);
```

```
    printf("ENDERECO de X = %d \n", &x);
```

```
    return 0;
```

```
}
```

& → Pode ser lido como
"o endereço de...".

1. Ponteiros

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

int main (int argc, char *argv[])
{
    int *p, x;

    x = 15;
    p = &x;

    printf("%d \n", p);

    return 0;
}
```




1. Ponteiros

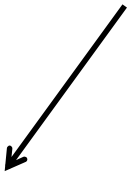
```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int *p, x;

    x = 15;
    p = &x;

    printf("%p \n", p);

    return 0;
}
```





1. Ponteiros

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

```
int main (int argc, char *argv[])
{
```

```
    int *p, x;
```

```
    x = 15;
```

```
    p = &x;
```

```
    printf ("%d \n", p);
```

```
    printf ("%d \n", *p);
```

```
    return 0;
```

```
}
```

* → Pode ser lido como
"o valor que está no
endereço armazenado em..."



1. Ponteiros

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

int main (int argc, char *argv[])
{
    int *p, q, x;

    x = 15;
    p = &x;
    q = *p;

    printf("%d \n", p);
    printf("%d \n", *p);
    printf("%d \n", q);

    return 0;
}
```



1. Ponteiros

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

int main (int argc, char *argv[])
{
    int *p1, *p2, x, y, z;

    x = 10;
    p1 = &x;
    p2 = p1;

    printf("x: %d \n", x);
    printf("&x: %d \n", &x);
    printf("p1: %d \n", p1);
    printf("&p1: %d \n", &p1);
    printf("p2: %d \n", p2);
    printf("&p2: %d \n", &p2);
```



1. Ponteiros

```
y = *p1;  
printf("y: %d \n", y);  
  
z = *p2;  
printf("z: %d \n", z);  
  
return 0;  
}
```

1. Ponteiros

- Aritmética de ponteiros:
 - ✓ Existem duas operações possíveis com ponteiros:
 - ✓ Adição; e
 - ✓ Subtração.

1. Ponteiros

- Aritmética de ponteiros:
 - ✓ Consideremos **p1** um ponteiro para um inteiro com valor atual 1000. Assuma, também, que os inteiros são de 4 bytes.
 - ✓ Após a expressão `p1++`, `p1` contém 1004.
 - ✓ Cada vez que **p1** é incrementado, ele aponta para o próximo inteiro.
 - ✓ O mesmo é verdade nos decrementos.
 - ✓ Ou seja, ponteiros incrementam ou decrementam **pelo tamanho do tipo de dado** que eles apontam.



1. Ponteiros

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

int main (int argc, char *argv[])
{
    int *p1, x = 10;

    p1 = &x;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %d \n", x);

    p1++;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %d \n", x);

    return 0;
}
```




1. Ponteiros

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

int main (int argc, char *argv[])
{
    char *p1, x = 'a';

    p1 = &x;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %c \n", x);

    p1++;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %c \n", x);

    return 0;
}
```



1. Ponteiros

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

int main (int argc, char *argv[])
{
    double x = 1.23212345, *p1;

    p1 = &x;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %lf \n", x);

    p1 = p1 + 2;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %lf \n", x);

    return 0;
}
```

1. Ponteiros

**Endereços de
memória**

	Memória
0996	1000
0997	
1998	
0999	
1000	
1001	
1002	
1003	
1004	
1005	
1006	

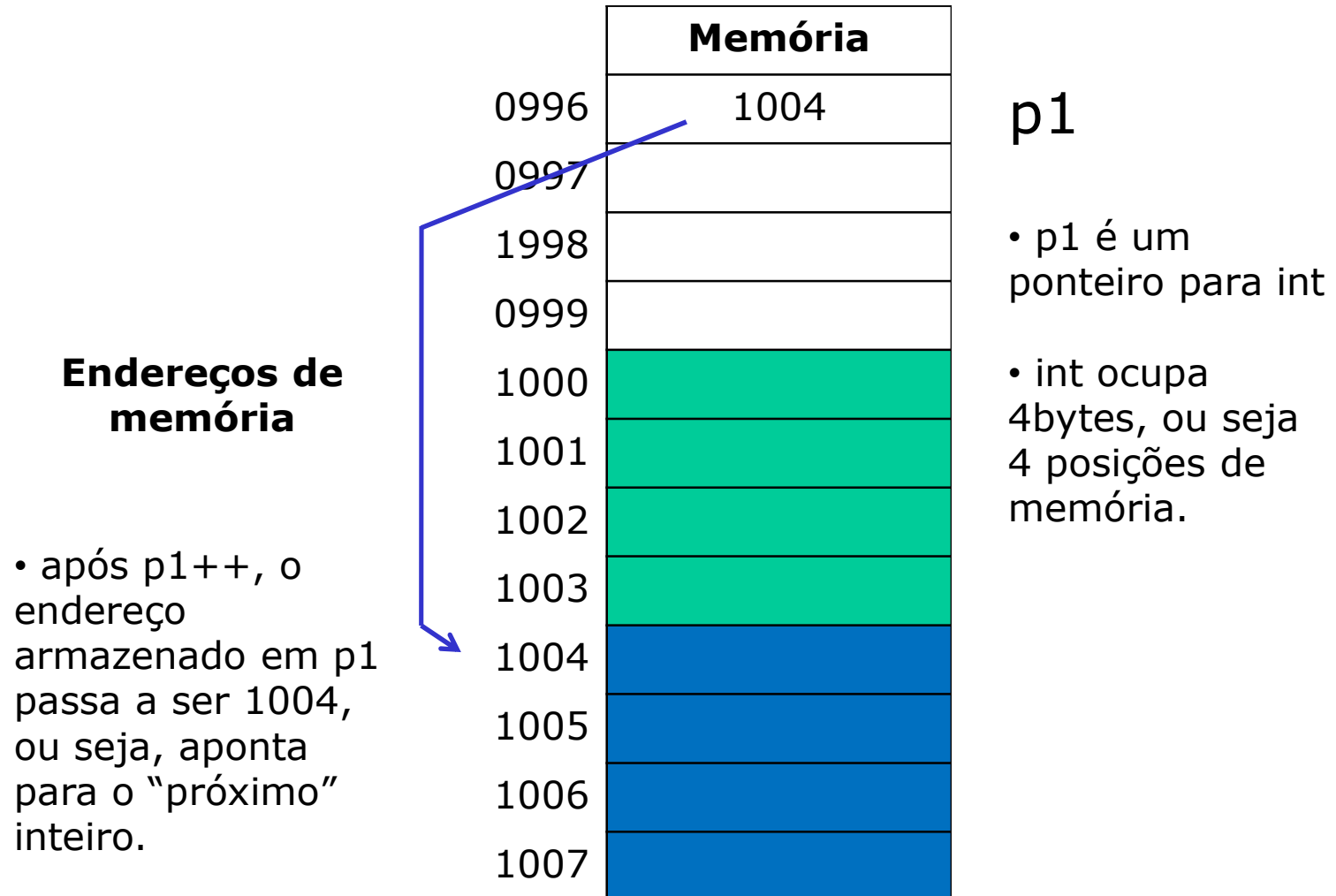
p1

- p1 é um ponteiro para int

- int ocupa 4bytes, ou seja 4 posições de memória.



1. Ponteiros





1. Ponteiros

- Curiosidade:

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

int main (int argc, char *argv[])
{

    int x[] = {1, 10}, *p, end;

    p = x;

    end = (int)p;
    end++;

    p = (int *)end;

    printf("p = %d \n", *p);

    return 0;
}
```

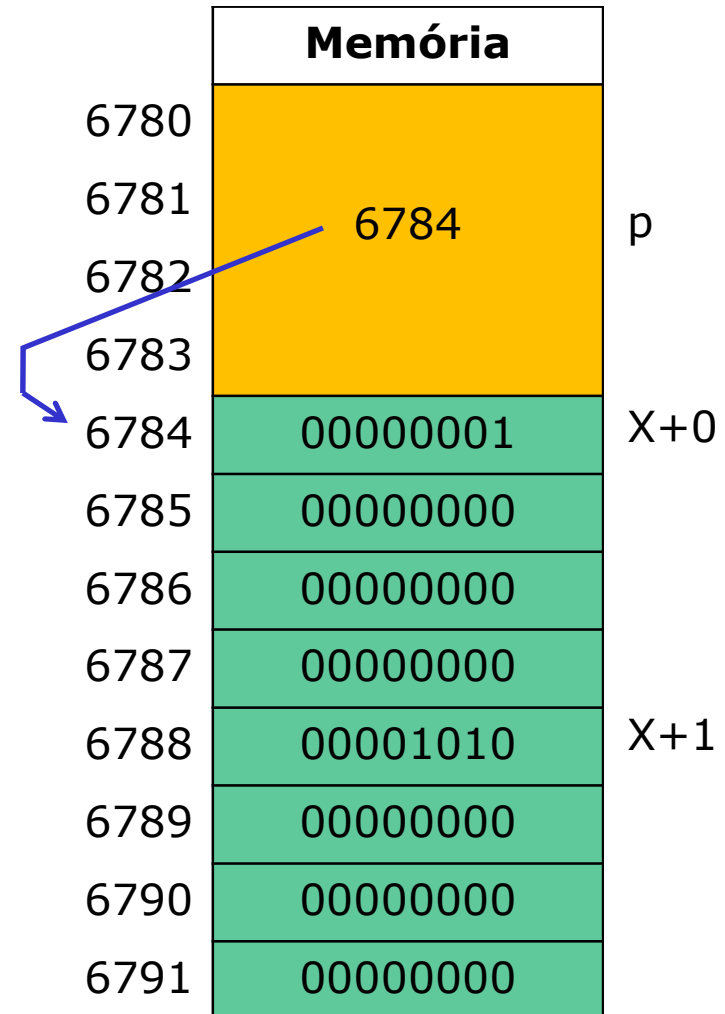
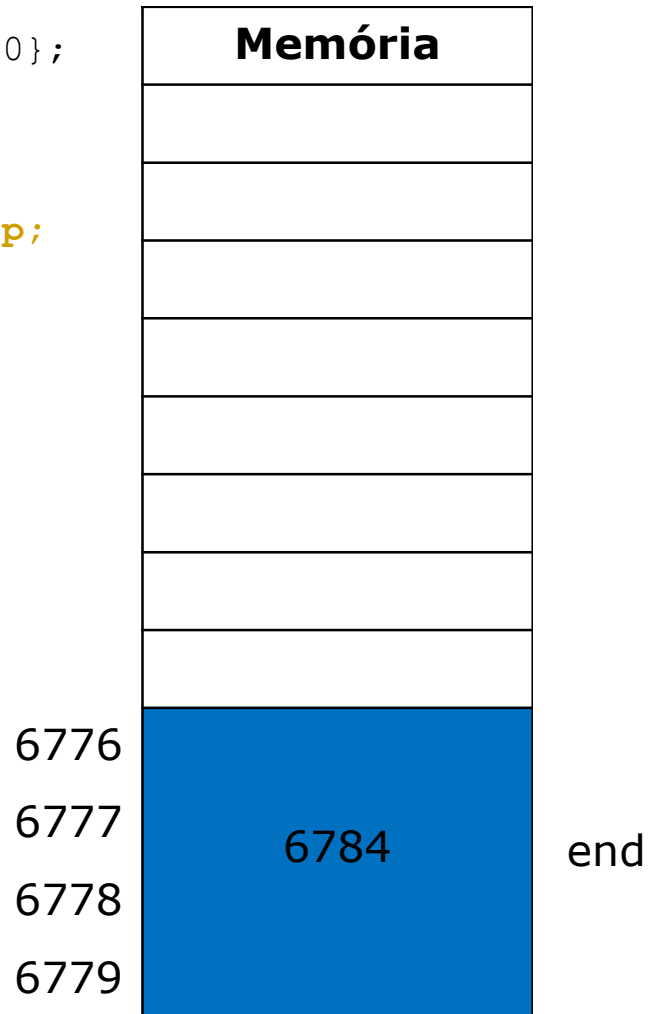


1. Ponteiros

```
x[] = {1, 10};
```

```
p = x;
```

```
end = (int)p;
```





1. Ponteiros

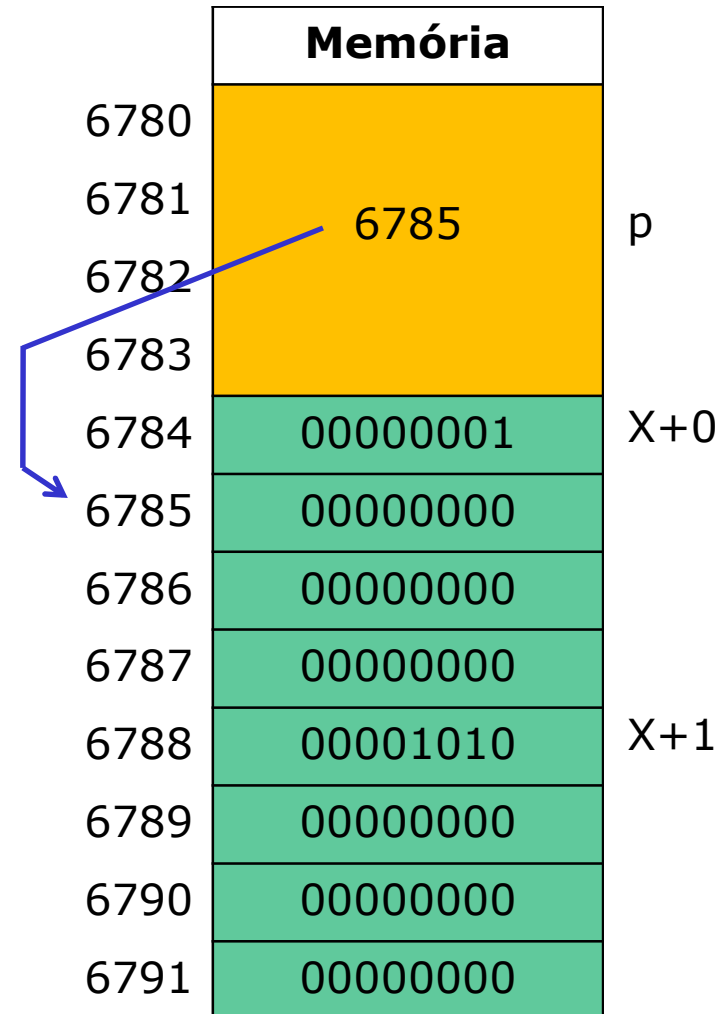
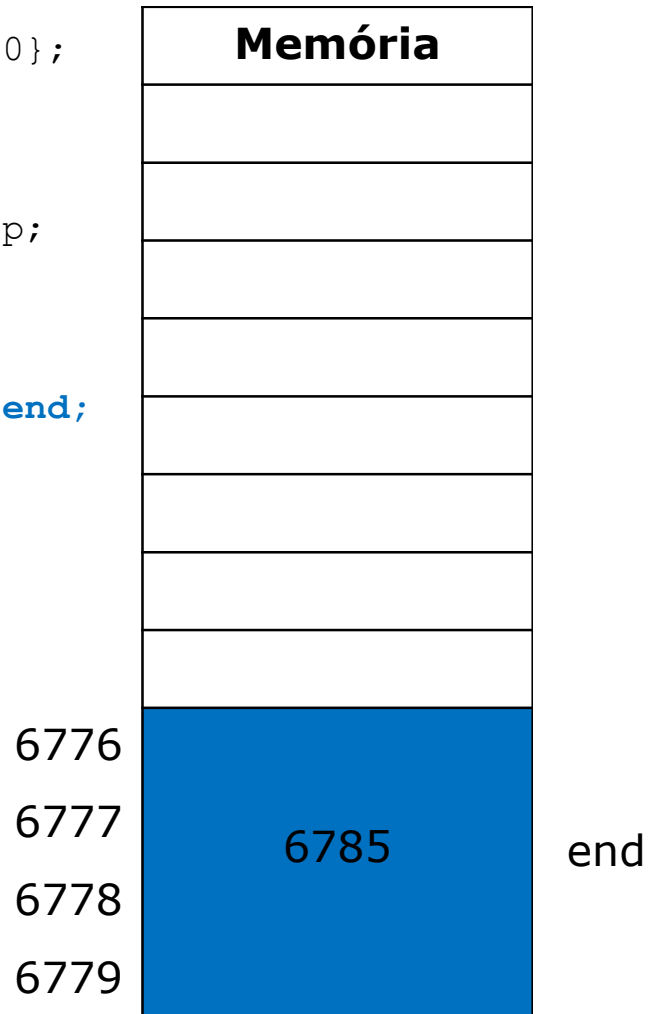
```
x[] = {1, 10};
```

```
p = x;
```

```
end = (int)p;
```

```
end++;
```

```
p = (int *)end;
```



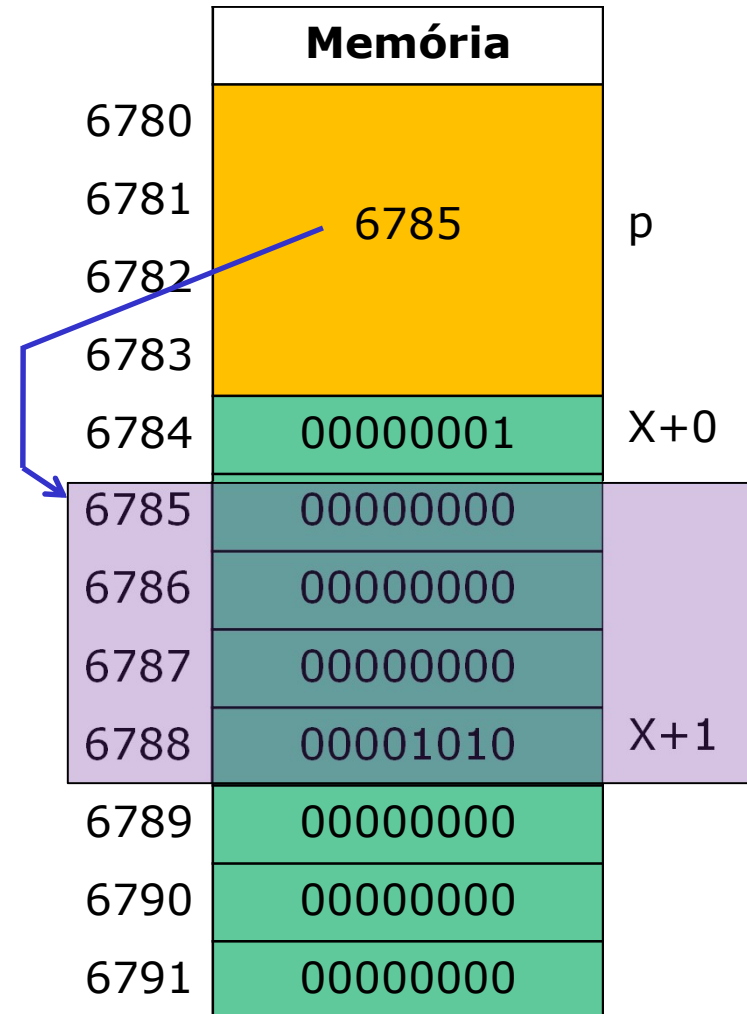


1. Ponteiros

```
x[] = {1, 10};  
  
p = x;  
  
end = (int)p;  
  
end++;  
  
p = (int *)end;  
  
printf("%d", *p);
```

$00001010\ 00000000\ 00000000\ 00000000_2 =$

167772160_{10}





2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
    int i, x[MAX] = {0,1,2,3};

    printf("Endereco\t Conteudo\t \n");

    for (i = 0; i<MAX; i++)
        printf("%d\t\t %d\t \n", &x[i], x[i]);

    return 0;
}
```



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
    int i, x[MAX] = {0,1,2,3};

    printf("Endereco\t Conteudo\t \n");
    printf("Notacao de vetor \n");
    printf("%d\t\t %d\t \n", &x[0], x[0]);
    printf("Notacao de ponteiro \n");
    printf("%d\t\t %d\t \n", x, *x);

    return 0;
}
```



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
    int i, x[MAX] = {0,1,2,3}, *p1;

    p1 = x;

    printf("Endereco\t Conteudo\t \n");
    printf("Notacao de vetor \n");
    printf("%d\t\t %d\t \n", &x[0], x[0]);
    printf("Notacao de ponteiro \n");
    printf("%d\t\t %d\t \n", p1, *p1);

    return 0;
}
```



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4
```

```
int main (int argc, char *argv[])
{
    int i, x[MAX] = {0,1,2,3}, *p1;

    p1 = x;

    printf("Endereco\t Conteudo\t \n");
    printf("Notacao de vetor \n");
    printf("%d\t\t %d\t \n", &x[2], x[2]);
    printf("Notacao de ponteiro \n");
    printf("%d\t\t %d\t \n", ???, ???);

    return 0;
}
```

E se eu quisesse
acessar o 3º
elemento?



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
    int i, x[MAX] = {0,1,2,3};

    printf("Endereco\t Conteudo\t \n");

    printf("Notacao de vetor:\n");
    for (i=0; i<MAX; i++)
        printf("%d\t\t %d\t \n", &x[i], x[i]);
}
```



2. Ponteiros e Vetores

```
printf("Notacao de ponteiro:\n");  
for (i=0; i<MAX; i++)  
    printf("%d\t\t %d\t \n", x+i, *(x+i));  
  
return 0;  
}
```



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
    int i, x[MAX] = {0,1,2,3}, *p1;

    p1 = x;

    printf("Endereco\t Conteudo\t \n");

    printf("Notacao de vetor:\n");
    for (i=0; i<MAX; i++)
        printf("%d\t\t %d\t \n", &x[i], x[i]);
```




2. Ponteiros e Vetores

```
printf("Notacao de ponteiro:\n");  
for (i=0; i<MAX; i++)  
    printf("%d\t\t %d\t \n", p1+i, *(p1+i));  
  
return 0;  
}
```



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
    int x[MAX], i, *p;

    p=x;

    for (i=0; i<MAX; i++)
        x[i]=i;

    for (i=0; i<MAX; i++)
        printf("%d ", *(p+i));

    return 0;
}
```



2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 2

int main (int argc, char *argv[])
{
    int *x[MAX], var1, var2;

    var1 = 3;
    var2 = 4;

    x[0] = &var1;
    x[1] = &var2;
```



2. Ponteiros e Vetores

```
printf("&var1: %d \n", &var1);  
printf("&var2: %d \n", &var2);  
printf("var1: %d\n", var1);  
printf("var2: % d\n", var2);  
printf("x[0]: %d \n", x[0]);  
printf("x[1]: %d \n", x[1]);  
printf("*x[0]: %d\n", *x[0]);  
printf("*x[1]: % d\n", *x[1]);  
  
return 0;  
}
```



2. Ponteiros e Vetores

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

int main(int argc, char *argv[])
{
    int *x[2], y0[2] = {0,1}, y1[2] = {2,3} ;

    x[0] = y0;
    x[1] = y1;

    printf("Conteudo de x[0] = y0: %d \n", x[0]);
    printf("Conteudo de x[1] = y1: %d \n", x[1]);
    printf("Endereço do primeiro elemento do vetor x: %d \n", &x[0]);
    printf("Endereço do segundo elemento do vetor x: %d \n", &x[1]);
}
```



2. Ponteiros e Vetores

```
printf("Conteudo de y0[0]: %d \n", *(x[0] + 0)); //y0[0]
printf("Conteudo de y0[1]: %d \n", *(x[0] + 1)); //y0[1]
printf("Endereço de y0[0]: %d \n", (x[0] + 0)); //&y0[0]
printf("Endereço de y0[1]: %d \n", (x[0] + 1)); //&y0[1]

printf("Conteudo de y1[0]: %d \n", *(x[1] + 0)); //y1[0]
printf("Conteudo de y1[1]: %d \n", *(x[1] + 1)); //y1[1]
printf("Endereço de y1[0]: %d \n", (x[1] + 0)); //&y1[0]
printf("Endereço de y1[1]: %d \n", (x[1] + 1)); //&y1[1]

return 0;
}
```

2. Ponteiros e Vetores

Memória		
&x[1] 788	768	X[1] = y1 = &y1[0]
&x[0] 784	776	X[0] = y0 = &y0[0]
&y0[1] = y0+1 = x[0]+1 780	1	y0[1] = *(y0+1) = *(x[0]+1)
&y0[0] = y0+0 = x[0]+0 776	0	y0[0] = *(y0+0) = *(x[0]+0)
&y1[1] = y1+1 = x[1]+1 772	4	y1[1] = *(y1+1) = *(x[1]+1)
&y1[0] = y1+0 = x[1]+0 768	3	y1[0] = *(y1+0) = *(x[1]+0)



3. Ponteiros e Strings

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

int main (int argc, char *argv[])
{
    char *p = "Segunda-feira";

    printf("%s \n", p);

    return 0;
}
```




3. Ponteiros e Strings

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int main (int argc, char *argv[])
{
    char *x[MAX] = {"Segunda-feira",
                    "Terça-feira",
                    "Quarta-feira",
                    "Quinta-feira",
                    "Sexta-feira"};

    printf("%s \n", x[2]);

    return 0;
}
```



3. Ponteiros e Strings

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int main (int argc, char *argv[])
{
    char *x[MAX] = {"Segunda-feira",
                    "Terça-feira",
                    "Quarta-feira",
                    "Quinta-feira",
                    "Sexta-feira"};

    printf("%s \n", *(x+2));

    return 0;
}
```



4. Passagem de Parâmetros por Referência

- Suponha o seguinte código (cuja função *divpordois* utiliza passagem de argumentos por valor):

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

float divpordois( float );

int main (int argc, char *argv[])
{
    float x, y = 5.0;

    printf("y = %.2f \n", y);
    x = divpordois(y);
    printf("%.2f/2 = %.2f \n", y, x);

    return 0;
}
```

4. Passagem de Parâmetros por Referência

```
float divpordois (float n){  
    float result;  
    result = n/2;  
    return result;  
}
```

4. Passagem de Parâmetros por Referência

```
float divpordois (float n){  
    float result;  
    result = n/2;  
    return result;  
}
```

Essa molezinha vocês dominam!



4. Passagem de Parâmetros por Referência

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

void divpordois (float *);

int main (int argc, char *argv[])
{
    float y = 5.0;

    printf("y = %.2f \n", y);
    divpordois(&y);
    printf("y = %.2f \n", y);

    return 0;
}

void divpordois (float *n){

    *n = *n/2;
}
```



4. Passagem de Parâmetros por Referência

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

int divpordois (float *);

int main (int argc, char *argv[])
{
    float y = 5.0, sucesso;;

    printf("y = %.2f \n", y);

    sucesso = divpordois(&y);

    printf("y = %.2f \n", y);

    printf("sucesso = %d \n", sucesso);

    return 0;
}
```

4. Passagem de Parâmetros por Referência

```
int divpordois( float *n) {  
  
    *n = *n/2;  
  
    return 0;  
}
```




4. Passagem de Parâmetros por Referência

- “Retornando” vários valores:

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

int retornavarios (float *, int *);

int main (int argc, char *argv[])
{
    float y = 5.0;
    int x = 5, sucesso;

    printf("y = %.2f - x = %d \n", y, x);

    sucesso = retornavarios(&y, &x);

    printf("y = %.2f - x = %d \n", y, x);
    printf("sucesso = %d \n", sucesso);
}
```



4. Passagem de Parâmetros por Referência

```
    return 0;
}

int retornavarios( float *n1, int *n2){

    *n1 = *n1/2;

    *n2 = *n2%2;

    return 0;
}
```



4. Passagem de Parâmetros por Referência

- Passando vetores por referência:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

int retornavetor (float *, int);

int main (int argc, char *argv[])
{
    float x[MAX] = {0,0,0,0,0,0,0,0,0,0};
    int i, sucesso;

    printf("Vetor antes de chamar a funcao\n");

    for (i = 0; i<MAX; i++)
        printf("%.2f \n", x[i]);

    sucesso = retornavetor(x, MAX);
```

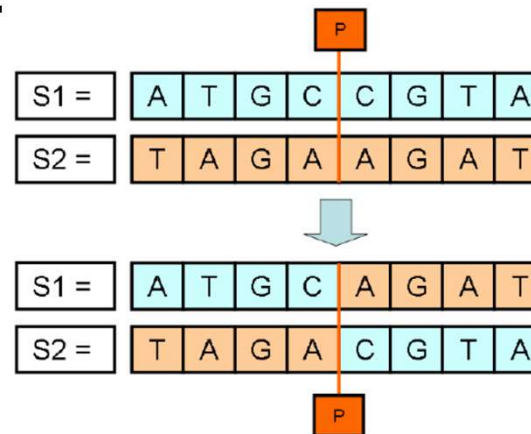


4. Passagem de Parâmetros por Referência

```
printf("Vetor depois de chamar a funcao\n");  
for (i = 0; i<MAX; i++)  
    printf("%.2f \n", x[i]);  
  
    return 0;  
}  
  
int retornavetor( float *vet, int N){  
  
    int i;  
  
    for (i = 0; i<N; i++)  
        *(vet+i) = i;  
  
    return 0;  
}
```

4. Passagem de Parâmetros por Referência

Exemplo: Um operador de *crossover* pode ser aplicado a duas strings *s1* e *s2* e consiste em se sortear aleatoriamente um ponto de *s1* e *s2* e, escolhido este ponto, é realizada a troca de informações de *s1* e *s2* tal como mostrado no esquema a seguir.



Escreva uma função que recebe duas strings *s1* e *s2* e realiza a operação de *crossover*. Escreva também um programa principal que utiliza a função proposta.



5. Retornando um Endereço de Memória

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

char *procuralettra(char *, char);

int main(int argc, char *argv[])
{
    char str[80], ch, *ptr;

    printf("Digite uma frase:");
    gets(str);

    printf("Digite um caractere:");
    ch = getchar();

    ptr = procuralettra(str, ch);
```



5. Retornando um Endereço de Memória

```
if( ptr != NULL){  
    printf("A primeira ocorrencia eh: %p \n", ptr);  
    printf("Sua posicao eh: %d \n", ptr-str);  
} else {  
    printf("Esse caractere nao existe nessa frase. \n");  
}  
  
    return 0;  
}  
  
char *procura letra(char *s, char c){  
    while(*s != c && *s != '\0')  
        s++;  
  
    if(*s != 0)  
        return s;  
  
    return NULL;  
}
```

6. Ponteiros e Matrizes

- Considere o código a seguir:

```
#include <stdio.h>
#include <stdlib.h>
#define LIN 3
#define COL 3

int main() {

    int m[LIN][COL];
    int i, j;

    for (i=0; i<LIN; i++){
        for (j=0; j<COL; j++){
            printf("Elemento %d %d = ", i, j);
            scanf("%d", &m[i][j]);
        }
    }
}
```




6. Ponteiros e Matrizes

```
// Notação de matriz
for (i=0; i<LIN; i++){
    for (j=0; j<COL; j++){
        printf("%d\t\t %d\t \n", &m[i][j], m[i][j]);
    }
}

return 0;
}
```

- Execute o programa e note que os elementos da matriz são organizados em posições consecutivas da memória.



6. Ponteiros e Matrizes

- Agora veja a notação de ponteiro:

```
#include <stdio.h>
#include <stdlib.h>
#define LIN 3
#define COL 3

int main() {

    int m[LIN][COL], *p;
    int i, j;

    for (i=0; i<LIN; i++){
        for (j=0; j<COL; j++){
            printf("Elemento %d %d = ", i, j);
            scanf("%d", &m[i][j]);
        }
    }
}
```

6. Ponteiros e Matrizes

```
p = &m[0][0];

// Notação de ponteiro
for (i=0; i<LIN; i++){
    for (j=0; j<COL; j++){
        printf("%d\t\t %d\t \n", p+i*COL+j, *(p+i*COL+j));
    }
}

return 0;
}
```

- Observe que o efeito é o mesmo.

7. Ponteiros e Structs

- Passando structs como parâmetro de funções.

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

// Estrutura
struct dados_aluno{
    char nome[80];
    float media;
};

// Protótipo da função que recebe estruturas
void imprime_struct( struct dados_aluno);
```



7. Ponteiros e Structs

```
int main ( int argc, char *argv[])  
{  
    struct dados_aluno aluno;  
  
    strcpy(aluno.nome, "Alexandre Zaghetto");  
    aluno.media = 9.5;  
  
    imprime_struct(aluno);  
  
    return 0;  
}  
  
// Imprime a estrutura  
void imprime_struct( struct dados_aluno parm){  
    printf ("%s \n", parm.nome);  
    printf ("%f \n", parm.media);  
}
```



7. Ponteiros e Structs

- Ponteiro para struct.

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

// Estrutura
struct dados_aluno{
    char nome[80];
    float media;
};

// Imprime a estrutura
void imprime_struct(struct dados_aluno *);

// Altera a estrutura
void altera_struct(struct dados_aluno *);
```



7. Ponteiros e Structs

```
int main (int argc, char *argv[])  
{  
    struct dados_aluno aluno, *p_aluno;  
  
    p_aluno = &aluno;  
  
    strcpy((*p_aluno).nome, "Alexandre Zaghetto");  
    (*p_aluno).media = 9.5;  
  
    printf("%s \n", (*p_aluno).nome);  
    printf("%f \n", (*p_aluno).media);  
  
    return 0;  
}
```



7. Ponteiros e Structs

```
int main (int argc, char *argv[])  
{  
    struct dados_aluno aluno, *p_aluno;  
  
    p_aluno = &aluno;  
  
    strcpy (p_aluno->nome, "Alexandre Zaghetto");  
    p_aluno->media = 9.5;  
  
    printf ("%s \n", p_aluno->nome);  
    printf ("%f \n", p_aluno->media);  
  
    return 0;  
}
```




7. Ponteiros e Structs

```
int main (int argc, char *argv[])  
{  
    struct dados_aluno aluno, *p_aluno;  
  
    p_aluno = &aluno;  
  
    strcpy(p_aluno->nome, "Alexandre Zaghetto");  
    p_aluno->media = 9.5;  
  
    imprime_struct(p_aluno);  
  
    altera_struct(p_aluno);  
  
    imprime_struct(p_aluno);  
  
    return 0;  
}
```



7. Ponteiros e Structs

```
// Imprime a estrutura
void imprime_struct(struct dados_aluno *parm) {
    printf ("%s \n", parm->nome);
    printf ("%f \n", parm->media);
}

// Altera a estrutura
void altera_struct(struct dados_aluno *parm) {
    strcpy (parm->nome, "Zaghetto Alexandre");
    parm->media = 5.9;
}
```



8. Ponteiro para Função

- Exemplo 1:

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

// Ponteiro para função

int pega_result(int, int, int (*) (int, int));
int max(int, int);
int min(int, int);
```



8. Ponteiro para Função

```
int main (int argc, char *argv[])  
{  
    int result, x1 = 10, x2 = 232;  
  
    result = pega_result(x1,x2, &max);  
  
    printf("O maximo entre %d e %d , %d\n",x1,x2, result);  
  
    result = pega_result(x1,x2, &min);  
  
    printf("O minimo de %d e %d , %d\n",x1,x2, result);  
  
    return 0;  
}
```



8. Ponteiro para Função

```
int pega_result(int a, int b, int (*compare)(int , int ))  
{  
    return compare(a, b); // Chama a função passada  
}
```

```
int max(int a, int b)  
{  
    printf("Em max:\n");  
    return (a > b) ? a: b;  
}
```

```
int min(int a, int b)  
{  
    printf("Em min:\n");  
    return (a < b) ? a: b;  
}
```



8. Ponteiro para Função

- Exemplo 1:

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

float funcao_pol(float);
float funcao_ln (float);
float funcao_sen(float);

float fun (float, float ( *) (float) );

typedef float (*Func) (float);
```



8. Ponteiro para Função

```
int main() {  
  
    Func funcoes [3] = {funcao_pol, funcao_ln, funcao_sen};  
    int n = 0, num_func;  
    float e, x0, x1, xm, fx0, fx1, fxm;  
  
    printf("Selecione e funcao:");  
    scanf("%d", &num_func);  
  
    printf("\n x0: ");  
    scanf("%f", &x0);  
    printf("\n x1: ");  
    scanf("%f", &x1);  
    printf("\n e: ");  
    scanf("%f", &e);  
}
```

8. Ponteiro para Função

```
do{  
  
    xm = (x0 + x1) / 2.0;  
  
    fx0 = fun(x0, funcoes[num_func]);  
    fx1 = fun(x1, funcoes[num_func]);  
    fxm = fun(xm, funcoes[num_func]);
```




8. Ponteiro para Função

```
if ((fx0 < 0) && (fx1 > 0)) {  
    if (fxm > 0) {  
        x1 = xm;  
    } else {  
        x0 = xm;  
    }  
} else {  
    if (fxm > 0) {  
        x0 = xm;  
    } else {  
        x1 = xm;  
    }  
}  
} while (fxm != 0 && fabs(x0-x1)>e );  
  
printf("Raiz: %f", xm);  
  
return 0;  
  
}
```



8. Ponteiro para Função

```
float funcao_pol(float x){  
  
    float a3, a2, a1, a0, fx;  
    static int set = 0;  
  
    if(x<=0){  
  
        printf("\\n Funcao polinomial : \\n");  
        printf("\\n a3: ");  
        scanf("%f", &a3);  
        printf("\\n a2 : ");  
        scanf("%f", &a2);  
        printf("\\n a1: ");  
        scanf("%f", &a1);  
        printf("\\n a0: ");  
        scanf("%f", &a0);  
        printf("\\n");  
    }  
    fx = ((a3 * (pow(x,3.0))) + (a2 * (pow(x,2.0))) + (a1 * x) + a0);  
    set++;  
    return fx;  
}
```

8. Ponteiro para Função

```
float funcao_ln (float x) {  
  
    float fx;  
  
    fx = x + log(x);  
  
    return fx;  
  
}
```

```
float funcao_sen(float x) {  
  
    float fx;  
  
    fx = 5-x-5*sin(x);  
  
    return fx;  
  
}
```

8. Ponteiro para Função

```
float fun (float x,    float (*funcao) (float)  ) {  
  
    return funcao(x);  
  
}
```

9. Alocação Dinâmica de Memória

- A alocação dinâmica permite ao programador alocar memória para variáveis enquanto o programa está sendo executado.
- É possível criar um vetor ou matriz cujo tamanho somente será definido em tempo de execução.
- A linguagem C define 4 funções para alocação dinâmica de memória, disponíveis na biblioteca `stdlib.h`:
 - ✓ *malloc()*: aloca memória;
 - ✓ *calloc()*: aloca memória;
 - ✓ *realloc()*: realoca memória; e
 - ✓ *free()*: libera memória alocada.

9. Alocação Dinâmica de Memória

- A função *malloc()* possui o seguinte protótipo:

```
void *malloc (size_t size);
```

- A função *malloc()* lê a quantidade **size** de *bytes* a alocar, reserva a memória correspondente e retorna o endereço do primeiro *byte* alocado.
- `size_t` é um tipo unsigned int.
- A função devolve um ponteiro do tipo *void*, o que significa que você pode atribuí-lo a qualquer tipo de ponteiro.
- Se não houver memória disponível para alocar, a função retorna um ponteiro nulo (NULL).

9. Alocação Dinâmica de Memória

- A função *calloc()* possui o seguinte protótipo:

```
void *calloc (size_t num, size_t size);
```

- A função *calloc()* lê a quantidade **num** de elementos a alocar, cada qual com um tamanho de **size** bytes, reserva (**num*size**) bytes, inicializa o espaço alocado com 0 e retorna o endereço do primeiro *byte* alocado.
- Se não houver memória disponível para alocar, a função retorna um ponteiro nulo (NULL).

9. Alocação Dinâmica de Memória

- A função *realloc()* possui o seguinte protótipo:

```
void *realloc (void *ptr, size_t size);
```

- A função *realloc()* realoca (expande ou contrai) um espaço de memória previamente alocado, apontado por *ptr*. O novo tamanho passa a ser **size bytes**.
- Se não houver memória disponível para realocar, a função retorna um ponteiro nulo (NULL).



9. Alocação Dinâmica de Memória

- A função *free()* possui o seguinte protótipo:

```
void free ( void * ptr );
```

- Desaloca um bloco de memória apontado por *ptr*, previamente alocado por meio das funções *malloc()*, *calloc()* ou *realloc()*.



9. Alocação Dinâmica de Memória

- Alocação de vetores:

- ✓ Exemplo de alocação para 30 valores do tipo double, utilizando *malloc()*:

```
double *p;
```

```
p = (double *) malloc(30 * sizeof(double));
```

- ✓ A operação realizada com o `(double *)` é chamada de *casting*. Ela garante que o ponteiro retornado pela função `malloc()` seja um ponteiro para double.

9. Alocação Dinâmica de Memória

- Alocação de vetores:

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

int main(){

    double *p;
    int N, numero, i;

    printf("Qual tamanho do vetor: ");
    scanf("%d", &N);

    p = (double *) malloc(N*sizeof (double));

    if (p == NULL){
        printf("Alocacao falhou. Finalizado.\n");
        exit(1);
    }
```



9. Alocação Dinâmica de Memória

```
for(i=0; i < N; i++)
{
    printf("Digite o valor %d do vetor: ", i);
    scanf("%lf", p+i);
    // scanf("%lf", &p[i]);
}

for(i=0; i < N; i++)
{
    printf("%.2lf \n", *(p+i));
    //printf("%.2lf \n", p[i]);
}

free(p);

return 0;
}
```



9. Alocação Dinâmica de Memória

```
destroi_vetor(p);

    system("PAUSE") ;
    return 0;
}

int *inicia_vetor(int N) {

    int i, *Vetor;

    Vetor = (int *)malloc(N*sizeof(int));

    for(i = 0 ; i<N; i++) Vetor[i] = 0;

    return Vetor;
}

void destroi_vetor(int *p) {
    free(p);
}
```

9. Alocação Dinâmica de Memória

- Alocação de memória com **calloc** e relocação com **realloc**:

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

int main(){

    double *p;
    int N, M, numero, i;

    printf("Qual tamanho do vetor: ");
    scanf("%d", &N);

    p = (double *) calloc(N, sizeof (double));

    if (p == NULL){
        printf("Alocacao falhou. Finalizado.\n");
        exit(1);
    }
```



9. Alocação Dinâmica de Memória

```
for(i=0; i < N; i++){
    printf("%.2lf \n", *(p+i));
} // calloc preenche o espaço alocado com 0's.

for(i=0; i < N; i++){
    printf("Digite o valor %d do vetor: ", i);
    scanf("%lf", p+i);
}

printf("Qual o novo tamanho do vetor: ");
scanf("%d", &M);

p = (double *) realloc(p, M*sizeof (double));

if (p == NULL){
    printf("Realocacao falhou. Finalizado.\n");
    exit(1);
}
```



9. Alocação Dinâmica de Memória

```
for(i=N; i < M; i++)
{
    printf("Digite o valor %d do vetor: ", i);
    scanf("%lf", p+i);
}

for(i=0; i < M; i++)
{
    printf("%.2lf \n", *(p+i));
}

free(p);

return 0;
}
```


9. Alocação Dinâmica de Memória



TIPO DE GRUPO: ABERTO - AMIGOS

	Mohinder Vader	
	Gangula Vader	
	Malloc Vader	
 	Sylar Vader	65
	Calloc Vader	58

9. Alocação Dinâmica de Memória

- O uso repetido de **malloc()**, **calloc()** ou **realloc()** e **free()** pode causar dois problemas:

- *Memory leak*
- Fragmentação da memória

9. Alocação Dinâmica de Memória

- *Memory leaks:*

✓ Um programa compilado em linguagem C cria e usa quatro regiões, logicamente distintas na memória, que possuem funções específicas:

Memória	Uso
Código do Programa	Instruções propriamente ditas e os dados só de leitura (por exemplo, constantes do programa).
Dados	Variáveis globais e estáticas (<i>static</i>).
<div>Pilha ↓</div>	Contém o endereço de retorno das chamadas de função, os argumentos para funções e variáveis locais.
<div>↑ Heap</div>	Região de memória livre destinada à alocação dinâmica.

9. Alocação Dinâmica de Memória

- *Memory leaks*:

✓ Se você alocar regiões de memória com `malloc()`, mas depois do uso não liberar estas regiões com o `free()`, temos caracterizado o *memory leak*.

Memória	Uso
Código do Programa	Instruções propriamente ditas e os dados só de leitura (por exemplo, constantes do programa).
Dados	Variáveis globais e estáticas (<i>static</i>).
<div>Pilha ↓</div>	Contém o endereço de retorno das chamadas de função, os argumentos para funções e variáveis locais.
<div>↑ Heap</div>	Região de memória livre destinada à alocação dinâmica.



9. Alocação Dinâmica de Memória

- *Memory leaks*: Exemplo 1.

```
int funcao(char *data) {  
  
    int *ptr = NULL;  
    int N = strlen(data), i;  
  
    ptr = (int *)malloc(N*sizeof(int));  
  
    if (N < 10)  
        return -1;  
    else  
        for(i = 0; i < N; i++) ptr[i] = i;  
  
    free(ptr);  
  
    return 0;  
}
```



9. Alocação Dinâmica de Memória

- *Memory leaks*: Exemplo 2.

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

int main(int argc, char *argv[])
{
    char *ptr1, *ptr2;

    ptr1 = (char *)malloc(10*sizeof(char));
    ptr1 = "Alexandre";
    ptr2 = (char *)malloc(10*sizeof(char));
    ptr2 = "Zaghetto";
    ptr1 = ptr2;

    free(ptr1);
    free(ptr2);
    return 0;
}
```

9. Alocação Dinâmica de Memória

- *Memory leaks*: Exemplo 3.

```
char *func ( )  
{  
    return malloc(20);  
}  
  
void callingFunc ( )  
{  
    func ( );  
    // Problema aqui  
    // O endereço de retorno  
    // não é armazenado.  
}
```

9. Alocação Dinâmica de Memória

- *Memory leaks*: Exemplo 4.



- Problema:

```
free(ptr1);
```

- Correto:

```
free(ptr1->ptr2);  
free(ptr1);
```



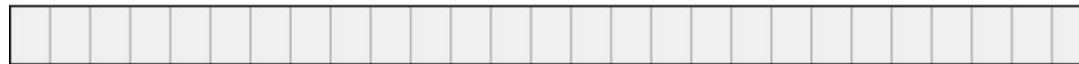

9. Alocação Dinâmica de Memória

- Fragmentação da memória:
 - *Heaps* acabam sendo compostas por **regiões de memória usadas** intercaladas com **regiões não usadas**, ou seja, a memória torna-se fragmentada.
 - Encontrar um espaço de memória livre do tamanho de que se necessita pode se tornar com o tempo um problema difícil.

9. Alocação Dinâmica de Memória

- Problema com *Heaps*:

Heap está inicialmente vazia.



Alocamos 7 bytes para foo.



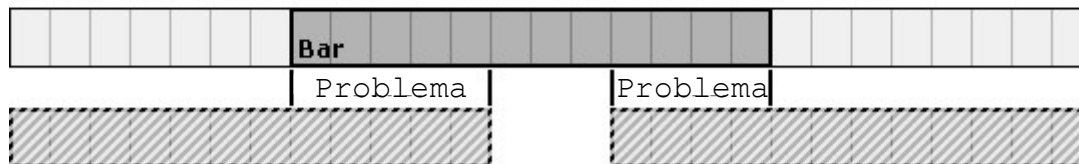
Alocamos 12 bytes para bar.



Desalocamos foo.



Alocar novos 12 bytes agora fica complicado.



9. Alocação Dinâmica de Memória

- Alocação de matrizes:
 - A alocação dinâmica de matrizes é realizada por meio das funções de manipulação de memória já apresentadas.
 - Pode ser feitas de duas maneiras:
 1. Utilizando um único ponteiro e “entendendo” os valores lidos como sendo elementos de uma matriz.
 2. Utilizando ponteiro para ponteiro.

9. Alocação Dinâmica de Memória

- Alocação de matrizes (utilizando um único ponteiro):

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

int main (int argc, char *argv[])
{

    int i,j, *mat;
    int Nlin, Ncol;

    printf("Digite o número de linhas da matriz:");
    scanf("%d", &Nlin);
    printf("Digite o número de colunas da matriz:");
    scanf("%d", &Ncol);

    mat = (int*) malloc(Nlin*Ncol*sizeof(int));
```

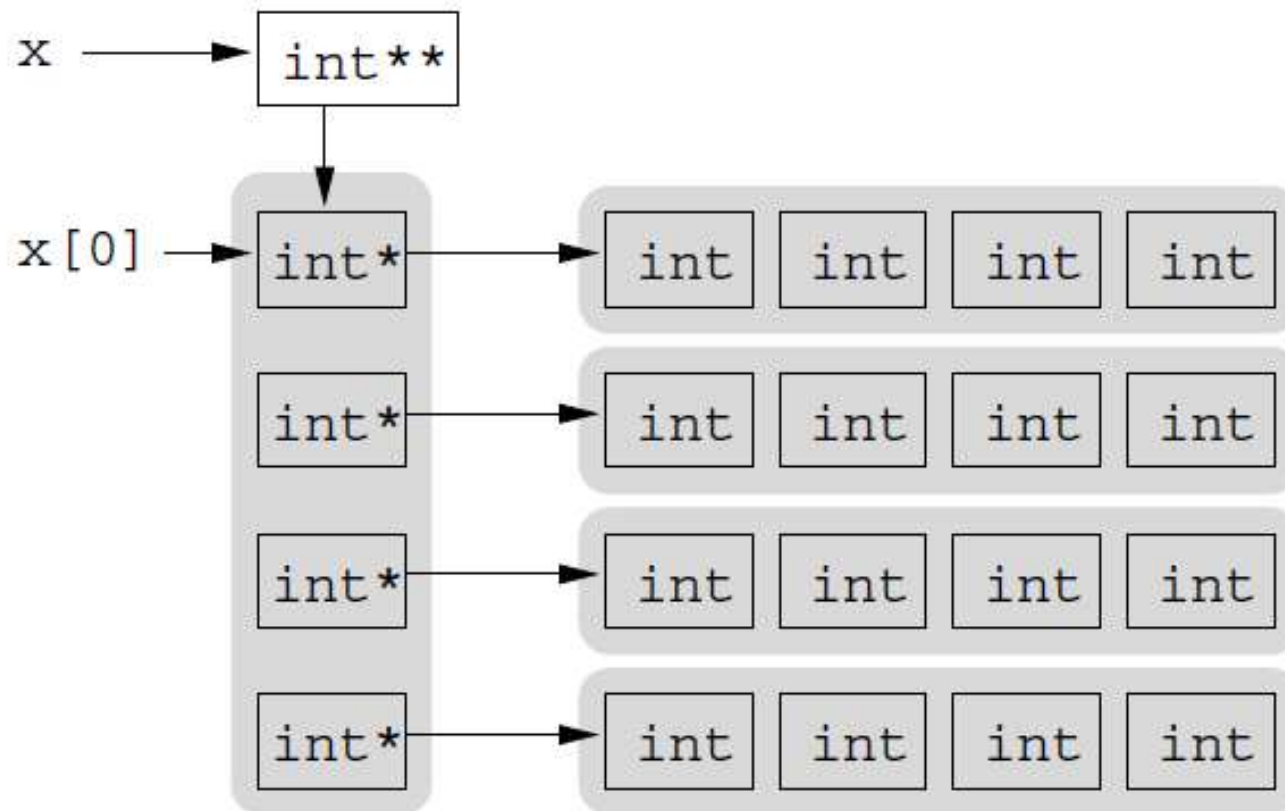


9. Alocação Dinâmica de Memória

```
for (i=0;i<Nlin;i++){  
    for (j=0;j<Ncol;j++){  
        {  
            printf("Digite o valor [%d][%d] da matriz:",i,j);  
            scanf("%d", mat+(i*Ncol)+j);  
        }  
    }  
  
    for (i=0;i<Nlin;i++){  
        for (j=0;j<Ncol;j++){  
            {  
                printf("MAT[%d][%d]: %d \n",i,j, *(mat+(i*Ncol)+j));  
            }  
        }  
    }  
  
    free(mat);  
  
    return 0;  
}
```

9. Alocação Dinâmica de Memória

- Alocação de matrizes (utilizando ponteiro para ponteiro):



9. Alocação Dinâmica de Memória

- Alocação de matrizes (utilizando ponteiro para ponteiro):

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

int main (int argc, char *argv[])
{
    int i,j, **mat;
    int Nlin, Ncol;

    printf("Digite o número de linhas da matriz:");
    scanf("%d", &Nlin);
    printf("Digite o número de colunas da matriz:");
    scanf("%d", &Ncol);

    mat = (int**) malloc(Nlin*sizeof(int *));

    for(i = 0; i<Nlin; i++)
        *(mat+i) = (int*) malloc(Ncol*sizeof(int));
    // mat[i] = (int*) malloc(Ncol*sizeof(int));
```



9. Alocação Dinâmica de Memória

```
for (i=0;i<Nlin;i++){  
    for (j=0;j<Ncol;j++){  
        printf("Digite o valor [%d][%d] da matriz:",i,j);  
        scanf("%d", *(mat+i)+j);  
        //scanf("%d",&mat[i][j]);  
    }  
}  
  
for (i=0;i<Nlin;i++){  
    for (j=0;j<Ncol;j++){  
        printf("MAT[%d][%d]: %d \n",i,j, (*(mat+i)+j));  
        //printf("MAT[%d][%d]: %d \n",i,j, mat[i][j]);  
    }  
}
```




9. Alocação Dinâmica de Memória

```
for (i=0;i<Nlin;i++)  
    free(* (mat+i));  
  
//for (i=0;i<Nlin;i++)  
//    free(mat[i]);  
  
free(mat);  
  
    return 0;  
}
```



9. Alocação Dinâmica de Memória

- Passando *matrizes* por referência:

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

void imprimematriz(int **, int, int);

int main(int argc, char *argv[])
{
    int L = 4, C = 3, **M;
    int i, j;

    M = (int **)malloc(L*sizeof(int *));

    for(i = 0; i < L; i++)
        *(M+i) = (int *)malloc(C*sizeof(int));
```



9. Alocação Dinâmica de Memória

```
for(i = 0; i<L; i++)
    for(j = 0; j<C; j++)
        M[i][j] = i*j;

imprimematriz(M, L, C);

return 0;
}

void imprimematriz(int **M, int L, int C){

    int i, j;

    for(i = 0; i<L; i++){
        for(j = 0; j<C; j++) printf("%d ", *(M+i)+j));
        printf("\n");
    }
}
```