

Estruturas de Dados

Prof. Fábio Chicout



Fábio Chicout



- Professor desde 2010 (UNIBRATEC, FACOL, UNIBRA, FSM)
 - Técnico, Graduação, Pós-Graduação (CC, ADS, Matemática)
- Quase Bacharel em Matemática, Graduação em ADS (IFPE), Mestrado em Web Semântica e BD (CIn/UFPE)
- Analista de TI UFPE 2009-atualmente
 - Diretor de Conectividade: 03/2020-02/2022
 - Assessor Técnico de Convênios: 02/2022-atualmente
- Áreas de atuação:
 - Arquitetura de Software; SRE; Gestão de Identidades;
 - Compras públicas, Gestão de TI

Público

- 2º período
- Currículo básico/científico
- Desenvolvedores/DBA/Cientistas

Horários de Aula

Sextas Feiras

1ª Parte: **18:30 - 20:10**

Intervalo: **10 min**

2º Parte: **20:20 - 22:00**

Materiais

- Slides
- Listas de Exercícios
- Livros
- Prática!
- <http://cslibrary.stanford.edu/101/>
- Classroom: **dv4xiqg**

Programa em C

- Detalhes adicionais: função main

- Por padrão a função principal deve ser do tipo **int** (inteiro) e retornar valor zero ao final de sua execução (significa que o programa terminou sem erros);
- Veremos mais sobre isso futuramente.

```
#include <stdio.h>

int main()
{
    //este programa calcula o cubo de um numero
    int num = 0;
    int cubo = 0;
    printf("Cubo de um numero\n\n");
    printf("Digite um numero: ");
    scanf("%d", &num);

    cubo = num*num*num;
    printf("\nCubo de %d = %d\n", num, cubo);

    return 0;
}
```

Comentários em Programas

- Utilizados para documentação para facilitar entendimento
- Podem ser colocados em qualquer parte do programa
- Compilador ignora
- Dois tipos:
 - Linha: // texto do comentário
 - Bloco:

```
/*  
    texto do comentário  
    texto do comentário  
    texto do comentário  
*/
```

Exercícios

- Calcule a média aritmética de 3 números dados
- Calcule o antecessor e sucessor de um número
- Calcule a área de um círculo de raio r
- Converta de Celsius para Fahrenheit

Variáveis e Constantes

```
#include <stdio.h>

int main()
{
    //este programa calcula o cubo de um numero
    int num = 0;
    int cubo = 0;
    printf("Cubo de um numero\n\n");
    printf("Digite um numero: ");
    scanf("%d", &num);

    cubo = num*num*num;
    printf("\nCubo de %d = %d\n", num, cubo);

    return 0;
}
```

Variáveis

Variáveis e Constantes

- Armazenam os dados dos programas em memória
- Armazenam um tipo de dado (inteiro, real ou caractere)
- Possuem um identificador (nome) para referenciar o seu conteúdo
- Declaração:

Sintaxe

```
<tipo> <identificador_1> [, identificador_2, ...];
```

Declarando Variáveis

Tipo: **short int**

Identificador: **dia3**

Valor inicial: **10**

```
1  #include <stdio.h>
2
3  main()
4  {
5      //Exemplos de declaração de inteiros
6      int inteiro;
7      short int dia1, dia2;
8      short dia3 = 10;
9
10     //Exemplo de declaração de reais
11     float salarioFuncionario = 1000.00;
12     double salarioChefe;
13
14     //Exemplo de declaração de caracteres
15     char ch;
16     char letra = 'a';
17 }
```

```
int cubo = 5;
```

Memória do Computador									
0:		1:		2:		3:		4:	
5:		6:		7:		8:		9:	
10:		11:		12:		13:		14:	
15:		16:		17:		18:		19:	

```
int cubo = 5;
```

1. Separe uma área da memória para armazenar o tipo da variável (inteiro: 32 bits)

Memória do Computador									
0:		1:		2:		3:		4:	
5:		6:		7:		8:		9:	
10:		11:		12:		13:		14:	
15:		16:		17:		18:		19:	

área reservada
de 32 bits

```
int cubo = 5;
```

1. Separe uma área da memória para armazenar o tipo da variável (inteiro: 32 bits)
2. Nomeie essa área de “cubo”

Memória do Computador									
0:		1:		2:		3:		4:	
5:		6:		7:		8:		9:	
10:		11:		12:		13:		14:	
15:		16:		17:		18:		19:	

← cubo

```
int cubo = 5;
```

1. Separe uma área da memória para armazenar o tipo da variável (inteiro: 32 bits)
2. Nomeie essa área de “cubo”
3. Coloque nessa área o número 5 (em binário: 101)

Memória do Computador									
0:		1:		2:		3:		4:	
5:		6:		7:		8:		9:	101
10:		11:		12:		13:		14:	
15:		16:		17:		18:		19:	

← cubo

Constantes

Sintaxe

```
const <tipo> <constante1> [, <constante2>, ...];
```

- Ao contrário das variáveis, constantes armazenam valores fixos

```
1 | #include <stdio.h>
2 |
3 | int main()
4 | {
5 |     const float pi;
6 |     pi = 3.14;    //Primeira inicialização
7 |     pi = 3.1415; //Erro!!!
8 |
9 |     const int diasMes = 30;
10 |    diasMes = 31; //Erro!!!
11 | }
12 |
```

- Após a primeira inicialização (que pode ser na própria declaração) uma constante não pode ter seu valor alterado.

Constantes de preprocessor

Sintaxe

```
#define <CONSTANTE> <VALOR>
```

- Em C, a diretiva **#define** é frequentemente utilizada ao invés de **const**.
- Neste caso, antes da compilação todas as ocorrências do nome da CONSTANTE são substituídas pelo VALOR definido.
- O tipo será inferido em tempo de compilação.

Obs.1: por convenção devem ser utilizadas letras maiúsculas no nome de constantes e underscore para separar palavras.

Obs.2: constantes deste tipo sempre tem escopo global

```
1  #include <stdio.h>
2
3  #define PI 3.1415
4  #define DIAS_MES 31
5
6  int main()
7  {
8      float dobroPI = 2*PI;
9      int doisMeses = 2*DIAS_MES;
10 }
```

Tipos de Dados

- Variáveis armazenam tipos de dados

- Quatro tipos de dados:

- Inteiro (**int**)
- Real (**float**, **double**)
- Caractere (**char**)
- Indefinido (**void**)

Declaração de variável

```
<tipo> <identificador_1> [, identificador_2, ...];
```

- Não possui o tipo lógico, que armazena verdadeiro ou falso: tipo **int** com valores (0: falso, ≠0: verdadeiro)
- Para cada tipo de dado, é necessária uma quantidade de bits para armazená-lo na memória

Tipos de Dados: Inteiro

- Os números inteiros, em C, se dividem em três tipos:

Tipo	Tamanho	Intervalo Suportado
short int (short)	16 bits	-32.768 a +32.767
int	32 bits	-2.147.483.648 a + 2.147.483.647
long int (long)	64 bits	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807

- Obs1.: O tipo **char** às vezes é utilizado com finalidade de representar um inteiro de 8 bits (0 a 255).
- Obs2.: O tamanho pode variar de acordo com o compilador ou com a plataforma para qual o programa está sendo compilado.

Tipos de Dados: Inteiro

- Tipo deve comportar o valor a ser armazenado
 - Ex: idade de um funcionário -> **short**
 - Ex: quantidade de eleitores de uma cidade grande -> **int**
- Podem ser combinados com o modificador **unsigned** (sem sinal)
 - Duplica o valor máximo que pode ser armazenado, iniciando a representação do zero (deixando de representar valores negativos).
 - Ex.: **unsigned short**, **unsigned int** ou **unsigned long**
- Para facilitar nosso estudo, sempre será usado o tipo **int** para armazenar os dados inteiros.

Tipos de Dados: Real

- Os números reais, em C, podem ser de dois tipos:

Tipo	Tamanho	Intervalo Suportado
float	32 bits	3.4E-38 a 3.4E+38
double	64 bits	1.7E-308 a 1.7E+308

- O tamanho pode variar de acordo com o compilador ou com a plataforma para qual o programa está sendo compilado.

Tipos de Dados: Caractere

- Tipo **char**
- Caractere alfa numérico (a, b, c,...z, A, B, C,...Z, 0...9) ou especial (como por exemplo: ; # ? @ ! < ?)
- O tipo **char** armazena um único caractere
- Ocupa 8 bits de memória
- Representado entre apostrofes: `char letra = 'a';`
- Sequência de caracteres (string): `char carro[] = "ferrari";`

Qual o Tipo?

- Número de quartos de um apartamento
- Peso
- Temperatura
- Número de alunos na disciplina

Identificadores

- Distinção de maiúsculas e minúsculas (case sensitive)
 - Ex: os identificadores: `Media`, `MEDIA`, `MediA` e `media` são considerados diferentes
- DICA: Boa Prática de Programação
 - Escolham bem os nomes das variáveis e constantes do programa.
 - Os identificadores escolhidos devem ser claros, a fim de explicitar o conteúdo que será armazenado, mas também não devem ser extensos para não dificultar a escrita.
 - Evite nomes como `a`, `b` e `c`, `num1`, `num2` (a não ser que façam sentido no contexto onde serão utilizados)

Operador de Atribuição (=)

- Armazenar um valor em uma dada variável ou constante (espaço de memória associado)
- Dado a ser armazenado deve ser compatível com o tipo da variável
 - Por exemplo, as variáveis reais podem receber valores reais e inteiros.
 - No entanto, uma variável inteira não pode receber um valor real (cuidado! o valor será convertido para inteiro podendo gerar resultados estranhos).

```
1  #include <stdio.h>
2
3  void main()
4  {
5      float pi = 3.14; //ok
6      int epsilon = 2.71; //Será atribuido 2 e não 2.71
7  }
```

Operador de Atribuição (=)

- Exemplo:
 - Define uma posição de memória chamada `x` para armazenar inteiros
 - Armazena o valor 5

```
1  #include <stdio.h>
2
3  void main()
4  {
5      int x;
6
7      x = 5;
8  }
```

Memória									
0:		1:		2:		3:		4:	
5:		6:		7:		8:		9:	x = 101
10:		11:		12:		13:		14:	
15:		16:		17:		18:		19:	

Operador de Atribuição (=)

- Pode ser usado em qualquer expressão válida em C
- Representado pelo símbolo de igual: =
- Forma geral: <nome_da_variável> = <expressão>

```
int main()
{
    int x;

    1 = x;
}
```



```
int main()
{
    int a = 2;
    int b = 3;

    a = b;
}
```

Qual o valor
de a e b?

```
int main()
{
    int a = 2;
    int b = 3;

    b = a;
}
```

```
1  #include <stdio.h>
2
3  void main()
4  {
5      int x;
6
7      x = 5;
8  }
```

- A ordem é importante! Atribuição, sempre da direita pra esquerda: □

Operadores Aritméticos

- Operadores aritméticos binários
- Dois operandos
- Notação: <operando> <operador> <operando>. Ex.: 4 * 2

Sinal	Ação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão (só para inteiros)

Conversões de Tipo

- Existem conversões automáticas de valores em uma avaliação de uma expressão quando operandos possuem tipos diferentes
 - Operando de tipo de menor tamanho é convertido automaticamente para o tipo de maior tamanho
 - Conversão é feita em área temporária da memória antes da avaliação da expressão
 - Resultado é novamente convertido para o tipo da variável à esquerda da atribuição

int a = 3/2.0 + 0.7;

1. O inteiro 3 é convertido para **float**;
2. Expressão é avaliada como 2.0,
3. Valor é convertido para um inteiro e atribuído à variável.

O resultado final de a é 2

Operadores de Conversão (Cast)

- Forma geral (os parênteses são necessários):
 $(\text{<tipo desejado>}) \text{<variável>}$ ou $(\text{<tipo desejado>}) (\text{<expressão>})$
- O armazenamento de um valor real em um tipo de dado inteiro gera erro ou perde-se precisão
 - `int a = 3/2 + 0.5;`
 - Resultado: `a` é 1
 - `int a = ((float)3)/2 + 0.5;`
 - Resultado: `a` é 2

Quais serão os valores das variáveis declaradas após a avaliação das expressões abaixo?

```
int a, r, s;  
double b, c;  
a = 3.5;  
b = a / 2.0;  
c = 1/2 + b;  
r = 10 % a;  
s = r + 2 * 3;
```

Resposta: a=3, b=1.5, c=1.5, r=1 e s = 7

Condicionais

- Comandos que permitem decidir se a execução de uma instrução deve ou não ser feita
- Baseada em **expressões booleanas**
 - Resultado da avaliação: **verdadeiro** ou **falso**
 - Em C, **NÃO** existe o tipo de dado *booleano*
 - **Falso** é representado como o inteiro 0 (zero)
 - Qualquer outro número diferente de zero indica **verdadeiro**.
 - Consideremos então:
 - 1: verdadeiro
 - 0: falso
 - Uma expressão booleana é composta de operandos booleanos (lógicos) e operadores **relacionais e/ou lógicos**

Condicionais

- Operadores relacionais:

Operador	Ação
<	<i>menor que</i>
>	<i>maior que</i>
<=	<i>menor ou igual que</i>
>=	<i>maior ou igual que</i>
==	<i>igual a</i>
!=	<i>diferente de</i>

Resultado de Comparação

Falso ou Verdadeiro

4 < 5 é verdadeiro (valor 1)

3 >= 10 é falso (valor 0)

Condicionais

- Operadores lógicos:

- São usados para combinar comparações
- Operam sobre valores booleanos (0 ou 1)

Operador	Ação
&&	<i>E</i>
	<i>Ou</i>
!	<i>Negação</i>

Resultado da Avaliação

```
int a , b ;  
int c = 23 ;  
int d = 27 ;  
a = ( c < 20 ) || ( d > c ) ;  
b = ( c < 20 ) && ( d < c ) ;
```

Qual será o valor
de a e b?

a = 1

b = 0

Expressões Booleanas

- Uma tabela verdade representa todas as combinações **verdadeiro-falso** dos operadores lógicos:

a	b	a && b	a b	!a
verdadeiro	verdadeiro	verdadeiro	verdadeiro	falso
verdadeiro	falso	falso	verdadeiro	falso
falso	verdadeiro	falso	verdadeiro	verdadeiro
falso	falso	falso	falso	verdadeiro

Expressões Booleanas

- Operadores **&&** e **||** são ditos ***short-circuited***
 - Operandos da direita só são avaliados, se necessário
- Erros comuns:
 - Confundir **&&** com **&**
 - Confundir **||** com **|**

Comandos Condicionais

- A linguagem C oferece 3 tipos de comandos condicionais:
 - if – else
 - switch
 - comando ternário (?:)

If-else

```
if (expressaoBooleana) {  
    // comandos  
} else {  
    // outros comandos  
}
```

- Se a avaliação de **expressaoBooleana** retornar **verdadeiro**:
 - **comandos** são executados
- Caso contrário:
 - executam-se **outros comandos**

If-else

- Exemplo:

```
1  #include <stdio.h>
2
3  void main()
4  {
5      float n1, n2, n3, m;
6      printf ("\nEntre com 3 notas " );
7      scanf ("%f %f %f", &n1, &n2, &n3);
8      m = (n1 + n2 + n3 ) / 3;
9      if (m >= 7.0) {
10         printf ("\n Aluno aprovado.");
11         printf (" Média igual a %f " , m);
12     } else {
13         printf ("\n Aluno reprovado.");
14         printf (" Média igual a %f ", m);
15     }
16 }
```

Obs.: A indentação facilita a leitura do programa

Variações: If-else

```
if (expressaoBoleana) {  
    comando;  
}
```

- Omissão do **else** quando não há comandos para o **else**:

```
if (expressaoBoleana)  
    comando;
```

- Omissão das chaves, quando há apenas um comando no **if** ou no **else**:

```
if (expressaoBoleana) {  
    comando1;  
    comando2;  
} else comando3;
```

```
if (expressaoBoleana)  
    comando;  
else {  
    comando1;  
    comando2;  
    ...  
}
```


If-else

- Outros exemplos:

```
1  #include <stdio.h>
2
3  void main ( )
4  {
5      int resposta ;
6      printf ("\n Qual o valor de 10 + 14?");
7      scanf ("%d", &resposta);
8      if (resposta == 10 + 14)
9          printf ("\n Resposta correta !");
10 }
```

If-else

- Comandos aninhados:

```
if (expressaoBoleana1) {  
    if (expressaoBoleana2)  
        comando1;  
    else  
        comando2;  
} else {  
    comando3  
}
```

```
if (expressaoBoleana1) {  
    comando1;  
} else {  
    if (expressaoBoleana2)  
        comando2;  
    else  
        comando3;  
}
```

- Obs.: O **else** é sempre associado ao **if** anterior mais próximo

If-else

- Outros exemplos de comandos **if** aninhados:

```
1  #include <stdio.h>
2
3  void main()
4  {
5      int temp ;
6      printf ("\nDigite a temperatura: ");
7      scanf ("%d", &temp) ;
8      if (temp < 30)
9          if (temp > 20)
10             printf ("\nTemperatura agradável");
11     else
12         printf ("\nTemperatura muito quente");
13 }
14
```

Há algo errado?

If-else

- Outros exemplos de comandos **if** aninhados:

```
1  #include <stdio.h>
2
3  void main()
4  {
5      int temp ;
6      printf ("\nDigite a temperatura: ");
7      scanf ("%d", &temp) ;
8      if (temp < 30) {
9          if (temp > 20)
10             printf ("\nTemperatura agradável");
11      } else
12         printf ("\nTemperatura muito quente");
13     }
14
```

O **else** é sempre associado ao **if** anterior mais próximo dentro do mesmo bloco { }

O comando switch

- Não é elegante muitas condições: **if-else** encadeados
- Para estes casos o comando **switch** pode ser a melhor opção

```
1  #include <stdio.h>
2
3  void main()
4  {
5      char operador;
6      float a, b;
7      float result = 0.0;
8      printf ("\n Informe os 2 números e a operação ");
9      scanf ("%f %f %c", &a, &b, &operador);
10     switch (operador) {
11         case '+': result = a + b;
12             break;
13         case '-': result = a - b;
14             break;
15         case '*': result = a * b;
16             break;
17         case '/': result = a / b;
18             break;
19         default : printf("\nOperador invalido");
20     }
21     printf("\nResultado igual a %f ", result);
22 }
```

O comando switch

```
15 switch(expressao) {  
16     case rotulo1:  
17         Comandos1  
18     break;  
19     case rotulo2:  
20         Comandos2  
21     break;  
22     ...  
23     default:  
24         Comandos  
25 }
```

Para executar um switch:

- Avalia-se **expressao**;
- Executam-se os comandos do case cujo rótulo é igual ao valor resultante da expressão;
- Executam-se os comandos de default caso o valor resultante não seja igual a nenhum rótulo;

O comando switch

- Expressão só pode ser: int ou **char**;
- Rótulos são constantes
- Existe no máximo uma cláusula **default** (é opcional);
- Os tipos dos rótulos têm que ser do mesmo tipo de **expressao**;
- Vários rótulos podem estar associados ao mesmo comando
 - Os comandos **break** são opcionais:
 - Sem o **break** a execução dos comandos de um rótulo continua nos comandos do próximo, até chegar ao final ou a um **break**.
 - No exemplo ao lado:
 - caso **expressao** seja avaliada para **rotulo1**, os comandos 1 e 2 serão executados.
 - caso **expressao** seja avaliada para **rotulo2**, apenas o comando 2 será executado.

```
switch (expressao) {  
    case rotulo1:  
        Comandos1;  
    case rotulo2:  
        Comandos2;  
    break;  
    case rotulo3:  
        Comandos3;  
}
```

O comando ternário de decisão (?:)

- O comando ternário (?:) é uma versão do **if-else** com sintaxe mais econômica;
- Sintaxe:
(condicao ? expressao1 : expressao2)
- Lê-se:
 - Caso **condicao** seja verdadeira: avalie e retorne como resultado **expressao1**, caso contrário, avalie e retorne como resultado **expressao2**

O comando ternário de decisão (?)

- Os exemplos abaixo são equivalentes:

```
#include <stdio.h>

int main()
{
    int num1, num2, max;
    printf("Digite dois numeros: ");
    scanf("%d %d", &num1, &num2);

    if (num1 > num2)
        max = num1;
    else
        max = num2;

    printf("O maior valor eh: %d", max);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int num1, num2, max;
    printf("Digite dois numeros: ");
    scanf("%d %d", &num1, &num2);

    max = (num1 > num2 ? num1 : num2);

    printf("O maior valor eh: %d", max);
    return 0;
}
```

Atividade 01

- Variáveis, Entrada e Saída

- Faça um programa que:

1. Leia dois valores do teclado e os armazena em duas variáveis;
2. Troque o conteúdo de uma variável com a outra;
3. Imprima os valores trocados na tela;

Atividade 02

- Variáveis, Entrada e Saída

- Faça um programa que leia do teclado cinco números e imprima na tela a soma destes cinco números. O programa só pode utilizar 2 (duas) variáveis.

Atividade 03

- Constantes, Variáveis, Entrada e Saída
 - Faça um programa que leia do teclado o raio de uma circunferência e imprima seu diâmetro, seu perímetro e sua área.
 - Obs.: Declare a constante π

Funções

Dividir para Conquistar

- Dividir um problema em subproblemas mais simples
- Os passos para isso são:
 1. Divisão do problema em subproblemas;
 2. Solução de cada um dos subproblemas;
 3. Composição das soluções dos subproblemas para solucionar o problema original.
- Chamado de programação modular

Programação Modular

- Vantagens:

- Módulos podem ser escritos uma vez apenas e reutilizados sempre que necessário
- Módulos podem ser compostos para solucionar problemas cada vez complexos



- Facilita a manutenção: um erro corrigido em um módulo reflete em todos os lugares onde esse módulo é utilizado;

Módulos em C - Funções

- **Função**: conjunto de instruções para realizar uma ou mais tarefas que são agrupadas em uma mesma unidade e que pode ser referenciada

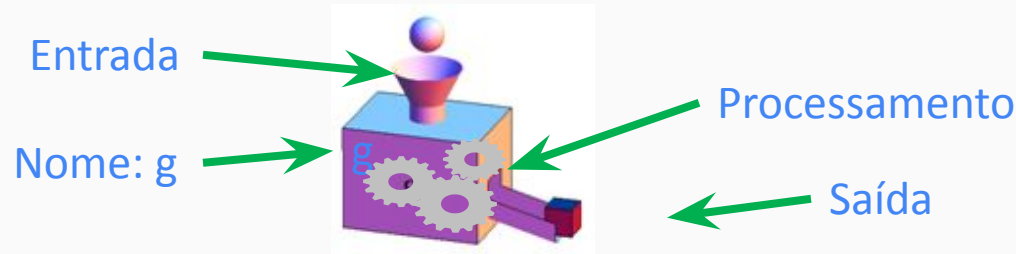
$$S = 1 + x - y + \frac{\sqrt{(x+y)^2}}{(x-y)^* 2!} - \frac{\sqrt{(x+y)^3}}{(x-y)^* 3!} + \dots$$

`pow(base,expoente)`

`fatorial(número)`

Funções em C

- Para criação, é necessário informar:
 - **Tipo das entradas (parâmetros):** tipos de dados dos dados que são necessários para executar sua função (opcional);
 - **Tipo da saída:** tipo de dados do resultado do processamento (opcional);
 - **Processamento:** transforma as entradas na saída desejada;
 - **Nome:** um identificador (segundo as regras para criação de identificadores para variáveis).



Exemplo da Sintaxe

Entradas e seus tipos:

1º parâmetro: float x

2º parâmetro: float a

3º parâmetro: float b

4º parâmetro: float c

Tipo de dados da
saída (retorno): float

```
float segundoGrau(float x, float a, float b, float c) {  
    float y;  
  
    y = a*x*x + b*x + c; // y = ax^2 + bx + c  
  
    return y;  
}
```

Processamento
“corpo da função”

Nome da função: “segundoGrau”

Retornar para a saída o
resultado do processamento

```

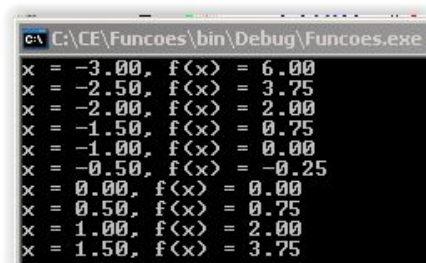
1  #include <stdio.h>
2
3  float segundoGrau(float x, float a, float b, float c) {
4      float y;
5
6      y = a*x*x + b*x + c; //y = ax^2 + bx + c
7
8      return y;
9  }
10
11  int main()
12  {
13      float x, y;
14
15      for (x=-3; x<2; x+=0.5) {
16          y = segundoGrau(x, 1, 1, 0); //y = x^2 + x
17
18          printf("x = %.2f, f(x) = %.2f\n", x, y);
19      }
20      return 0;
21  }

```

Declaração da função
"segundoGrau"

Chamada da função com passagem dos argumentos

Nota: **argumento** é o nome dado aos valores passados para os parâmetros de uma função.



```

C:\CE\Funcoes\bin\Debug\Funcoes.exe
x = -3.00, f(x) = 6.00
x = -2.50, f(x) = 3.75
x = -2.00, f(x) = 2.00
x = -1.50, f(x) = 0.75
x = -1.00, f(x) = 0.00
x = -0.50, f(x) = -0.25
x = 0.00, f(x) = 0.00
x = 0.50, f(x) = 0.75
x = 1.00, f(x) = 2.00
x = 1.50, f(x) = 3.75

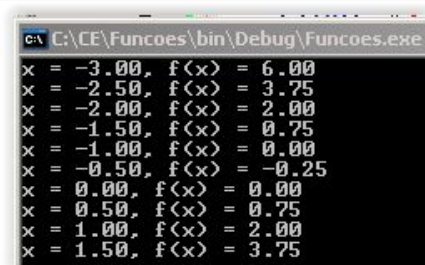
```

Exemplo de Utilização

- O programa anterior equivale a:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x, y;
6
7      for (x=-3; x<2; x+=0.5) {
8          y = 1*x*x + 1*x + 0; //y = x^2 + x
9
10         printf("x = %.2f, f(x) = %.2f\n", x, y);
11     }
12     return 0;
13 }
14
15
16
```

Note que “main” é também uma função. Todo programa em C é uma função que deve retornar um código inteiro. Valor zero para este código indica que o programa terminou sem erros, qualquer outro valor indica um código de erro com significado definido pelo programador.



```
C:\CE\Funcoes\bin\Debug\Funcoes.exe
x = -3.00, f(x) = 6.00
x = -2.50, f(x) = 3.75
x = -2.00, f(x) = 2.00
x = -1.50, f(x) = 0.75
x = -1.00, f(x) = 0.00
x = -0.50, f(x) = -0.25
x = 0.00, f(x) = 0.00
x = 0.50, f(x) = 0.75
x = 1.00, f(x) = 2.00
x = 1.50, f(x) = 3.75
```

O comando **return**

- Funções que retornam valores devem utilizar o comando **return**:

```
int segundos(int hora, int min) {  
    return 60 *(min + hora*60);  
}  
  
double porcetagem(double val,double tx) {  
    double valor = val*tx/100;  
    return valor;  
}
```

- Obs.: O comando **return** pode aparecer em qualquer ponto do corpo da função, e uma vez atingido, a execução da função é terminada:

```
double porcetagem(double val,double tx) {  
    double valor = val*tx/100;  
    return valor;  
    printf("O valor foi calculado\n");//<- Nunca será executado  
}
```

O comando return

- Utilização:
return expressão;
- Para executar este comando o programa:
 - Avalia expressão, obtendo um valor. Ex.: **return** (a*x*x+b*x+c);
- Uma função que não tem valor para retornar: **void**
Ou
 - Uso do **return** é opcional:

```
void imprimeMenu() {  
    printf("1: Dilma\n");  
    printf("2: Aecio\n");  
    printf("3: Branco\n");  
    printf("4: Invalido\n");  
}
```

```
void imprimeMenu() {  
    printf("1: Dilma\n");  
    printf("2: Aecio\n");  
    printf("3: Branco\n");  
    printf("4: Invalido\n");  
    return;  
}
```

Variações

- Algumas funções não precisam receber parâmetros. Neste caso, a lista de parâmetros fica vazia, mas os parênteses ainda são obrigatórios:

```
void imprimeMenu() {  
    printf("1: Dilma\n");  
    printf("2: Aecio\n");  
    printf("3: Branco\n");  
    printf("4: Invalido\n");  
}
```

Chamada ou Invocação de Funções

- Um programa em C, sempre inicia na função principal: `main()`;
- Apenas declarar uma função não fará com que ela seja executada
- Para que seja executada é necessário que ela seja **chamada** (invocada) -> fornecidos valores para os parâmetros
- Quando chamada, o fluxo de controle do programa é desviado para a função e o código que está nela é executado;
- Quando a função termina de ser executada, o fluxo de controle do programa **retorna** para a instrução logo após a chamada da função;
- O valor de retorno da função pode ser capturado e armazenado em uma variável utilizando o comando de atribuição '='.

Voltando ao exemplo:

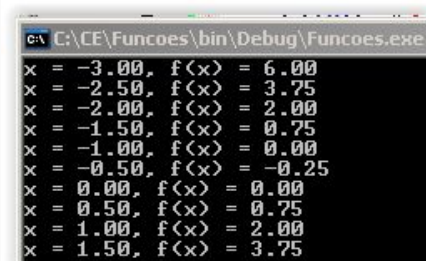
```
1  #include <stdio.h>
2
3  float segundoGrau(float x, float a, float b, float c) {
4      float y;
5
6      y = a*x*x + b*x + c; //y = ax^2 + bx + c
7
8      return y;
9  }
10
11  int main()
12  {
13      float x, y;
14
15      for (x=-3; x<2; x+=0.5) {
16          y = segundoGrau(x, 1, 1, 0); //y = x^2 + x
17          printf("x = %.2f, f(x) = %.2f\n", x, y);
18      }
19      return 0;
20  }
```

Declaração da função
"segundoGrau"

Chamada da função

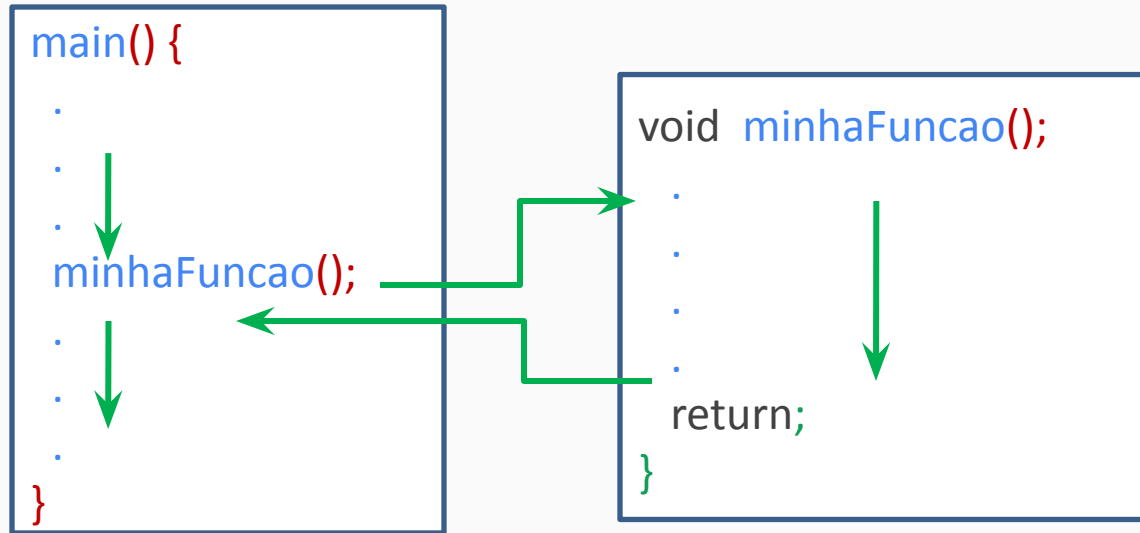
Passagem de valores para os parâmetros

Captura do valor retornado e
armazenamento da na variável y.

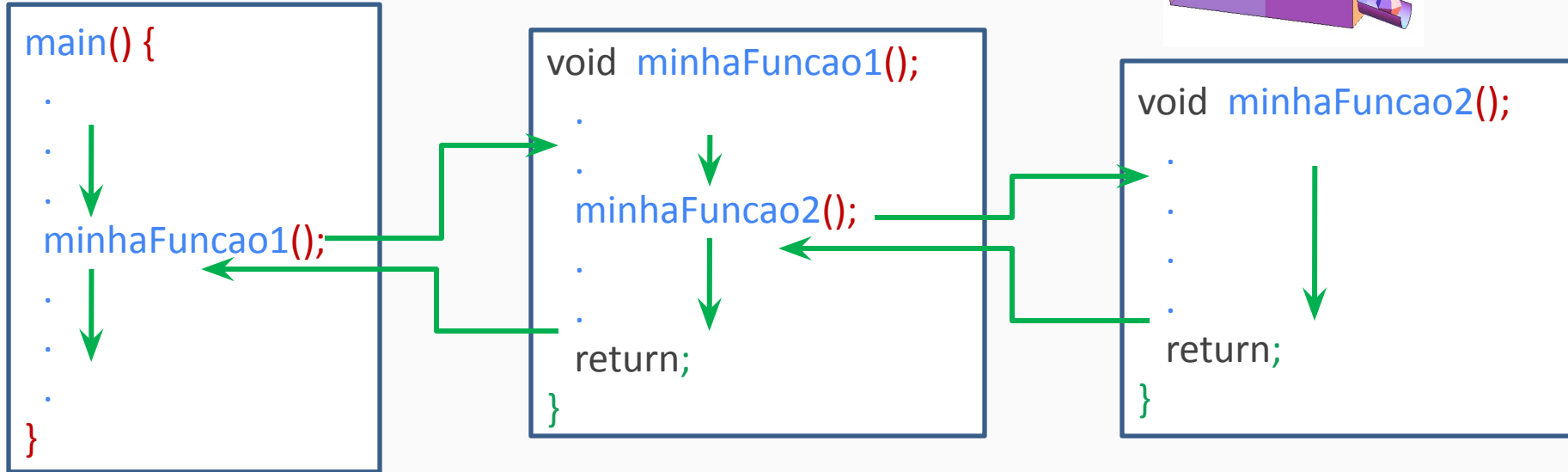
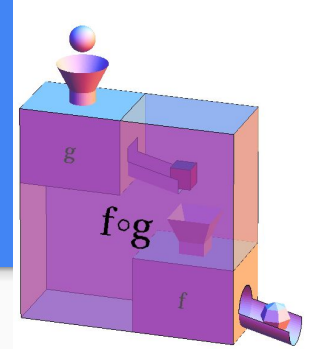


x	f(x)
-3.00	6.00
-2.50	3.75
-2.00	2.00
-1.50	0.75
-1.00	0.00
-0.50	-0.25
0.00	0.00
0.50	0.75
1.00	2.00
1.50	3.75

Desvio da Execução



Desvio da Execução



Escopo das Variáveis

- Define a área do programa onde esta variável pode ser referenciada
- Variáveis globais: declaradas fora das funções (inclusive fora da função **main**)
 - Podem ser referenciadas por todas as funções do programa abaixo do ponto onde foram declaradas
- Variáveis locais: declaradas dentro de uma função (inclusive dentro da função **main**)
 - Só podem ser referenciadas dentro desta função

Variáveis Globais

- Podem ser usadas em qualquer parte do código;
- Existem durante todo o ciclo de vida do programa (ocupando memória);
 - Se não forem explicitamente inicializadas, são inicializadas para zero pelo compilador.
- Normalmente declaradas no início do programa ou em arquivos do tipo header (*.h)
- Declaradas uma única vez
- Deve-se evitar o uso abusivo delas, pois:
 - Pode penalizar o consumo de memória;
 - Pode dificultar a legibilidade e manutenção do código (se pode ser acessada e alterada em qualquer lugar como encontrar onde está o erro?).

Variáveis Globais

```
1  #include <stdio.h>
2
3  int i; //Variável global
4
5  void incrementa() {
6      i++;
7  }
8
9  int main()
10 {
11     i = 0;
12     incrementa();
13     printf("Valor de i: %d", i);
14     return 0;
15 }
```

Variável global: declarada
fora de qualquer função

Acessível em qualquer
ponto do código após
sua declaração

Variáveis Locais

- Declaradas dentro de uma função
- Só existem durante a execução da função -> só ocupam a memória durante a execução da função
- Não são inicializadas automaticamente
- São visíveis apenas dentro da função onde foram declaradas
- Outras funções não podem referenciá-las
- Parâmetros de funções podem ser vistos como variáveis locais

Variáveis Locais

```
1  #include <stdio.h>
2
3  void incrementa() {
4      int i = 0;
5      i++;
6  }
7
8  int main()
9  {
10     i = 0;
11     incrementa();
12     printf("Valor de i: %d", i);
13     return 0;
14 }
15
```

Variável local: declarada
dentro de uma função

Não é acessível fora da
função onde foi
declarada.
Error: 'i' undeclared!

Parâmetros e Argumentos

- Os parâmetros são nomes que aparecem na declaração de uma função:

```
void imprimir(int valor)
```

- Os argumentos são expressões que aparecem na expressão de invocação da função:

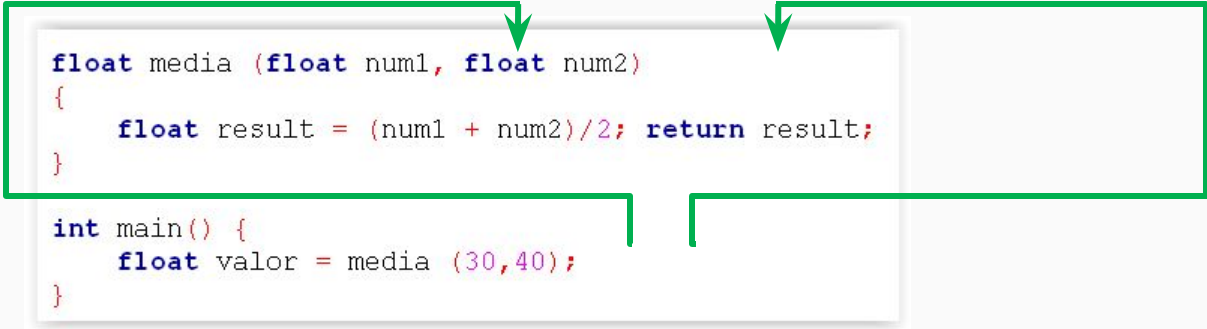
```
imprimir(10);
```

```
imprimir(8+2);
```

```
Imprimir(2*5);
```

Parâmetros e Argumentos

- Quando uma função é chamada, os argumentos da chamada são copiados para os parâmetros (formais) presentes na assinatura da função:



```
float media (float num1, float num2)
{
    float result = (num1 + num2)/2; return result;
}

int main() {
    float valor = media (30,40);
}
```

The diagram illustrates the flow of data from the `main()` function to the `media()` function. A green box highlights the function signature of `media()` and the call to `media()` in `main()`. Two green arrows originate from the arguments `30` and `40` in the `media()` call and point to the parameters `num1` and `num2` in the `media()` signature, respectively. Another green arrow points from the `media()` signature to the `main()` function, indicating the return value.

- Parâmetros são como variáveis locais da função (não é necessário declarar novamente)
- Não se deve declarar variáveis locais com o mesmo nome de parâmetros

Escopo das Variáveis

- Variáveis em escopos diferentes podem ter o mesmo nome, porém, referenciam endereços de memória diferentes!

```
1  #include <stdio.h>
2
3  void incrementa() {
4      int i = 0;
5      i++;
6  }
7
8  int main()
9  {
10     int i = 10;
11     incrementa();
12     printf("Valor de i: %d", i);
13     return 0;
14 }
15
```

Mesmo nome, porém
são variáveis distintas

Qual valor será impresso?

Escopo das Variáveis

- Uma variável de escopo local, com o mesmo nome de uma variável com escopo global oculta (sobrepe) a de escopo global.

```
1  #include <stdio.h>
2
3  int i = 0;
4
5  void incrementa() {
6      i++;
7      printf("Valor de i em incrementa: %d", i);
8  }
9
10 int main()
11 {
12     int i = 10;
13     incrementa();
14     printf("Valor de i na main: %d", i);
15     return 0;
16 }
```

Referência a variável global

A variável de escopo local na main, sobrepe a de escopo global

Referência a variável local

Quais valores serão impressos?

Ordem da Definição de Funções

- Onde uma função deve ser definida?
 - Antes da **main**; ou
 - Depois da **main**, desde que sua assinatura seja declarada antes da **main**.
- A assinatura de uma função deve indicar:
 - seu nome;
 - Os tipos das entradas;
 - O tipo da saída.
- Ex.: Função "segundos":
 - Transforma horas e minutos
 - em segundos.
- Assinatura da função "segundos":
 - O nome dos parâmetros é opcional:

```
int segundos(int hora, int min) {  
    return 60 * (min + hora*60);  
}
```

```
int segundos(int, int);
```

```
int segundos(int hora, int min)
```

Ordem da Definição de Funções

- Onde uma função deve ser definida?

- Antes da **main**:

```
int segundos(int hora, int min) {  
    return 60 * (min + hora*60);  
}  
  
int main() {  
    int minutos, hora, seg ;  
    printf("Digite a hora:minutos\n");  
    scanf ("%d:%d",&hora,&minutos) ;  
    seg = segundos(hora,minutos);  
    printf("\n%d:%d tem %d segundos.",hora,minutos,seg);  
    return 0 ;  
}
```

Ordem da Definição de Funções

- Onde uma função deve ser definida?
 - Depois da **main** com declaração prévia da assinatura:

```
int segundos(int, int);

int main() {
    int minutos, hora, seg;
    printf("Digite a hora:minutos\n");
    scanf ("%d:%d",&hora,&minutos);
    seg = segundos(hora,minutos);
    printf("\n%d:%d tem %d segundos.",hora,minutos,seg);
    return 0;
}

int segundos(int hora, int min) {
    return 60 *(min + hora*60);
}
```

Assinatura da função antes da chamada

Chamada da função

Declaração da função após a chamada.

A regra básica é que o compilador precisa encontrar a definição de uma função ou sua assinatura antes de encontrar sua chamada

Estruturas (Structs)

Tipos de Dados Primitivos vs Estruturados

- Tipos primitivos: reais (float, double), inteiros (int), caractere (char);
- Tipos estruturados: informações são compostas por diversos campos com tipos diferentes
- São chamados de:
 - Tipos de dados estruturados ou registros (em c: struct)

Tipos de Dados Estruturados

- Permitem agrupar conjuntos de tipos de dados distintos sob um único nome

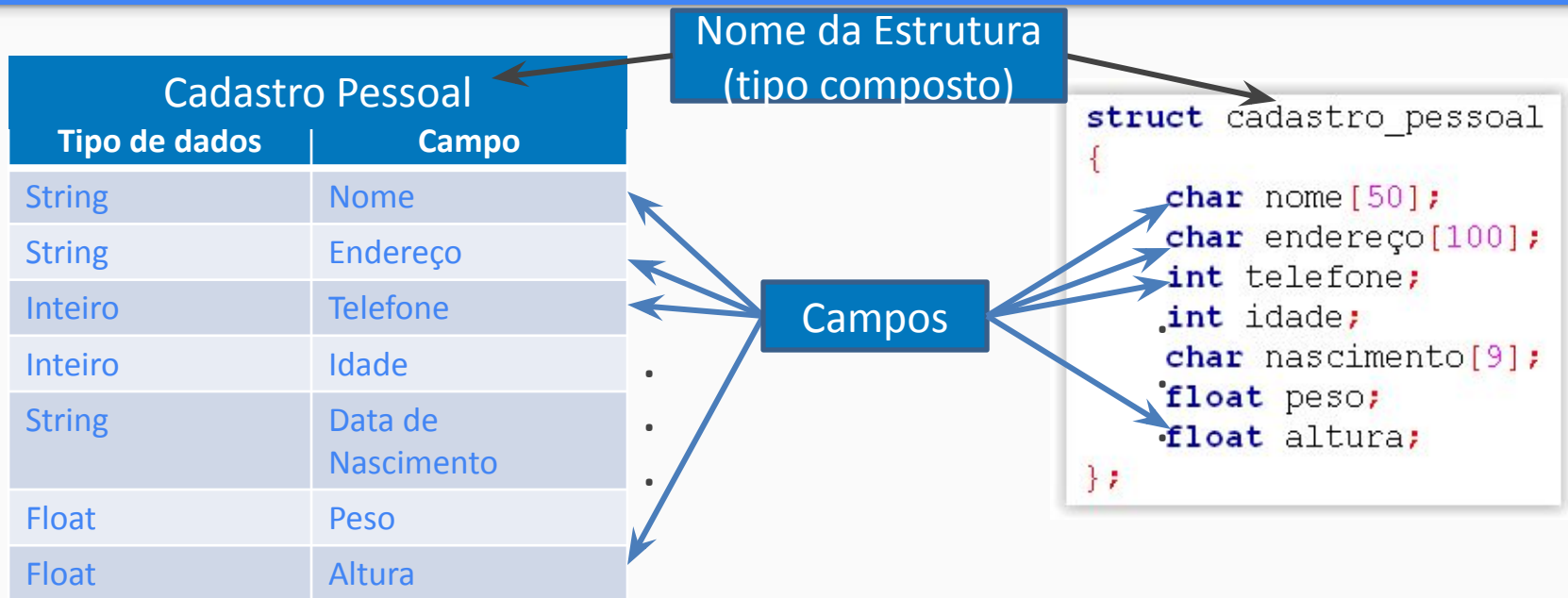
Nome da Estrutura (tipo composto)	Cadastro Pessoal		Campos ou Membros
	Tipo de dados	Campo	
	String	Nome	
	String	Endereço	
	Inteiro	Telefone	
	Inteiro	Idade	
	String	Data de Nascimento	
	Float	Peso	
	Float	Altura	

Definindo Estruturas de Dados

- Forma geral:

```
struct nome_do_tipo {  
    tipo campo_1;  
    tipo campo_2;  
    tipo campo_3;  
    ...  
};
```

Definindo Estruturas de Dados



Importância de Estruturas de Dados

- Considere um ponto representado por duas coordenadas: x e y
- Sem estrutura de dados:

```
int main()  
{  
    float x;  
    float y;  
    ...  
}
```

- Não deixa claro que estas variáveis estão conectadas

Importância de Estruturas de Dados

- Serve para agrupar diversas variáveis dentro de um único contexto:

```
struct Ponto2D {  
    float x ;  
    float y ;  
};
```

- Estrutura *Ponto2D* é um tipo.
- Declarar uma variável deste tipo da seguinte forma:

```
int main() {  
    struct Ponto2D p1;  
    ...  
}
```

Acessando Membros do Tipo Ponto

- Operador de acesso ("."):

```
struct Ponto2D {  
    float x ;  
    float y ;  
};  
  
int main() {  
    struct Ponto2D p1;  
    p1.x = 0.0;  
    p1.y = 7.5;  
    ...  
}
```

O nome da variável do tipo struct deve vir antes do "."

Após o "." vem o nome do campo que será acessado.

Forma geral: nome_variavel.nome_do_campo

Exemplo:

```
/* programa que captura e imprime coordenadas*/
#include <stdio.h>

struct Ponto2D
{
    float x ;
    float y ;
} ;

int main ()
{
    struct Ponto2D p ;
    printf("\nDigite as coordenadas do ponto (x,y): ") ;
    scanf ("%f %f", &p.x , &p.y) ;
    printf("O ponto fornecido foi:(%f,%f)\n",p.x, p.y);
    return 0 ;
}
```


Onde Declarar um Tipo Estruturado?

- Fora das funções
 - Escopo da declaração engloba todas as funções no mesmo arquivo fonte
- Dentro de funções
 - Neste caso, escopo do tipo estruturado é na função
- Há outras formas de declarar estruturas e variáveis. Ex.:

```
struct Ponto2D
{
    float x ;
    float y ;
} p;
```

```
struct
{
    float x ;
    float y ;
} p;
```

Obs: Com estas formas, perde-se em legibilidade.

Inicializando Variáveis de Tipos Estruturados

- Estrutura com o auxílio do abre-fecha chaves (“{” e “}”):

```
struct pessoa
{
    char nome[60] ;
    int idade ;
};

int main()
{
    struct pessoa p = {"Ana", 30};
    ...
}
```

**Deve-se inicializar
os membros na
ordem correta!**

Atribuição de Estruturas

- Pode ser atribuída a outra variável deste mesmo tipo:

```
struct pessoa
{
    char nome[60] ;
    int idade ;
};
int main()
{
    struct pessoa p1, p2 = {"Ana", 30};
    p1 = p2;
    ...
}
```

Atribuição da
estrutura contida
em p2 para p1

Atribuição de Estruturas

- A inicialização de uma estrutura deve ser feita no ato de sua declaração:

```
struct pessoa
{
    char nome[60] ;
    int idade ;
};

int main()
{
    struct pessoa p1;
    p1 = {"Ana", 30};
    ...
}
```

Errado!

Outras Operações com Estruturas

- Como escrever um programa que imprime a soma das coordenadas de dois pontos?

```
struct ponto
{
    float x ;
    float y ;
};

int main()
{
    struct ponto p1 = {0.0, 4.5};
    struct ponto p2 = {1.0, 2.5};
    struct ponto p3;
    p3 = p1 + p2;
    printf("O x e y do novo ponto é:%f,%f",p3.x,p3.y);
    return 0;
}
```

Errado!
Não podemos somar
estruturas inteiras

Outras Operações com Estruturas

- Como escrever um programa que imprime a soma das coordenadas de dois pontos?

```
struct ponto
{
    float x ;
    float y ;
};

int main()
{
    struct ponto p1 = {0.0, 4.5};
    struct ponto p2 = {1.0, 2.5};
    struct ponto p3;
    p3.x = p1.x + p2.x;
    p3.y = p1.y + p2.y;
    printf("O x e y do novo ponto é:%f,%f", p3.x, p3.y);
    return 0;
}
```

Certo!
Temos que atuar
membro a membro

Usando typedef

- Permite criar novos nomes para tipos existentes
- Útil para abreviar nomes de tipos ou tipos complexos
- Forma Geral:

```
typedef tipo_existente novo_nome;
```

- Ex.:

```
typedef unsigned int uint;
```

- Define a palavra “uint” como sendo um novo tipo, novo nome do tipo “unsigned int”;

Usando typedef

- Muito útil para evitar a necessidade de utilizar a palavra “struct” nas declarações de variáveis tipo estrutura:

```
struct pessoa
{
    char nome[60] ;
    int idade ;
};

typedef struct pessoa Pessoa;

int main()
{
    Pessoa p = {"Ana", 30};
    ...
}
```

ou

```
typedef struct
{
    char nome[60] ;
    int idade ;
} Pessoa;

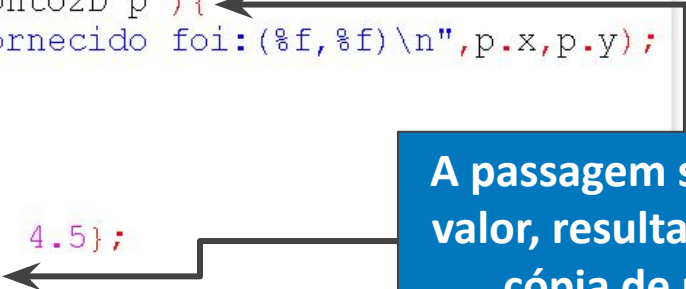
int main()
{
    Pessoa p = {"Ana", 30};
    ...
}
```


Passagem de Estruturas para Funções

```
typedef struct
{
    float x;
    float y;
} Ponto2D;

void imprimePonto ( Ponto2D p ){
    printf("O ponto fornecido foi: (%f,%f)\n",p.x,p.y);
}

int main()
{
    Ponto2D p1 = {0.0, 4.5};
    imprimePonto(p1);
    return 0;
}
```



A passagem será feita por valor, resultando em uma cópia de p1 para p

Retornando Estruturas

- Uma função pode retornar uma estrutura:

```
typedef struct
{
    float x;
    float y;
} Ponto2D;

Ponto2D somaPontos(Ponto2D p1, Ponto2D p2) {
    Ponto2D s;
    s.x = p1.x + p2.x;
    s.y = p1.y + p2.y;
    return s;
}

int main() {
    Ponto2D p1, p2, p3;
    printf("Digite as coordenadas do ponto1 (x,y): ");
    scanf ("%f %f", &p1.x , &p1.y ) ;
    printf("Digite as coordenadas do ponto2 (x,y): ");
    scanf ("%f %f", &p2.x , &p2.y ) ;
    p3 = somaPontos(p1,p2);
    printf("p1 + p2 = (%f,%f)\n",p3.x,p3.y);
    return 0;
}
```

O retorno será feito
por valor, resultando
em uma cópia de s

p1 p2

Estruturas Aninhadas

- Membros de uma estrutura podem ser outras estruturas previamente definidas Ex.:


```
typedef struct
{
    float x ;
    float y ;
} Ponto2D;

typedef struct
{
    Ponto2D centro;
    float raio;
} Circulo;
```

```
int main() {
    Circulo c;
    c.centro.x = 0;
    c.centro.y = 0;
    c.raio = 2;
}
```

Exemplo: Ponto está dentro do Círculo?

```
float distancia(Ponto2D p, Ponto2D q)
{
    float d = sqrt((q.x - p.x) * (q.x - p.x) +
                   (q.y - p.y) * (q.y - p.y));
    return d ;
}
```


$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
int interior(Circulo c, Ponto2D p)
{
    float d = distancia(c.centro, p);
    return (d <= c.raio);
}
```

Um ponto está no interior de um círculo se sua distância para o centro é menor que o raio do círculo

Vetores de Estruturas

- Considere o cálculo do centro geométrico de um conjunto de pontos:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} \text{ e } \bar{y} = \frac{\sum_{i=1}^n y_i}{n}$$

```
int main() {  
    Ponto2D vetorP2D[3];  
    vetorP2D[0].x = 0.5;  
    vetorP2D[0].y = 1.5;  
    vetorP2D[1].x = 3.0;  
    vetorP2D[1].y = 2.5;  
    vetorP2D[2].x = 2.0;  
    vetorP2D[2].y = 3.5;  
  
    Ponto2D centro;  
    centro = centroegeometrico(vetorP2D, 3);  
}
```

```
Ponto2D centroegeometrico(Ponto2D vet[], int n)  
{  
    int i ;  
    Ponto2D centro = {0.0 , 0.0} ;  
    for ( i = 0 ; i < n ; i++ )  
    {  
        centro.x += vet[i].x ;  
        centro.y += vet[i].y ;  
    }  
    centro.x /= n ;  
    centro.y /= n ;  
    return centro ;  
}
```

Atividade

- Faça um programa que declare uma estrutura “Funcionario” com os campos, nome, cargo, endereço, CPF, idade e salário.
- O programa deve ler do teclado as informações de 5 funcionários em um **vetor de estruturas**;
- Em seguida o programa imprime o nome e o salário de cada funcionário;
- Por fim, o programa imprime o total pago para todos os funcionários. Ex.:

```
Maria.....R$ 2.500,00
João.....R$ 2.000,00
Pedro.....R$ 1.500,00
-----
Total.....R$ 6.000,00
```

Atividade

- Adicione uma função ao programa “salarioCargo” que recebe como parâmetro o vetor de funcionários e uma string contendo o nome de um cargo, e retorna o total pago para todos os funcionários daquele cargo.
- Altere a função main para que, após imprimir o salário dos funcionários, solicite ao usuário o nome de um cargo e imprima o valor total pago para todos os funcionários do cargo informado.

Strings

Recapitulando: Vetores

- Representar uma coleção de variáveis de um mesmo tipo em uma dimensão

- Ex: `float notas[5];`

ou

`float notas[5] = {2.5,3.2,1.9,4.1,2.0};`

2.5	3.2	1.9	4.1	2.0
<code>notas[0]</code>	<code>notas[1]</code>	<code>notas[2]</code>	<code>notas[3]</code>	<code>notas[4]</code>

Recapitulando: Matrizes

- Representar uma coleção de variáveis de um mesmo tipo em duas dimensões

- Ex: `float mat[2][3];`

ou

`float mat[2][3] = {{2.5,3.2,1.9},{4.1,2.0,5.4}};`

`mat[0,0]` `mat[0,1]` `mat[0,2]`

2.5	3.2	1.9
4.1	2.0	5.4

`mat[1,0]` `mat[1,1]` `mat[1,2]`

Manipular Vetores de Caracteres (Strings)

- Caracteres em C

- Entrada/Saída de caracteres
- Funções que manipulam caracteres

- Vetores de caracteres (Strings)

- Inicialização
- Entrada/Saída de Strings
- Funções de Manipulação de Strings

Caracteres: Tipo char

- Usado para representar caracteres
- Armazena valores inteiros (em 1 byte)
- Um *literal char* é escrito entre aspas simples:

```
#include <stdio.h>

int main()
{
    char letraA = 'A';
    char letraC;
    letraC = 'C';
    printf ( " %c %c ", letraA , letraC);
    return 0;
}
```

Caracteres: Representação Interna

- Representados internamente na memória do computador por códigos numéricos

```
#include <stdio.h>

int main() {
    char letraA = 65; //Código da letra A
    char letraC;
    letraC = 67; //Código da letra C
    printf ( "%c %c ", letraA , letraC) ;
}
```

- Tabela ASCII: mapeamento entre caractere e código numérico
- Na tabela ASCII:
 - os dígitos são codificados em sequência
 - as letras minúsculas e maiúsculas também

Tabela ASCII do 30 ao 126

	0	1	2	3	4	5	6	7	8	9
30			sp	!	“	#	\$	%	&	‘
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Caracteres de controle

- 000 – ‘\0’ (fim de string)
- 009 – ‘\t’ (tabulação)
- 010 – ‘\n’ (fim de linha)
- 013 – ‘\r’ (retorno de linha)

Observe que as letras maiúsculas (65-90), minúsculas (97-122) e os dígitos (48-57) estão dispostos em sequência!

Codificação Sequencial

- Ajuda a identificar o tipo de caractere
- A função abaixo verifica se um dado caractere é um dígito entre ' 0 ' e ' 9 ':

```
int digito (char c) {  
    int ehDigito;  
    if(( c >= '0') && (c <= '9')) {  
        ehDigito = 1;  
    }  
    else {  
        ehDigito = 0;  
    }  
    return ehDigito;  
}
```

Conversão para Maiúscula

Testa se é minúscula

```
char maiuscula(char c)
{
    char maiusc = c;
    if ((c >= 'a') && (c <= 'z'))
    {
        maiusc = c - ('a' - 'A');
    }
    return maiusc ;
}
```

A diferença entre qualquer caractere minúsculo e seu respectivo maiúsculo é a mesma da letra 'a' minúscula para a letra 'A' maiúscula.

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Impressão de Caracteres

- Duas formas diferentes usando o `printf` (`%d`: como valor ASCII, `%c`: como caractere):

```
char lc = 97 ;  
printf("%d %c",lc,lc);
```

Saída:

97 a

```
char la = 'a' ;  
printf("%d %c",la,la );
```

Saída:

97 a

- Função `putchar` (`stdio.h`) permite a impressão de um caractere:

```
char la = 'a'; //ou: la = 97;  
putchar(la);
```

Saída:

a

Leitura de Caracteres

- Função `scanf`

```
char a ;  
scanf("%c", &a);
```

ou

```
char a ;  
scanf(" %c", &a);
```

Este espaço diz ao `scanf` para ignorar espaços, tabulações, ou outros caracteres de controle que estejam no buffer do teclado

- Função `getchar` (`stdio.h`) permite a leitura de um caractere

```
char a ;  
a = getchar();
```

Leitura de Caracteres

- As funções `scanf` e `getchar` obrigam que a tecla *enter* `'\n'` seja pressionada após a entrada dos dados.
- Existem funções para ler dados sem esperar pelo *enter* em C **para ambientes Windows:**

- Função `getche` – definida em `conio.h`:

```
char letra;  
letra = getche();
```

Lê um caractere e o **exibe** na tela

- Função `getch` – definida em `conio.h`:

```
char letra;  
letra = getch();
```

Lê um caractere e não **exibe** na tela

Strings: vetores de caracteres

- Vetor do tipo **char**

R	i	o	\0
---	---	---	----
- Terminadas pelo caractere nulo: '**\0**';
- Para imprimir printf: %s
- Muitas funções que manipulam strings o fazem caractere a caractere, a partir do endereço do primeiro até que '**\0**' seja encontrado.

```
int main() {  
    char cidade[4];  
    cidade[0]='R';  
    cidade[1]='I';  
    cidade[2]='O';  
    cidade[3]='\0';  
    printf("%s", cidade);  
}
```

O identificador "cidade"
(sem colchetes)
referencia o endereço do
primeiro caractere da
string.


Inicialização de Strings

- Na declaração:

```
int main() {  
    char cidade[] = {'R', 'I', 'O', '\0'} ;  
    printf ("%s\n", cidade ) ;  
}
```

```
int main() {  
    char cidade[] = "RIO";  
    printf ("%s\n", cidade);  
}
```

O caractere nulo é representado implicitamente e o vetor é declarado com tamanho 4



- Através da escrita dos caracteres entre aspas duplas:

Declaração de Strings

- Inicialização do vetor de caracteres na declaração:

```
char s2[] = "Rio de Janeiro";
```

```
char s3[81];
```

```
char s4[81] = "Rio";
```

Representa um vetor com
15 elementos

Representa um vetor de no
máximo, 80 caracteres válidos

Representa um vetor de no máximo
80 caracteres válidos, mas com um
valor já inicializado

Literais do tipo String

- Devem ser declarados entre aspas duplas.

- Ex:

```
printf("Um literal string!\n");  
printf("Eu moro em %s ", "Recife");
```

- Para cada literal presente no código, é criada e inicializada uma região de memória que cabe todos os caracteres e o `\0`:

Endereço:	100	101	102	103	104	105	106
Dados:	R	e	c	i	f	e	\0

Literais do tipo String

- A atribuição de literais para strings não é permitida:

```
int main(){  
    char cidade[4];  
    cidade = "Rio";  
}
```



- A não ser na declaração:

```
int main(){  
    char cidade[] = "RIO";  
    printf("%s\n", cidade);  
}
```



- Ou pode ser utilizada a função `strcpy` em `string.h`

```
int main(){  
    char cidade[4];  
    strcpy(cidade, "Rio");  
}
```



Leitura de Strings

```
char cidade [81];  
scanf ("%s", cidade );
```

- Pode ser utilizada a função `scanf` com o especificador `%s`.
- `scanf` precisa do endereço de memória onde os dados lidos serão armazenados (identificador `cidade`, já se refere ao endereço de memória do primeiro elemento do vetor)
- `"%s"` para a captura com qualquer caractere de controle (espaço, tabulação, etc)

Buffer Overflow

- Quando o usuário digitar mais caracteres do que cabem na string
- Solução: função `scanf` com outros especificadores:

```
int main(){  
    char cidade[10];  
    scanf("%[^\\n]s", cidade);  
    printf("%s", cidade);  
    return 0;  
}
```

- `[%^\\n]` diz para a função `scanf` parar apenas quando encontrar um caractere `\\n` (enter).
- Podem ser utilizados outros caracteres: `[%^abc\\n]` diz para `scanf` parar apenas ao encontrar um dos caracteres: `'a'`, `'b'`, `'c'`, ou `\\n`.

Leitura de Strings

- Para evitar *buffer overflow*, pode-se utilizar um número após % para indicar o máximo de caracteres a serem lidos:

```
int main(){
    char cidade[10];
    scanf("%9[^\n]s", cidade);
    printf("%s", cidade);
    return 0;
}
```

Lê no máximo 9 caracteres digitados, pois deve deixar espaço para o `'\0'`.

- `%9[^\n]` diz para a função `scanf` parar apenas quando encontrar um caractere `'\n'` (enter) ou quando o limite de 9 caracteres for atingido. Adiciona o `'\0'` automaticamente ao final da string.
- Outra opção é utilizar a função `fgets`:

```
int main(){
    char cidade[10];
    fgets(cidade, 9, stdin);
    printf("%s", cidade);
    return 0;
}
```

Diferentemente do `scanf`, a função `fgets` inclui o caractere `'\n'` (enter) digitado pelo usuário ao final da string.

Leitura de Strings: fflush

- Caracteres digitados pelo usuário e não lidos ficam no buffer do teclado e serão entregues para a próxima chamada de leitura.
- Utilize `fflush(stdin)` para descartar o que ficou no buffer do teclado antes de fazer a próxima leitura. Para garantir que os novos dados digitados pelo usuário é que serão lidos:

```
int main()
{
    char cidade1[10], cidade2[10];

    printf("Cidade1: ");
    scanf("%9[^\n]", cidade1);
    printf("%s\n", cidade1);
    fflush(stdin);
    printf("Cidade1: ");
    scanf("%9[^\n]", cidade2);
    printf("%s\n", cidade2);
    return 0;
}
```

Descarta o que não foi lido pelo primeiro `scanf`

Impressão de Strings

- Exemplo de uma função que imprime uma string, caractere a caractere até encontrar o '\0':

```
void imprime(char s[]) {  
    int i;  
    for (i = 0; s[i] != '\0'; i++){  
        printf("%c", s[i]) ;  
    }  
    printf("\n");  
}
```

- Função análoga a:

```
printf("%s\n", s)
```

Copiando Strings

- Copiar uma string para outra: origem -> destino

```
void copia (char dest[], char orig[] )  
{  
    int i;  
    for (i = 0; orig[i]!='\0'; i++)  
    {  
        dest[i] = orig[i];  
    }  
    /* fecha a cadeia copiada */  
    dest[i] = '\0';  
}
```

- Função `strcpy` (definida em `string.h`):

```
strcpy(char* dest, char* orig);
```

```
#include <stdio.h>  
#include <string.h>  
  
int main(){  
    char s1[] = "ABC";  
    char s2[4];  
    strcpy(s2,s1);  
    printf(":%s:\n",s1);  
    return 0;  
}
```

Comparação de Strings (string.h)

```
strcmp(char *str1, char *str2);
```

- Função strcmp: comparar strings (não é permitido comparar strings com os operadores '<', '>', '==', '<=', e '>=')

- ```
char nome1[20]="Joao da Silva", nome2[20]="Jose Santos";
if (strcmp(nome1,nome2)!=0)
 printf("Os nomes são diferentes!");
```

- Retorna um inteiro positivo se **str1** é lexicamente posterior a **str2**; zero se as duas são idênticas; e negativo se **str1** é lexicamente anterior que **str2**; Ex.:

- `strncat`: concatena `n` caracteres de origem para destino:

```
strncat(char *dest, char *origem, int n)
```

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

int main(){
 char s1[20] = "Rio ";
 char s2[20] = "de Janeiro";
 strncat(s1, s2, 10);
 printf("%s\n", s1);
 return 0;
}
```

- `strncpy`: copia `n` caracteres da origem no destino:

```
strncpy(char *dest, char *origem, int n)
```

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

int main(){
 char s1[20] = "Rio de Janeiro";
 char s2[10];
 strncpy(s2, s1, 3);
 printf("%s\n", s2);
 return 0;
}
```

Certifique-se de que a string de destino possui espaço suficiente para caber o que está sendo colocado nela



# Vetor de Strings

```
1 /* Exemplos dos slides */
2
3 #include <stdio.h>
4 #define MAX 50
5
6 int main ()
7 {
8 int i , numAlunos ;
9 char alunos[MAX][121] ;
10 do {
11 printf("Digite o numero de alunos: ") ;
12 scanf ("%d",&numAlunos);
13 for (i = 0; i < numAlunos; i++)
14 scanf("%120[^\n]s", alunos[i]) ; // Lê uma string na linha i
15 }
16 while (numAlunos > MAX);
17 return 0 ;
18 }
```

strings

Cada linha da  
matriz guarda  
uma string

# Atividade 1

- Faça um programa que solicita ao usuário digitar o nome e endereço completo (armazenando em duas strings). Em seguida o programa imprime na tela o que foi digitado.

# Atividade 2

- Faça um programa que solicita ao usuário digitar o nome e sobrenome.
- Em seguida o programa solicita ao usuário digitar rua, número, bairro, cidade (capturando todos os dados como string).
- Finalmente o programa concatena o nome e sobrenome e mostra na tela.
- Depois o programa concatena os dados do endereço e imprime o endereço de uma só vez.

# Atividade 3

- Faça um programa que solicita o usuário digitar uma mensagem (string). Em seguida o programa converte todos os caracteres da string para maiúsculo e depois imprime os resultados.

# Atividade 4

- Faça um programa que solicita o usuário digitar o nome de 5 pessoas, leia estes nomes em um vetor de strings e, em seguida, imprima o primeiro e o último deles na ordem alfabética.

# Ponteiros

# Variáveis e Endereços

- Memória: sequência contínua de posições endereçáveis
- Cada posição armazena informação: desde a posição zero até a posição máxima de memória.



# Variáveis e Endereços

- Variáveis: associadas a endereços de memória

```
int x = 2;
char str1[] = "abcd";
char str2[] = "olá!";
```

Obs.: inteiros ocupam 4 bytes enquanto cada caractere ocupa 1 byte

| 0 | ... | 12 | 13 | 14 | 15 | 16  | 17  | 18  | 19  | 20   | 21  | 22  | 23  | 24  | 25   | ... |
|---|-----|----|----|----|----|-----|-----|-----|-----|------|-----|-----|-----|-----|------|-----|
| 2 | 1   | 2  |    |    |    | 'a' | 'b' | 'c' | 'd' | '\0' | 'o' | 'l' | 'a' | '!' | '\0' | ... |

x

str1

str2

| ... | 31999994 | 31999995 | 31999996 | 31999997 | 31999998 | 31999999 |
|-----|----------|----------|----------|----------|----------|----------|
| ... | '#'      | '%'      | 1234     | 321      | 9985345  | 123984   |

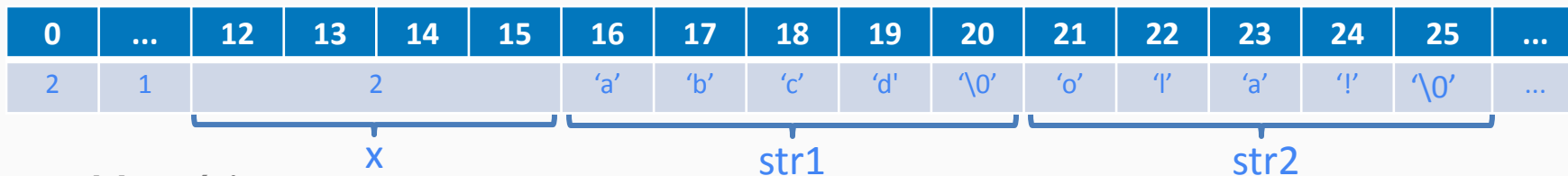
Lixo de memória que sobrou de execuções de programas anteriores



# Variáveis e Endereços

```
int x = 2;
char str1[] = "abcd";
char str2[] = "olá!";
```

- Conteúdo de cada variável está armazenado em um endereço de memória e possui um tamanho



- Memória:

| Variável | Endereço     | Tamanho      | Conteúdo     |
|----------|--------------|--------------|--------------|
| x        | <div>?</div> | <div>?</div> | <div>?</div> |
| str1     | <div>?</div> | <div>?</div> | <div>?</div> |
| str2     | <div>?</div> | <div>?</div> | <div>?</div> |

# Endereço, Tamanho, e Conteúdo

- A notação utilizada para se obter:
  - Conteúdo de uma variável: `x` (ou `x[i]` para vetores)
  - Endereço: `&x` (ou `x` para vetores)
  - Tamanho: `sizeof(x)` (ou `sizeof(tipo)` para saber o tamanho de um tipo)

Obs.: para vetores, o nome da variável já indica também seu endereço inicial.

```
#include <stdio.h>
#include <string.h>
int main()
{
 int x = 2;
 char str1[] = "abcd";
 char str2[] = "ola!";

 printf("x:\n Endereço: %d\n Tamanho: %d\n Conteúdo: %d\n\n",
 &x, sizeof(x), x);
 printf("str1:\n Endereço: %d\n Tamanho: %d\n Conteúdo: %s\n\n",
 str1, sizeof(str1), str1);
 printf("str2:\n Endereço: %d\n Tamanho: %d\n Conteúdo: %s\n\n",
 str2, sizeof(str2), str2);
 printf("str2:\n Endereço: %d\n Tamanho: %d\n Conteúdo: %s\n\n",
 &str2[0], sizeof(char)*(strlen(str2)+1), str2);
 return 0;
}
```

```
x:
Endereco: 2293580
Tamanho: 4
Conteúdo: 2

str1:
Endereco: 2293575
Tamanho: 5
Conteúdo: abcd

str2:
Endereco: 2293570
Tamanho: 5
Conteúdo: ola!

str2:
Endereco: 2293570
Tamanho: 5
Conteúdo: ola!
```

# Declarando Variáveis do Tipo Ponteiro em C

- Para declarar: notação '\*':
- `int x:`
  - Declara uma variável que armazena um inteiro;
- 
- `int *y:`
  - Declara uma variável que armazena o endereço de memória de um inteiro. Ou, como se diz: guarda um ponteiro para uma variável do tipo inteiro.
- 
- Forma geral:
- `tipo *nomeDoPonteiro;`

# Operador &

- Operador unário que fornece o endereço de uma variável:
- Forma geral:

```
int x;
int *y;
&nomeDaVariavel
y = &x;
```

Faz o ponteiro y  
apontar para x

Obs.: Esse operador não pode ser utilizado com literais:

`y = &3;` //Errado!

# Ponteiros

```
int main ()
{
 int x = 2;
 int *y = &x;

 printf("x:\n Endereco: %d\n Tamanho: %d\n Conteudo: %d\n\n",
 &x, sizeof(x), x);
 printf("y:\n Endereco: %d\n Tamanho: %d\n Conteudo: %d\n Conteudo apontado: %d\n\n",
 &y, sizeof(y), y, *y);
 return 0;
}
```

x:

Endereco: 16

Tamanho: 4

Conteudo: 2

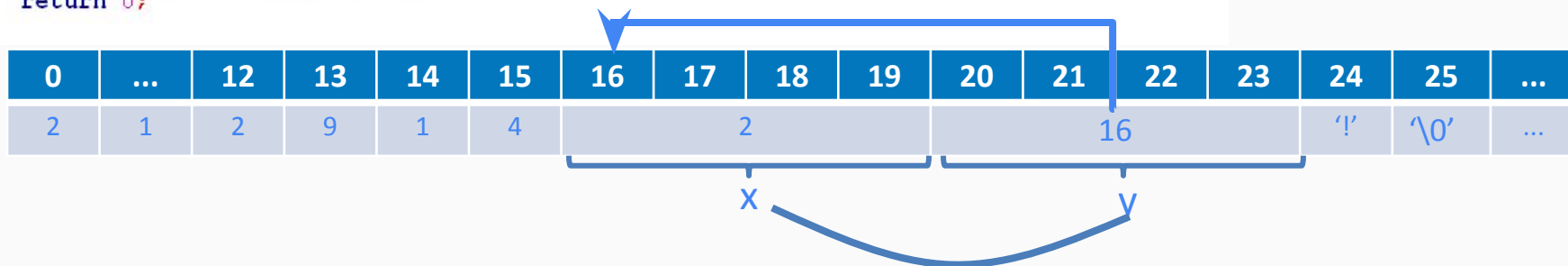
y:

Endereco: 20

Tamanho: 4

Conteudo: 16

Conteudo apontado: 2



- Dizemos que y aponta para x

## Utilizando Ponteiros

```
int i, j;
```

```
int *ip;
```

```
i = 12;
```

```
ip = &i;
```

```
j = *ip;
```

```
*ip = 21;
```

A variável **ip** armazena um ponteiro para um inteiro

O endereço de **i** é armazenado em **ip**

O conteúdo da posição apontada por **ip** é armazenado em **j**

O conteúdo da posição apontada por **ip** passa a ser **21**

# Utilizando Ponteiros

1)

```
int i,j;
```

```
int *ip;
```

|    |   |     |
|----|---|-----|
|    |   |     |
|    |   |     |
| ip | - | 112 |
| j  | - | 108 |
| i  | - | 104 |

2)

```
i = 12;
```

|    |    |     |
|----|----|-----|
|    |    |     |
|    |    |     |
| ip | -  | 112 |
| j  | -  | 108 |
| i  | 12 | 104 |

3)

```
ip = &i;
```

|    |     |     |
|----|-----|-----|
|    |     |     |
|    |     |     |
| ip | 104 | 112 |
| j  | -   | 108 |
| i  | 12  | 104 |

4)

```
j = *ip;
```

```
*ip = 21;
```

|    |     |     |
|----|-----|-----|
|    |     |     |
|    |     |     |
| ip | 104 | 112 |
| j  | 12  | 108 |
| i  | 21  | 104 |

# Ponteiros como Parâmetro de Função

- Como uma função pode alterar variáveis de quem a chamou?
  - 1) função chamadora passa os endereços dos valores que devem ser modificados ;
  - 2) função chamada deve declarar os endereços recebidos como ponteiros;

```
#include <stdio.h>

void somaprod(int a, int b, int* p, int* q)
{
 *p = a + b ;
 *q = a * b ;
}

int main ()
{
 int s, p ;
 somaprod (3, 5, &s, &p) ;
 printf("Soma= %d e Produto = %d \n", s, p) ;
 return 0 ;
}
```

Passagem de  
parâmetros por  
referência



# Para que Ponteiros são Usados?

- Possibilitar que funções modifiquem os argumentos que recebem;
- Manipular vetores e strings - Vetores são passados como parâmetro através de um ponteiro para o primeiro elemento;
- Criar estruturas de dados mais complexas, como listas encadeadas, árvores binárias etc (não abordado no curso);
- Reduz a necessidade de variáveis globais, melhorando a modularidade;
- Podem ser utilizados para produzir código com melhor desempenho evitando cópias de dados de uma variável para outra.

# Operações com Ponteiros

```
#include <stdio.h>

int main()
{
 int x=5, y=6;
 int *px, *py;
 px = &x;
 py = &y;
 if (px < py)
 printf("py-px = %d\n", py-px);
 else
 printf("px-py = %d\n", px-py);
 return 0;
}
```

Se px e py apontam para 65488  
e 65484, respectivamente...

A saída será:  
**px - py = 1**  
(um inteiro de diferença)

- Testes relacionais  $\geq$ ,  $\leq$ ,  $<$ ,  $>$  e  $==$  são aceitos em ponteiros;
- A diferença entre dois ponteiros será dada na unidade do tipo de dado apontado. Ex.: A diferença entre dois endereços de inteiros será dada em inteiros;

# Operações com Ponteiros: Incremento

```
int main()
{
 int x=5, y=6;
 int *px, *py;
 px = &x;
 py = &y;
 printf("px = %u\n", px);
 printf("py = %u\n", py);
 py++;
 printf("py = %u\n", py);
 py = px+3;
 printf("py = %u\n", py);
}
```

**Podemos utilizar operador de incremento com ponteiros**

**Podemos fazer aritmética de ponteiros**

- Todas as operações são realizadas levando-se em consideração o tamanho do tipo apontado pelo ponteiro (No exemplo: int, em geral 4 bytes).

# Operações com Ponteiros

- O incremento de um ponteiro acarreta na movimentação do mesmo para o próximo valor do **tipo apontado**
  - Ex: Se `px` é um ponteiro para `int` com valor `3000`, depois de executada a instrução `px++`, o valor de `px` será `3004` e não `3001` !!!
- Deslocamento varia de compilador para compilador dependendo do número de bytes adotado para o referido tipo
- Ponteiros podem não apontar para lugar nenhum. Isto ocorre quando o ponteiro aponta para `0` (zero), ou `NULL` (significa nulo);
- `NULL` é uma constante tipo macro definida em algumas bibliotecas da seguinte maneira:  
`#define NULL 0`

# Operações com Ponteiros

- Na aritmética entre ponteiros é valido:

- somar ou subtrair um inteiro a um ponteiro ( $pi \pm int$ )
- incrementar ou decrementar ponteiros ( $pi++$ ,  $pi--$ )
- subtrair ponteiros (produz um inteiro) ( $pf - pi$ )
- comparar ponteiros ( $>$ ,  $>=$ ,  $<$ ,  $<=$ ,  $==$ )

- Não é válido:

- somar ponteiros ( $pi + pf$ )
- multiplicar ou dividir ponteiros ( $pi * pf$ ,  $pi / pf$ )
- operar ponteiros com double ou float ( $pi \pm 2.0$ )

- É possível percorrer vetores utilizando ponteiros:

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

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

 for (pvet = &vet[0]; pvet<=&vet[4]; pvet++)
 printf("%d\n", *pvet);
}
```



c:\Z:\UFPE\Grad  
5  
4  
3  
2  
1  
5  
4  
3  
2  
1

Equivalente a:  
**pvet = vet;**

Aponta para a próxima  
posição do vetor

Referencia o conteúdo do vetor  
na posição atualmente  
apontada

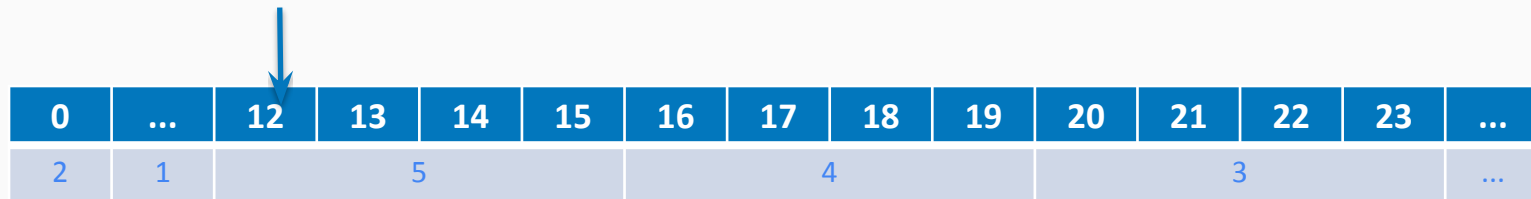
# Vetores e Ponteiros

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

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

 for (pvet = &vet[0]; pvet<=&vet[4]; pvet++)
 printf("%d\n", *pvet);
}
```

pvet



|   |     |    |    |    |    |    |    |    |    |    |    |    |    |     |
|---|-----|----|----|----|----|----|----|----|----|----|----|----|----|-----|
| 0 | ... | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | ... |
| 2 | 1   | 5  |    |    |    |    | 4  |    |    |    | 3  |    |    | ... |

# Vetores e Ponteiros

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

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

 for (pvet = &vet[0]; pvet<=&vet[4]; pvet++)
 printf("%d\n", *pvet);
}
```

pvet



| 0 | ... | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21  | 22 | 23 | ... |
|---|-----|----|----|----|----|----|----|----|----|----|-----|----|----|-----|
| 2 | 1   | 5  |    |    | 4  |    |    | 3  |    |    | ... |    |    |     |



# Vetores e Ponteiros

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

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

 for (pvet = &vet[0]; pvet<=&vet[4]; pvet++)
 printf("%d\n", *pvet);
}
```

pvet

↓

|   |     |    |    |    |    |    |    |    |    |    |    |    |    |     |
|---|-----|----|----|----|----|----|----|----|----|----|----|----|----|-----|
| 0 | ... | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | ... |
| 2 | 1   | 5  |    |    |    | 4  |    |    |    | 3  |    |    |    | ... |

# Atividade 1

- Escreva uma função que imprime os elementos de um vetor de float a partir de um endereço inicial (ponteiro) até um endereço final;
- A função recebe como parâmetros dois ponteiros para float (os endereços inicial e final) e deve utilizar notação de ponteiros não de vetores;
- Deve ser criada uma função main() para testar a função implementada com três vetores de tamanhos e conteúdos diferentes;
- A função main deve imprimir duas partes distintas de cada um dos três vetores, utilizando a função criada.

```
#include <stdio.h>

void imprimir_vetor(float *primeiro, float *ultimo){
 for(;primeiro <= ultimo;primeiro++){
 printf("%f\n",*primeiro);
 }
}

int main(){
 float nums[5] = {1.0,2.0,3.0,4.0,5.0};
 imprimir_vetor(&nums[0],&nums[4]);
 return 0;
}
```

# Alocação Dinâmica

# Ponteiros e Vetores

- Considere a declaração de um vetor:

```
int v[10];
```

- O símbolo `v`:
  - É o nome do vetor (identificador)
  - É uma constante que representa seu endereço inicial
  - É um ponteiro constante apontando para o primeiro elemento do vetor

# Ponteiros e Vetores

- Em C existe um relacionamento muito forte entre ponteiros e vetores:
  - O identificador de um vetor representa um endereço, ou seja, é um ponteiro;
  - Qualquer operação que possa ser feita com índices de um vetor, também pode ser feita com ponteiros;
  - Isto ocorre pois, a maioria dos processadores é capaz de manipular diretamente ponteiros (endereço) e não vetores completos.
- Lembrando que:
  - Seja `v`, um ponteiro que aponta para o endereço da variável `a`:

```
int a = 10;
int *v;
v = &a;
```
  - A notação `*v`, refere-se ao conteúdo da variável `a`, podendo esta notação ser utilizada para ler ou atribuir novo conteúdo para `a`. Ex.:

```
int b = *v; // Atribui o valor contido em a (10) para b.

*v = 20; // Atribui 20 para a variável a
```

# Ponteiros e Vetores

- Como vimos, C permite aritmética de ponteiros.
- Se tivermos a declaração:

`int v[10];`

- Podemos acessar elementos do vetor através de aritmética de ponteiros:
  - `v` ou `(v + 0)`: Aponta para o primeiro elemento do vetor;
  - `v + 1`: Aponta para o segundo elemento do vetor;
  - `v + 9`: Aponta para o último elemento do vetor;
- Portanto, se equivalem:

`&v[i]`  $\leftrightarrow$  `(v + i)`  
`v[i]`  $\leftrightarrow$  `*(v + i)`

# Ponteiros e Vetores

- Vetores podem ser tratados como ponteiros em C!

```
int main () {

 int a[10];
 int *pa;
 pa = &a[0];
 pa = a;

 //Expressões equivalentes:
 *pa <-> a[0] <-> pa[0]
 *(pa+i) <-> a[i] <-> pa[i] <-> *(a+i)
 a+i <-> &a[i]
}
```

<->

Expressões equivalentes



# Ponteiros e Vetores

- Vetores podem ser tratados como ponteiros em C!
- Com notação de vetores:

```
int nums[] = {1, 4, 8};
int cont;
for(cont=0; cont < 3; cont++)
{
 printf("%d\n", nums[cont]);
}
```

- Com notação de ponteiros:

```
int nums[] = {1, 4, 8};
int cont;
for(cont=0; cont < 3; cont++)
{
 printf("%d\n", *(nums + cont))
}
```

# Ponteiros Constantes vs Ponteiros Variáveis

- Observe o seguinte código:

```
int nums[] = {1, 4, 8};
int cont;
for(cont=0; cont < 3; cont++)
{
 printf("%d\n", *(nums++));
}
```

Errado!

- Declaração de uma constante do tipo ponteiro para inteiros (ponteiro constante);
- A expressão “**nums++**” tenta incrementar o endereço constante **numse** atualizar a constante com novo endereço, o que é inválido.

# Ponteiros Constantes vs Ponteiros Variáveis

- Observe o seguinte código:

```
int nums[] = {1, 4, 8};
int* pnums = nums;
int cont;
for(cont=0; cont < 3; cont++)
{
 printf("%d\n", *(pnums++));
}
```

Correto!

- Declaração de uma variável do tipo ponteiro para inteiros
- (ponteiro variável);
- A expressão “**pnums++**” Incrementa endereço armazenado na variável **pnums** e atualiza a variável com novo endereço.

# Alocação de Memória

- Quando declaramos uma variável, o compilador reserva (aloca) um espaço na memória suficiente para armazenar valores do tipo da variável;
- Alocação estática (em tempo de compilação):

```
int main()
{
 int var ;
 char s1 [10];
 char* s2;
}
```

Aloca espaço para 1 int

Aloca espaço para 10 char

Aloca espaço para 1 endereço

- E quando queremos alocar espaço dependendo de entradas fornecidas pelo usuário (em tempo de execução)?

# Alocação Estática vs Dinâmica

- Modos de alocar espaço em memória:
- Alocação estática:
  - Variáveis globais (e estáticas): O espaço reservado para a variável existe enquanto o programa estiver sendo executado
  - Variáveis locais: O espaço existe enquanto a função, que declarou a variável, estiver sendo executada.
- Alocação dinâmica:
  - Requisitar memória em **tempo de execução**;
  - **O espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado pelo programa.**

# Alocação Dinâmica

- Função básica para alocar memória é **malloc** presente na biblioteca **stdlib.h**

```
void* malloc(unsigned nbytes);
```

- Recebe como argumento um número inteiro sem sinal que representa a quantidade de bytes que se deseja alocar;
- Retorna o endereço inicial da área de memória alocada
- A função **malloc** retorna um ponteiro genérico, para qualquer tipo, representado por **\*void**;
- Faz-se a conversão para o tipo apropriado usando o operador de molde de tipo (cast). Ex.: **(int \*)**, **(char \*)**, etc.

# Alocação Dinâmica

- Aloca somente a quantidade de memória necessária.
- Exemplo:

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

- Se a alocação for bem sucedida, `v` armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros (40 bytes, supondo inteiros de 4 bytes).
- Equivalente a declaração de um vetor de inteiros de 10 posições (`int v[10]`), porém, de forma dinâmica.

# Alocação Dinâmica

- Se não houver espaço livre suficiente para realizar a alocação, a função `malloc` retorna um endereço nulo, 0 (zero), `NULL`;
- É uma boa prática de programação testar se a alocação foi bem sucedida para evitar erros de execução:

```
int *v ;
v = (int *) malloc (10 * sizeof(int));
if (v==NULL) {
 printf("Erro: não foi possível alocar memória!");
 return -1;
}
```



# Liberando o Espaço Alocado

- Uso da função **free** para liberar espaço de memória alocada dinamicamente;

```
void free(void *)
```

- Recebe como parâmetro o ponteiro da memória a ser liberada
- O espaço de memória fica livre para ser alocado futuramente pelo próprio programa ou por outro programa;
- Recomenda-se liberar espaço de memória previamente alocado quando o mesmo não é mais necessário, evitando desperdício.

- Ex.:

```
//Libera a memória alocada em v
free(v);
```

## Exemplo do Ciclo Alocação-Liberação Completo

```
int main()
{
 //Tentar alocar memória
 int *v ;
 v = (int *) malloc (10 * sizeof(int));

 //Verifica se foi alocada
 if (v==NULL) {
 printf("Erro: não foi possível alocar memória!");
 return -1;
 }

 //Acesso ao vetor v
 //(...)

 //Libera a memória alocada em v
 free(v);
 return 0;
}
```

## Média com alocação dinâmica

```
int main()
{
 int qtdNumeros, contador = 0;
 float* numeros;
 float soma = 0.0;
 do {
 printf("Quantidade de numeros?:\n");
 scanf("%d", &qtdNumeros);
 }
 while (qtdNumeros <= 0);
 numeros = (float*) malloc(qtdNumeros*sizeof(float));
 if (numeros != NULL) {
 while (contador < qtdNumeros) {
 scanf("%f", &numeros[contador]);
 soma = soma + numeros[contador];
 contador++;
 }
 printf("media: %f", soma/contador);
 free(numeros);
 }
 return 0;
}
```

# Realocando espaço

- Podemos mudar o espaço de memória alocado previamente de forma dinâmica;
- Para isso, podemos utilizar a função **realloc**:

```
void* realloc(void* ptr, unsigned qtdBytes);
```

- Recebe endereço do bloco de memória alocado previamente e um número inteiro sem sinal que representa a quantidade de bytes que se deseja alocar;
- Retorna o endereço inicial da área de memória alocada ou **NULL** se não conseguir alocar memória;
- Copia a informação da memória antiga para a nova memória;

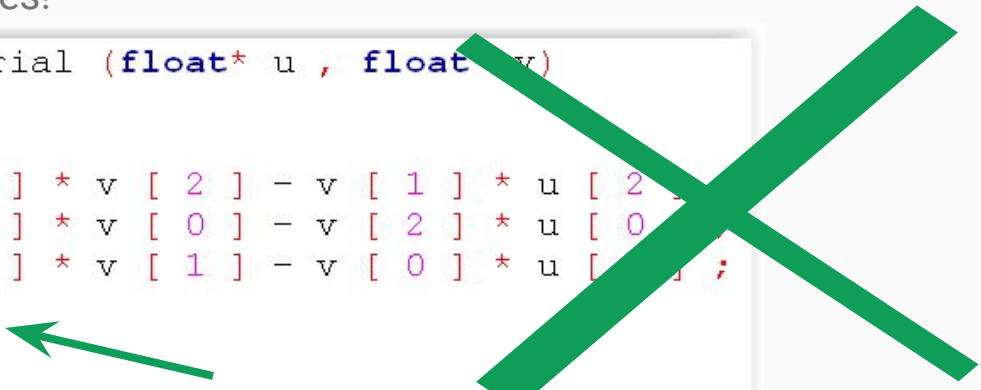
# Realocando espaço

```
int main() {
 int qtdNumeros = 5, contador = 0;
 char resposta;
 float media = 0.0;
 float *nums, *numsR;
 nums = (float*) malloc(qtdNumeros*sizeof(float));
 if (nums == NULL) {
 printf("Memoria insuficiente");
 exit(1);
 }
 printf("Programa calcula media de 5 numeros.");
 printf("Deseja mais/menos? (s/n)\n");
 scanf("%c",&resposta);
 if (resposta == 's') {
 printf("Quantidade de numeros?:\n");
 scanf("%d", &qtdNumeros);
 numsR = (float*) realloc(nums,qtdNumeros*sizeof(float));
 if (numsR != NULL){
 nums = numsR;
 }
 }
 //Calcula a média (...)
}
```

# Retornando Vetores

- Cuidado ao retornar vetores!

```
float* prod_vetorial (float* u , float* v)
{
 float p[3] ;
 p[0] = u [1] * v [2] - v [1] * u [2] ;
 p[1] = u [2] * v [0] - v [2] * u [0] ;
 p[2] = u [0] * v [1] - v [0] * u [1] ;
 return p ;
}
```



- Não se pode retornar vetores locais, pois a memória alocada de forma estática é desalocada ao final da função.

# Retornando Vetores

- Forma correta:

```
float* prod_vetorial (float* u , float* v)
{
 float* p = (float*) malloc(3 * sizeof(float)) ;
 p[0] = u [1] * v [2] - v [1] * u [2] ;
 p[1] = u [2] * v [0] - v [2] * u [0] ;
 p[2] = u [0] * v [1] - v [0] * u [1] ;
 return p ;
}
```

- Pergunta?

- Quem deve desalocar a memória alocada pela função?

- Resposta:

- A função que a chamou deve desalocar a memória.

# Atividade 1

- Faça um programa que calcula a média e o desvio padrão das notas de uma turma;
- O programa deve solicitar no início a quantidade de alunos na turma e alocar dinamicamente um vetor de float onde as notas dos alunos digitadas devem ser armazenadas;
- Você deve criar uma função “mediaDesvio” para calcular a média e o desvio padrão e retornar estes valores através de parâmetros passados por referência (ponteiros).
- A função recebe como parâmetro o ponteiro para o vetor, a quantidade de elementos no vetor, e dois ponteiros: media e desvio onde deve guardar os resultados;



# Atividade 2

- Escreva uma função que encontra um valor em um vetor de inteiros e retorna um ponteiro para o primeiro endereço onde este valor foi encontrado ou NULL caso o valor não esteja no vetor;
- A função recebe como parâmetro dois ponteiros para inteiros (os endereços inicial e final) e deve utilizar notação de ponteiros não de vetores;
- Deve ser criada uma função main() para testar a função implementada. Na função main deve ser declarado um vetor de tamanho 10, com três ocorrências do valor 2;
- A main() deve utilizar a função criada para encontrar e imprimir os endereços de memória de todas as ocorrências do valor 2, em um laço, até que todo o vetor tenha sido pesquisado.
- Defina a constante NULL se necessário.

# Bibliografia Básica

- VELOSO, Paulo; SANTOS, Clesio dos. Estruturas de dados. 13.ed. Rio de Janeiro: Campus, 1983. 228p.
- VILLAS, Marcos Vianna, FERREIRA, Andrea Gomes de Matos. Estruturas de dados: conceitos e técnicas de implementação. 6.ed. Rio de Janeiro: Campus, 1993. 298p.
- TENENBAUM, Aaron M.; LANGSAM, Yedidah. Estruturas de dados usando C. São Paulo: Makron Books, 1995. 884p.
- FORBELONE, André; EBERSPÄCHER, Henri. Lógica de Programação: a construção de algoritmos e estruturas de dados - 3ª edição. Editora Pearson. 232 p. (Biblioteca Virtual Pearson).

# Bibliografia Complementar

- SWAIT JUNIOR, Joffre Dan. Fundamentos computacionais, algoritmos e Estruturas de dados. São Paulo: Makron Books, 1991. 295p.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. Estruturas de dados e seus algoritmos. Rio de Janeiro: Conselho Regional de Administração, 1994. 320p.
- OLIVEIRA, S. M. Estruturas de dados com pascal. Rio de Janeiro: Infobook, 1993. 197p.

# Bibliografia Complementar

- FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. Lógica de programação: a construção de algoritmos e Estruturas de dados. 3.ed. São Paulo: Prentice Hall, 2005. 218p.
- FARRER, Harry; BECKER, Alfredo Augusto. Algoritmos estruturados. 3.ed. Rio de Janeiro: LTC, 1999. 284p.