Linguagem de Programação C

Notas de Aulas
(parte 1)

Professora: Luciana Rita Guedes

março/2014

Índice

1. LINGUAGEM C – INTRODUÇÃO

1.1 Características do C:

- **A linguagem C é uma linguagem de alto nível, genérica;**
- **Tem características como flexibilidade e portabillidade;**
- **Classifica-se como um linguagem estruturada;**
- **Programas em C passam pelo processo de compilação para gerar programas executáveis;**
- **Nasceu na década de 70 a partir de uma linguagem chamada B.**

1.2 Estrutura de um programa em C

- **um cabeçalho contendo as *diretivas de compilador* onde se definem o valor de constantes simbólicas, declaração de variáveis, inclusão de bibliotecas, declaração de rotinas, etc.**
- **um bloco de instruções *principal* e outros blocos de *rotinas*.**
- **documentação do programa: *comentários*.**

Exemplo 1: Primeiro programa em C

```
/* ************************************************************
 Proposito: Calcula média e frequencia de varios alunos.
************************************************************ */
#include <stdio.h>
#define MEDMIN 7       // nota mínima
#define FRQMIN 75      // freqüência mínima
main()        // inicia programa principal...
{
   float N1,N2,N3,MEDIA;   // 3 notas do aluno e media
   int   AULAS,FALTAS;     // total de aulas e faltas do aluno
   float FREQ;             // frequencia do aluno
   char  RESP;             // resposta do usuário

   printf("\n\nDigite o nr. total de aulas dadas: ");
   scanf("%i",&AULAS);

   do              // faca...
   {
      printf("\nDigite a primeira nota do aluno: ");
      scanf("%f",&N1);
      printf("\nDigite a segunda nota do aluno: ");
      scanf("%f",&N2);
      printf("\nDigite a terceira nota do aluno: ");
      scanf("%f",&N3);
      printf("\nDigite o nr. de faltas do aluno: ");
      scanf("%i",&FALTAS);

      MEDIA = (N1+N2+N3)/3;
      FREQ = (AULAS - FALTAS)*100/AULAS;

      printf("\nMédia do aluno.....: %.2f",MEDIA);
      printf("\nFreqüência do aluno: %.2f\n",FREQ);

      if ((MEDIA>=MEDMIN) && (FREQ>=FRQMIN)) // se ...
      {
         puts("Resultados foram bons.");
      }
      else                  // ...senao...
      {
         puts("Resultados foram ruins.");
      };

      printf("\nTecle S para fazer mais cálculos: ");
      scanf("%s",&RESP);

   } while (RESP=='S');   // enquanto usuário responder S

};  // fim do programa
```

2. CONSTANTES E VARIÁVEIS

2.1 Constantes

- **As constantes podem ser dos tipos:**
  - **inteiro:** `3, -45, 0x1a, 045`
  - **ponto flutuante:** `2.5, 0.234, 3.14E+3`
  - **caracter:** `'a', 'S', 'x'` **(armazenam números)**
  - **string: "Fulano de Tal"**

2.2 Identificadores válidos

- **Os identificadores devem seguir as seguintes regras de construção:**
  - **Os identificadores** devem **começar por uma letra (`a – z , A – Z`) ou um *underscore* (`_`).**
  - **O resto do identificador deve conter apenas letras, *underscores* ou dígitos (`0 – 9`).** Não pode **conter outros caracteres. Em C, os identificadores podem ter até 32 caracteres.**
  - **Em C, letras maiúsculas são** diferentes **de letras minúsculas: Por exemplo: `MAX`, `max`, `Max` são nomes diferentes para o compilador. Esta propriedade é chamada de *case sensibility*.**

Exemplo 2: Identificadores válidos:

```
abc, y24, VetorPontosMovimentoRobo, nota_1, TAM_MAX
```

Exemplo 3: Identificadores não-válidos:

```
3dia, vetor-1, pao&leite, iteração.
```

2.3 Tipos de Dados

2.3.1 Tipos Básicos

Tipo	Tamanho	Intervalo	Uso
char	1 byte	-128 a 127	nº muito pequeno e caracter ASCII
int	2 bytes	-32768 a 32767	contador, controle de laço
float	4 bytes	3.4e-38 a 3.4e38	real (precisão de 7 dígitos)
double	8 bytes	1.7e-308 a 1.7e308	científico (precisão de 15 dígitos)

2.3.2 Declaração de variáveis:

- **A sintaxe para declaração de variáveis é a seguinte:**

```
tipo variavel_1 [, variavel_2, ...] ;
```

- **Onde *tipo* é o tipo de dado e *variavel_1* é o nome da variável a ser declarada. Se houver mais de uma variável, seus nomes são separados por vírgulas.**

Exemplo 4: Declaração de variáveis:

```
int i;
int x,y,z;
char letra;
float nota_1,nota_2,media;
double num;
```

2.3.3 Tipos modificados

Tipo	Tam.(bytes)	Intervalo
unsigned char	1	0 a 255
unsigned int	2	0 a 65 535
long int	4	-2.147.483.648 a 2.147.483.647
unsigned long int	4	0 a 4 294 967 295
long double	10	3.4e-4932 a 1.1e4932

2.3.4 Strings

- **Para declararmos uma variável para receber um conjunto caracter devemos escrever:**

```
char* var;
```

Exemplo 5: Declaração de string

```
char* nome;
nome = "Fulano de Tal";
```

2.3.5 Inicialização de variáveis

- **A sintaxe para a inicialização de variáveis é:**

```
tipo var_1 = valor_1 [, var_2 = valor_2, ...] ;
```

- **Onde *tipo* é o tipo de dado, *var_1* é o nome da variável a ser inicializada e *valor_1* é o valor inicial da variável.**

Exemplo 6: Inicialização de variáveis:

```
int i = 0, j = 100;
float num = 13.5;
char* titulo = " Programa Teste ";
```

2.4 Constantes simbólicas

2.4.1 Constantes definidas pelo programador

- **O programador pode definir constantes simbólicas em qualquer programa. A sintaxe da instrução de definição de uma constante simbólica é:**

```
#define nome valor
```

- **Onde `#define` é uma diretiva de compilação que diz ao compilador para trocar as ocorrências do texto *`nome`* por *`valor`*. Observe que *não há ;* (ponto-e-vírgula) no final da instrução pois trata-se de um comando para o compilador e não para o processador. A instrução `#define` deve ser escrita antes da instrução de declaração da rotina principal.**

Exemplo 7: a seguir definimos algumas constantes simbólicas.

```
#define PI  3.14159
#define ON  1
#define OFF 0
#define ENDERECO 0x378
void main(){
...
```

2.4.2 Constantes pré-definidas:

Biblioteca	Constante	Valor	Significado
math.h	M_PI	3.14159...	π
math.h	M_PI_2	1.57079...	π/2
math.h	M_PI_4	0,78539...	π/4
math.h	M_1_PI	0,31830...	1/π
math.h	M_SQRT2	1,41421...	√2
conio.h	BLACK	0	valor da cor (preto)
conio.h	BLUE	1	valor da cor (azul)
conio.h	GREEN	2	valor da cor (verde)
conio.h	CYAN	3	valor da cor (cyan)
conio.h	RED	4	valor da cor (vermelho)
conio.h	MAGENTA	5	valor da cor (magenta)
limits.h	INT_MAX	32767	limite superior do tipo int
limits.h	INT_MIN	-32768	limite inferior do tipo int

3. OPERADORES, EXPRESSÕES E FUNÇÕES

3.1 Operador de Atribuição

- **A operação de consiste de atribuir valor de uma *expressão* a uma *variável*. A sintaxe da operação de atribuição é a seguinte:**

```
identificador = expressão;
```

- **onde *identificador* é o nome de uma variável e *expressão* é uma expressão válida (ou outro identificador).**

Exemplo 8: Operações de atribuição válidas

```
a = 1;
delta = b * b - 4. * a * c;
i = j;
```

3.1.1 Atribuição Múltipla

- **É possível atribuir um valor a muitas variáveis em uma única instrução. A esta operação dá-se o nome de *atribuição múltipla*. A sintaxe da atribuição múltipla é seguinte:**

```
var_1 = [var_2 = ... ] expressão;
```

- **onde var_1, var_2, ... são os identificadores de variáveis e expressão é uma expressão válida.**

Exemplo 9: Operações de atribuição múltipla:

```
int i, j, k;
double max, min;
i = j = k = 1;
max = min = 0.0;
```

3.2 Operadores Aritméticos

Operador	Operação
+	adição.
-	subtração.
*	multiplicação
/	divisão
%	módulo (resto da divisão inteira)

Obs: o operador % só pode ser usado com operandos inteiros

- **A sintaxe de uma *expressão aritmética* é:**

```
operando operador operando
```

- **onde *operador* é um dos caracteres mostrados acima e *operando* é uma constante ou um identificador de variável.**

Exemplo 10: Algumas expressões aritméticas:

```
1+2   a-4.0   b*c   valor_1/taxa   num%2
```

3.3 Operadores de atribuição aritmética

- **Muitas vezes queremos alterar o valor de uma variável realizando alguma operação aritmética com ela, como por exemplo: `i = i + 1` ou `val = val * 2`.**
- **A linguagem C possui instruções otimizadas com o uso de *operadores de atribuição aritmética*.**
- **A sintaxe da atribuição aritmética é a seguinte:**

```
var += exp;   // corresponde a var = var + exp;
var -= exp;   // corresponde a var = var - exp;
var *= exp;   // corresponde a var = var * exp;
var /= exp;   // corresponde a var = var / exp;
var %= exp;   // corresponde a var = var % exp;
```

- **onde *var* é o identificador da variável e *exp* é uma expressão válida.**

Exemplo 11: Operadores de atribuições aritméticas:

Atribuição aritmética	Instrução equivalente
`i += 1;`	`i = i + 1;`
`j -= val;`	`j = j - val;`
`num *= 1 + k;`	`num = num * (1 + k);`
`troco /= 10;`	`troco = troco / 10;`
`resto %= 2;`	`resto = resto % 2;`

3.4 Operadores incrementais

- **Uma instrução de incremento *adiciona* (++) uma unidade ao conteúdo de uma variável. Uma instrução de decremento *subtrai* (--) uma unidade do conteúdo de uma variável.**
- **A sintaxe dos operadores incrementais é a seguinte:**

	instrução equivalente
`++ var`	`var = var + 1`
`var ++`	`var = var + 1`
`-- var`	`var = var - 1`
`var --`	`var = var – 1`

- **onde *`var`* é o nome da variável da qual se quer incrementar ou decrementar um unidade.**

- **Se o operador for colocado *à esquerda* da variável, o valor da variável será incrementado (ou decrementado) *antes* que a variável seja usada em alguma outra operação.**
- **Caso o operador seja colocado *à direita* da variável, o valor da variável será incrementado (ou decrementado) *depois* que a variável for usada em alguma outra operação.**

Exemplo 12: Operadores incrementais

	valor das variáveis
`int a, b, c, i = 3;`	`// a: ?  b: ?  c: ?  i: 3`
`a = i++;`	`// a: 3  b: ?  c: ?  i: 4`
`b = ++i;`	`// a: 3  b: 5  c: ?  i: 5`
`c = --i;`	`// a: 3  b: 5  c: 4  i: 4`

3.5 Operadores relacionais e lógicos

- **Uma condições de controle é uma *expressão lógica* que é avaliadas como *verdadeira* (valor igual a 1) ou *falsa* (valor igual a 0). Uma expressão lógica é construída com *operadores relacionais* e *lógicos*.**

3.5.1 Operadores relacionais

Operador	Significado
>	maior que
<	menor que
>=	maior ou igual a (não menor que)
<=	menor ou igual a (não maior que)
==	igual a
!=	não igual a (diferente de)

- Sintaxe: **A sintaxe das expressões lógicas é:**

```
expressão_1 operador expressão_2
```

- **onde *expressão_1* e *expressão_2* são duas expressões numéricas quaisquer, e *operador* é um dos operadores relacionais.**

Exemplo 13: Expressões lógicas

Suponha que i e j são variáveis int cujos valores são 5 e -3, respectivamente.
As variáveis r e s são float com valores 7.3 e 1.7, respectivamente.

Expressão	Valor
i == 7	0
r != s	1
i > r	0
6 >= i	1
i < j	0
s <= 5.9	1

3.5.2 Operadores lógicos

- **São três os operadores lógicos de C: `&&`, `||` e `!`. Estes operadores tem a mesma significação dos operadores lógicos booleanos AND, OR e NOT.**
- **A sintaxe de uso dos operadores lógicos é:**

```
expr_1 && expr_2    // E (AND)
expr_1 || expr_2    // OU (OR)
!expr               // NÃO (NOT)
```

- **onde *expr_1* , *expr_2* e *expr* são expressões quaisquer.**

Tabelas Verdade

Operador &&: op_1 && op_2	op_1	op_2	Res
	1	1	1
	1	0	0
	0	1	0
	0	0	0

Operador ||: op_1 || op_2	op_1	op_2	Res
	1	1	1
	1	0	1
	0	1	1
	0	0	0

Operador !: !op	op	Res
	1	0
	0	1

Precedência de Operadores na Linguagem C

Categoria	Operadores	Prioridade
parênteses	`( )`	**interno → externo**
função	`nome()`	**Esquerda → Direita**
incremental, lógico	`++ -- !`	**Esquerda ← Direita**
aritmético	`* / %`	**Esquerda → Direita**
aritmético	`+ -`	**Esquerda → Direita**
relacional	`< > <= >=`	**Esquerda → Direita**
relacional	`== !=`	**Esquerda → Direita**
lógico	`&&`	**Esquerda → Direita**
lógico	`||`	**Esquerda → Direita**
condicional	`?:`	**Esquerda ← Direita**
atribuição	`= += -= *= /= %=`	**Esquerda ← Direita**

Maior precedência no topo, menor precedência na base.

4. ENTRADA E SAÍDA

4.1 Saída formatada: printf()

- **Biblioteca: `stdio.h`**
- **Declaração:**

```
int printf (const char* st_contr [, lista_arg]);
```

- **Propósito: A função `printf()` (*print formated*) imprime dados da lista de argumentos *`lista_arg`* formatados de acordo com a string de controle *`st_contr`*.**

- **Sintaxe:**

Depois do sinal %, seguem-se alguns modificadores, cuja sintaxe é a seguinte:

% *[flag] [tamanho] [.precisão] tipo*

[flag]	**justificação de saída: (Opcional)**
-	justificação à esquerda.
+	conversão de sinal (saída sempre com sinal: + ou -)
<espaço>	conversão de sinal (saídas negativas com sinal, positivas sem sinal)

[tamanho]	**especificação de tamanho (Opcional)**
n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por brancos).
0n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por zeros).

[.precisão]	**especificador de precisão, dígitos a direita do ponto decimal. (Opcional)**
(nada)	padrão: 6 dígitos para reais.
.0	nenhum digito decimal.
.n	são impressos n dígitos decimais.

Tipo	**caracter de conversão de tipo (Requerido)**
d	inteiro decimal
o	inteiro octal
x	inteiro hexadecimal
f	ponto flutuante: [-]dddd.dddd.
e	ponto flutuante com expoente: [-]d.dddd**e**[+/-]ddd
c	caracter simples
s	string

Exemplo 14: Saída formatada

```
Instrução:
  printf("Tenho %d anos de vida",idade);
Saída:
  Tenho 29 anos de vida

Instrução:
  printf("Total: %f.2 \nDinheiro: %f.2 \nTroco:
  %f.2",tot,din,din-tot);
Saída:
  Total: 12.30
  Dinheiro: 15.00
  Troco: 2.70
```

4.2 Leitura formatada: scanf()

- **Biblioteca: `stdio.h`**
- **Declaração:**

```
int scanf(const char* st_contr [, end_var, ...]);
```

- **Propósito: A função `scanf()` (*scan formated*) permite a entrada de dados numéricos, caracteres e 'strings' e sua respectiva atribuição a variáveis cujos endereços são *`end_var`*.**

- **Sintaxe:**

Depois do sinal %, seguem-se alguns modificadores, cuja sintaxe é a seguinte:

```
[*] [tamanho] tipo
```

*****	**indicador de supressão (Opcional)**
<presente>	Se o indicador de supressão estiver presente o campo não é lido. Este supressor é útil quando não queremos ler um campo de dado armazenado em arquivo.
<ausente>	O campo é lido normalmente.

Tamanho	**especificador de tamanho(Opcional)**
n	Especifica n como o numero máximo de caracteres para leitura do campo.
<ausente>	Campo de qualquer tamanho.

Tipo	**define o tipo de dado a ser lido (Requerido)**
d	inteiro decimal (int)
f	ponto flutuante (float)
o	inteiro octal (int)
x	inteiro hexadecimal (int)
i	inteiro decimal de qualquer formato(int)
u	inteiro decimal sem sinal (unsigned int)
s	string (char*)
c	caracter (char)

4.3 Entrada de caracter individual: getchar()

- **Biblioteca: `stdio.h`**
- **Declaração:**

```
int getchar(void);
```

- **Propósito: A função `getchar()` (*get character*) lê um caracter individual da entrada padrão (em geral, o teclado).**

4.4 Saída de caracter individual: putchar()

- **Biblioteca: `stdio.h`**
- **Declaração:**

```
int putchar(int c);
```

- **Propósito: Esta função `putchar()` (*put character*) imprime um caracter individual `c` na saída padrão (em geral o monitor de vídeo).**

4.5 Leitura de teclado: getch(), getche()

- **Biblioteca: `conio.h`**
- **Declaração:**

```
int getch(void);
int getche(void);
```

- **Propósito: Estas funções fazem a leitura dos códigos de teclado. Estes códigos podem representar tecla s de caracteres (`A`, `y`, `*`, `8`, etc.), teclas de comandos (`[enter]`, `[delete]`, `[Page Up]`, `[F1]`, etc.) ou combinação de teclas (`[Alt]` + `[A]`, `[Shift]` + `[F1]`, `[Ctrl]` + `[Page Down]`, etc.).**

- **Ao ser executada, a função `getch()` (*get character*) aguarda que uma tecla (ou combinação de teclas) seja pressionada, recebe do teclado o código correspondente e retorna este valor.**

- **A função `getche()`(*get character and echoe*) também escreve na tela, quando possível, o caracter correspondente.**

5. ESTRUTURAS DE CONTROLE

5.1 Condição de controle

- **Uma condição de controle é uma expressão lógica ou aritmética cujo resultado pode ser considerado verdadeiro ou falso.**
- **A linguagem C não possui, entretanto, variáveis ou constantes lógicas, possui somente expressões numéricas, assim quando uma *expressão numérica* se encontra em uma *condição de controle*, ela será considerada *falsa* se seu valor for *igual a zero*, e *verdadeira* se seu valor for *diferente de zero*.**

Exemplo 15: Condições de controle.

```
Considere as variáveis int i = 0, j = 3;
```

condição	valor numérico	significado lógico
(i == 0)	1	verdadeiro
(i > j)	0	falso
(i)	0	falso
(j)	3	verdadeiro

5.2 Estrutura do...while

- **Esta é uma estrutura básica de repetição condicional. Permite a execução de um bloco de instruções repetidamente. Sua sintaxe é a seguinte:**

- **Sintaxe:**

```
do
{
  bloco
} while(condição);
```

- **onde: *condição* é uma expressão lógica ou numérica.**
  ***bloco* é um conjunto de instruções.**

- **Esta estrutura faz com que o bloco de instruções seja executado pelo menos uma vez. Após a execução do bloco, a condição é avaliada. Se a condição é *verdadeira* o bloco é executado outra vez, caso contrário a repetição é terminada.**

Exemplo 16: Uso da estrutura do...while...

```
do
{
   puts("Digite um número positivo:");
   scanf("%f",&num);
} while(num <= 0.0);
```

5.3 Estrutura while

- **A estrutura de repetição condicional `while` é semelhante a estrutura `do...while`. Sua sintaxe é a seguinte:**

- **Sintaxe:**

```
while(condição)
{
   bloco
}
```

- **onde: *`condição`* é uma expressão lógica ou numérica.**
  ***`bloco`* é um conjunto de instruções.**

- **Esta estrutura faz com que a condição seja avaliada em primeiro lugar. Se a condição é *verdadeira* o bloco é executado uma vez e a condição é avaliada novamente. Caso a condição seja *falsa* a repetição é terminada sem a execução do bloco. Nesta estrutura, o bloco de instruções pode não ser executado nenhuma vez, desde que a condição seja inicialmente falsa.**

Exemplo 17: Uso da estrutura while

```
puts("Digite o valor máximo a imprimir:");
scanf("%i",&num);
x = 1;
while( x <= num )
{
  printf("%i\n",x);
  x++;
}
```

5.4 Estrutura for

- **A estrutura de repetição `for` costuma ser utilizada quando se quer um número determinado de ciclos. A contagem dos ciclos é feita por uma variável chamada de *contador*. A estrutura `for` é, as vezes, chamada de estrutura de *repetição com contador*.**
- **Sintaxe:**

```
for(inicialização; condição; incremento)
{
   bloco
}
```

- **onde:**

`inicialização` **é uma expressão de inicialização do contador.**
`condição` **é uma expressão lógica de controle de repetição.**
`incremento` **é uma expressão de incremento do contador.**
`bloco` **é um conjunto de instruções a ser executado.**

Exemplo 18: Uso da estrutura for

```
for(i=1; i<=10; i++)
{
  printf(" %d",i);
}
```

- **No trecho anterior, o contador `i` é inicializado com o valor `1`. O bloco é repetido enquanto `i<=10`. O contador é incrementado com a instrução `i++`. Esta estrutura, deste modo, imprime os números `1, 2, ..., 9, 10`.**

5.5 Estrutura de decisao if...else

- **A estrutura `if...else` permite executar um entre vários blocos de instruções. O controle de qual bloco será executado será dado por uma *`condição`* (expressão lógica ou numérica). Esta estrutura pode se apresentar de modos ligeiramente diferentes.**

5.5.1 Decisão de um bloco if

- **A estrutura de decisão de um bloco permite que se execute (ou não) um bloco de instruções conforme o valor de uma condição seja verdadeiro ou falso. Se a condição** verdadeira**, o *`bloco`* é executado. Caso contrário, o bloco não é executado.**
- **Sintaxe**:

```
if (condição)
{
   bloco
}
```

- **onde: *`condição`* é uma expressão lógica ou numérica.**
  ***`bloco`* é um conjunto de instruções.**

Exemplo 19: Decisão com um bloco

```
printf("Digite o número de repetições: (máximo
10)";
scanf("%d",&iter);
if(iter > 10)
{
   iter = 10;
}
```

5.5.2 Decisão de dois blocos if...else...

- **É possível escrever uma estrutura que execute um entre dois blocos de instruções. Se a `condição` for** verdadeira **o *`bloco 1`* é executado. Caso contrário, o *`bloco 2`* é executado.**
- **Sintaxe:**

```
if (condição)
{
   bloco 1;
}
else
{
   bloco 2;
}
```

- **onde: *`condição`* é uma expressão lógica ou numérica.**
  ***`bloco1`* e *`bloco2`* são conjuntos de instruções.**

```
if (raiz*raiz > num)
{
   max = raiz;
}
else
{
   min = raiz;
}
```

5.5.3 Decisao de múltiplos blocos if...else...if...

- **É possível escrever uma estrutura que execute um entre múltiplos blocos de instruções.**
- **Se a condição 1 for** verdadeira **o *`bloco 1`* é executado. Caso contrario, a condição 2 é avaliada. Se a condição 2 for** verdadeira **o *`bloco 2`* é executado. Caso contrario, a condição 3 é avaliada e assim sucessivamente. Se nenhuma condição é** verdadeira ***`bloco P`* é executado. Observe que apenas um dos blocos é executado.**

- **Sintaxe:**

```
if(condição 1)
{
  bloco 1;
  ...
}
else if(condição N)
{
  bloco N;
}
else
{
  bloco P
}
```

- **onde:**
  condição 1, condição 2,... **são expressões lógicas ou numéricas.**
  bloco 1, bloco 2,... **são conjuntos de instruções.**

Exemplo 20: Decisão de múltiplos blocos

```
if(num > 0)
{
  a = b;
}
else if(num < 0)
{
  a = b + 1;
}
else
{
  a = b - 1;
}
```

5.6 Estrutura switch...case...

- **A estrutura `switch...case` é uma estrutura de decisão que permite a execução de um conjunto de instruções a partir pontos diferentes conforme o resultado de uma expressão inteira de controle.**

- **O resultado desta expressão é comparado ao valor de cada um dos rótulos, e as instruções são executadas a partir desde rótulo.**

- **Sintaxe:**

```
switch (var)
  {
     case valor1:
        bloco 1;
     case valor2:
        bloco 2;
     ...
     case valorn:
       bloco n;
  }
```

- **onde: *var* é uma variável numérica**
  ***valor 1, valor 2*,... são expressões numéricas**
  ***bloco 1,bloco 2*,... são conjuntos de instruções.**

Exemplo 21: Estrutura switch...case...

```
puts("Digite o código do produto:");
scanf("%d",codigo);
switch (codigo)
{
   case 1: preco = 0.60;
   case 2: preco = 0.50;
   case 3: preco = 0.20;
   case 4: preco = 1.00;
}
```

6. FUNÇÕES

- **Funções (também chamadas de *rotinas*, ou *sub-programas*) são segmentos de programa que executam uma determinada tarefa específica.**

- **Já vimos o uso de funções já providenciadas pelas bibliotecas-padrão do C (como `sqrt()`, `getch()` ou `putchar()`).**

- **É possível ao programador, alem disso, escrever suas próprias rotinas. São as chamadas de *funções de usuário* ou rotinas de usuário.**

- **Deste modo pode-se segmentar um programa grande em vários programas menores. Esta segmentação é chamada de *modularização* e permite que cada segmento seja escrito, testado e revisado individualmente sem alterar o funcionamento do programa como um todo.**

6.1 Estrutura das funções de usuário

- **Uma função de usuário constitui-se de um *bloco de instruções* que definem os procedimentos efetuados pela função, um *nome* pelo qual a chamamos e uma *lista de argumentos* passados a função. Chamamos este conjunto de elementos de *definição da função*.**

Exemplo 22: Função para calcular média de dois números:

```
float media2(float a, float b)
{
     float med;
     med = (a + b) / 2.0;
     return(med);
}
```

- **No exemplo acima definimos uma função chamada `media2` que recebe dois argumentos tipo `float`: `a` e `b`.**
- **A média destes dois valores é calculada e armazenada na variável `med` declarada internamente.**
- **A função retorna, para o programa que a chamou, um valor também do tipo `float`: o valor da variável `med`.**
- **Este retorno de valor é feito pela função `return()` que termina a execução da função e retorna o valor de `med` para o programa que a chamou.**

- **Depois de definimos um função, podemos usá-la dentro de um programa qualquer. Dizemos que estamos fazendo uma *chamada* a função.**

Exemplo 23: Chamado à função dentro do programa principal

```
void main()
{
   float num_1, num_2, m;
   puts(”Digite dois números:”);
   scanf(”%f %f”, &num_1, &num_2);
   m = media2(num_1, num_2);   // chamada a função
   printf(”\nA media destes números e´ %f”, m);
}
```

- **No exemplo acima chamamos a função `media2()` dentro de um programa principal.**

6.2 Definição de funções

- **De modo formal, a sintaxe de uma função é a seguinte:**

```
tipo_retorno nome_função(tipo_1 arg_1, tipo_2 arg_2, ...)
  {
    [bloco de instruções da função]
  }
```

- **A primeira linha da função contém a *declaração* da**

função. Na declaração de uma função se define o *nome* da função, seu *tipo de retorno* e a *lista de argumentos* que recebe.

- **Em seguida, dentro de chaves `{}`, definimos o bloco de instruções da função.**

  - **O *tipo de retorno* da função especifica qual o tipo de dado retornado pela função, podendo ser qualquer tipo de dado: `int`, `float`, etc. Se a função não retorna nenhum valor para o programa que a chamou devemos definir o retorno como `void`, ou seja um retorno ausente. Se nenhum tipo de retorno for especificado o compilador entenderá que o retorno será tipo `int`.**
    - **Vale notar que existe apenas *um* valor de retorno para funções em C. Porém isto não é um limitação séria pois o uso de ponteiros (mais adiante) contorna o problema.**

- **O *nome da função* segue as mesmas regras de definição de identificadores.**

- **A *lista de argumentos* da função especifica quais são os valores que a função recebe. As variáveis da lista de argumentos podem ser manipuladas normalmente no corpo da função.**

- **A chamada de uma função termina com a instrução `return()` que transfere o controle para o programa chamador da função. Esta instrução tem duas finalidades: determina o *fim lógico* da rotina e o *valor de retorno* da função. O argumento de `return()` será retornado como valor da função.**

6.3 Localização das funções

- **Existem basicamente duas posições possíveis para escrevermos o corpo de uma função: ou *antes* ou *depois* do programa principal. Podemos ainda escrever uma função no *mesmo arquivo* do programa principal ou em *arquivo separado*.**

6.3.1 Corpo da função antes *do programa principal (no mesmo arquivo)*

- **Quando escrevemos a definição de uma função** antes **do programa principal e no mesmo arquivo deste, nenhuma outra instrução é necessária.**

Exemplo 24: Função acima do programa principal.

```
float media2(float a, float b)  // função
{
     float med;
     med = (a + b) / 2.0;
     return(med);
}

void main()                  // programa principal
{
   float num_1, num_2, med;
   puts("Digite dois números:");
   scanf("%f %f", &num_1, &num_2);
   med = media2(num_1, num_2);  //chamada à função
   printf("\nA media destes números e´ %f", med);
}
```

6.3.2 Corpo da função depois *do programa principal (no mesmo arquivo)*

- **Quando escrevemos a definição de uma função *depois* do programa principal e no mesmo arquivo deste, devemos incluir um *protótipo* da função chamada.**
- **Um protótipo é uma instrução que define o nome da função, seu tipo de retorno e a quantidade e o tipo dos argumentos da função. Os protótipos das funções indicam ao compilador quais funções serão usadas no programa principal.**

Exemplo 25: Função de usuário após o programa principal.

```
void main()                    // programa principal
{

   float media2(float,float);  // protótipo de media2()

   float num_1, num_2, med;
   puts(”Digite dois números:”);
   scanf(”%f %f”, &num_1, &num_2);
   med = media2(num_1, num_2);    // chamada à função
   printf(”\nA media destes números e´ %f”, med);
}

float media2(float a, float b)  // função media2()
{
   float med;
   med = (a + b) / 2.0;
   return(med);
}
```

6.3.3 Corpo da função escrito em arquivo separado

- **É possível criair uma função em um *arquivo separado* e fazer com que um programa a chame em outro *arquivo distinto*.**
- **Esta facilidade permite a criação de *bibliotecas de usuário*: um conjunto de arquivos contendo funções escritas pelo usuário.**
- **Quando escrevemos a definição de uma função em *arquivo separado* do programa principal devemos *incluir* este arquivo no conjunto de arquivos de *compilação* do programa principal. Esta inclusão é feita com a diretiva `#include`. Esta diretiva instrui o compilador para incluir na compilação do programa outros arquivos que contém a definição das funções de usuário e de biblioteca.**

Exemplo 26: Função de usuário escrita em arquivo separado.

```
#include "c:\tc\userbib\stat.h"    // inclusão da função
void main()                        // programa principal
{
   float num_1, num_2, med;
   puts("Digite dois números:");
   scanf("%f %f", &num_1, &num_2);
   med = media2(num_1, num_2);     // chamada à função
   printf("\nA media destes números e´ %f", med);
}
```

6.4 Hierarquia de funções

- **Sempre é possível que um programa principal chame uma função que por sua vez chame outra função... e assim sucessivamente. Quando isto acontece dizemos que a função chamadora tem *hierarquia maior que* (ou superior a) função chamada. Ou que a função chamadora está em um *nível hierárquico superior* à função chamada.**
- **Quando isto ocorre, devemos definir (ou incluir) as funções em ordem crescente de hierarquia, isto é, uma função *chamada* é escrita antes de uma função *chamadora*.**

6.5 Regra de escopo para variáveis

- **A regra de escopo define o *âmbito de validade* de variáveis. Em outras palavras define onde as variáveis e funções são reconhecidas. Em C, uma variável só pode ser *usada* após ser *declarada*.**
- **O local do programa onde uma variável é declarada define seu *escopo* de validade.**
- **Uma variável pode ser *local*, *global* ou *formal* de acordo com o local de declaração.**

6.5.1 Variáveis Locais

- **Uma variável é dita *local,* se for declarada *dentro do bloco* de uma função.**
- **Uma variável local tem validade apenas dentro do bloco onde é declarada, isto significa que podem ser apenas acessadas e modificadas dentro de um bloco.**

- **O espaço de memória alocado para esta variável é *criado* quando a execução do bloco é iniciada e *destruído* quando encerrado. Assim, variáveis de mesmo nome mas declaradas em *blocos distintos*, são *variáveis distintas*.**

6.5.2 Variáveis Formais

- **Uma *variável formal* é uma variável local declarada na *lista de parâmetros* de uma função. Deste modo, tem validade apenas *dentro da função* onde é declarada, porém serve de suporte para os valores passados pelas funções.**
- **As variaveis formais na *declaração* da função e na *chamada* da função podem ter nomes distintos. A única exigência é de que sejam do mesmo tipo.**
- **Por serem variáveis locais, os valores que uma função passa para outra não são alterados pela função chamada. Este tipo de passagem de argumentos é chamado de** passagem por valor **pois os valores das variáveis do programa chamador são copiados para as correspondentes variáveis da função chamada.**

Exemplo 27: Uso de variáveis locais e formais

- **No exemplo a seguir, `med` é uma variável local definida pela função `media2()`. Outra variável `med` é também definida pela função `main()`. Para todos os efeitos estas variáveis *são distintas*.**
- **As variáveis `a` e `b` são parâmetros formais (variáveis formais) declarados na função media2(). Observe que a função é chamada com os valores de `num_1` e `num_2`. Mesmo que os valores de `a` e `b` fossem alterados os valores de `num_1` e `num_2` não seriam alterados.**

```
/* Exemplo de variáveis locais e formais */

float media2(float a, float b)
// a E b da linha acima SÃO VARIÁVEIS FORMAIS
{
     float med;            // med é VARIÁVEL LOCAL
     med = (a + b) / 2.0;
     return(med);
}

void main()
{
     float num_1, num_2, med;
     puts(”Digite dois números:”);
     scanf(”%f %f”, &num_1, &num_2);
     med = media2(num_1, num_2);
     printf(”\nA media dos números e´ %f”, med);
}
```

6.5.3 *Variáveis Globais*

- **Uma variável é dita *global*, se for declarada *fora do bloco* de uma função. Uma variável global tem validade no escopo de *todas as funções*, isto é, pode ser acessadas e modificada por qualquer função.**
- **O espaço de memória alocado para esta variável é** criado **no momento de sua declaração e** destruído **apenas quando o programa é encerrado.**

Exemplo 28: Uso de variáveis globais.

- **No exemplo a seguir, `a`, `b`, `med` são variáveis globais definidas fora dos blocos das funções `media()` e `main()`. Deste modo ambas as funções tem pleno acesso às variáveis, podendo ser acessadas e modificadas por**

quaisquer uma das funções. Assim não é necessário a passagem de parâmetros para a função.

```
/* Exemplo de uso de variáveis globais */

float a, b, med;  // a,b,med são VARIÁVEIS GLOBAIS
                  // pois estão definidas fora dos
                  // blocos das funções
void media2(void)
{
  med = (a + b) / 2.0;
}

void main()
{
    puts("Digite dois números:");
    scanf("%f %f", &a, &b);
    media2();  // chamada à função sem argumentos
    printf("\nA media dos números e´ %f", med);
}
```

6.6 Recursividade

- ***Recursividade*** **ou** ***recursão*** **é o processo pelo qual uma função chama a si mesma repetidamente um número finito de vezes. Esta característica é muito útil em alguns tipos de algoritmos chamados de** ***algoritmos recursivos*****.**

Exemplo 29: Uso de recursividade

- **Um exemplo** ***clássico*** **de algoritmos recursivo é o cálculo do** ***fatorial*** **de um número. A definição de fatorial é:**

```
n! = n . (n-1) . (n-2) . ... . 3 . 2 . 1
0! = 1
```

- **onde `n` é um numero inteiro positivo. Uma propriedade dos fatoriais é que:**

```
n! = n . (n-1)!
```

- **Esta propriedade é chamada de *propriedade recursiva*: o fatorial de um numero pode ser calculado através do fatorial de seu antecessor.**
- **Podemos utilizar esta propriedade para escrevermos uma rotina recursiva para o calculo de fatoriais. Para criarmos uma rotina recursiva, basta criar uma chamada a própria função dentro dela mesma, como no exemplo a seguir.**

```
/* Função recursiva para o cálculo do fatorial */

long double fat(unsigned n) // declaracao da funcao
{
   long double valor;         // variavel temporaria
   if(n == 0.0)               // se fim da recursao...
   {
      valor = 1.0;            // calcula ultimo valor.
   }
   else                       // senao...(se valor>0)
   {
      valor = n * fat(n-1.0); // ... chama fat(n-1).
   };
   return(valor);             // retorna valor.
};
```

- **Uma função recursiva cria a cada chamada um novo conjunto de variáveis locais. Não existe ganho de velocidade ou espaço de memória significativo com o uso de funções recursivas.**
- **Teoricamente, um algoritmo recursivo pode ser escrito de forma iterativa e vice-versa.**
- **A principal vantagem é que algumas classes de algoritmos (inteligência artificial, busca e ordenação em arvore binaria, etc.) são mais facilmente implementadas com o uso de rotinas recursivas.**

7. VETORES

7.1 Introdução

- **Vetores (ou matrizes) são *estruturas de dados homogêneas*, ou seja, são conjuntos de dados que armazenam valores de mesmo tipo (int, float, double, char).**

Exemplo 30: Representação de vetor unidimensional

- **A maneira mais simples de entender um vetor é através da visualização de um *lista* de elementos com um nome coletivo e um índice de referência aos valores da lista.**

```
n   nota
0   8.4
1   6.9
2   4.5
3   4.6
4   7.2
```

- **Nesta lista, `n` representa um número de referência e `nota` é o nome do conjunto. Assim podemos dizer que a 2ª nota é `6.9` ou representar `nota[1] = 6.9`**

7.2 Declaração e inicialização de vetores

7.2.1 Declaração de vetores

- **Um vetor é um conjunto de variáveis de um *mesmo tipo* que possui um nome identificador e um índice de referência.**
- **A sintaxe para a declaração de um vetor é a seguinte:**

```
tipo nome[tam];
```

- **onde:**

  tipo **é o** tipo **dos elementos do vetor: `int, float, double`...**
  nome **é o** nome **identificador do vetor e segue as mesmas regras de nomenclatura usadas em variáveis;**
  tam **é o tamanho do vetor, isto é, o número de elementos que o vetor pode armazenar.**

Exemplo 31: Declarações de vetores

```
int idade[100]; // declara um vetor chamado 'idade'
                // do tipo 'int' que pode
                // armazenar 100 elementos.

float nota[25]; // declara um vetor chamado 'nota'
                // do tipo 'float' que pode
                // armazenar 25 números.

char nome[80];  // declara um vetor chamado 'nome'
                // do tipo 'char' que pode
                // armazenar 80 caracteres.
```

- **Na declaração de um vetor estamos *reservando espaço de memória* para os elementos de um vetor.**
- **A quantidade de memória (em *bytes*) usada para armazenar um vetor pode ser calculada como:**

```
quantidade de memória = tamanho do tipo * tamanho do vetor
```

- **Assim, no exemplo anterior, a quantidade de memória utilizada pelos vetores é, respectivamente:**
  - **`200 bytes(2x100)`**
  - **`100 bytes(4x25)`**
  - **`80 bytes(80x1)`**

7.2.2 Referência a elementos de vetor

- **Cada elemento do vetor é referenciado pelo *nome* do vetor seguido de um *índice* inteiro.**
- **O *primeiro* elemento do vetor tem índice `0` e o *último* tem índice `tam-1`. O índice de um vetor deve ser *inteiro*.**

Exemplo 32: Referências a vetores

```
#define MAX 5
int i = 7;
float valor[10];           // declaração de vetor
valor[1] = 6.645;
valor[MAX] = 3.867;
valor[i] = 7.645;
valor[random(MAX)] = 2.768;
valor[sqrt(MAX)] = 2.705;     // NÃO é válido!
```

7.2.3 Inicialização de vetores

- **Assim como podemos inicializar variáveis podemos inicializar vetores, isto é, podemos atribuir valores iniciais para os elementos de um vetor.**
- **A sintaxe para a inicialização dos elementos de um vetor é:**

```
tipo nome[tam] = {lista de valores};
```

  - **onde: *`lista de valores`* é uma lista, separada por vírgulas, dos valores de cada elemento do vetor.**

Exemplo 33: Inicializando vetores na declaração

```
int dia[7] = {12,30,14,7,13,15,6};
float nota[5] = {8.4,6.9,4.5,4.6,7.2};
char vogal[5] = {'a', 'e', 'i', 'o', 'u'};
```

Exemplo 34: Inicializando vetores no corpo do programa

```
int cor_menu[4] = {BLUE,YELLOW,GREEN,GRAY};

// esta forma de inicialização é equivalente a:

int cor_menu[4];
cor_menu[0] = BLUE;
cor_menu[1] = YELLOW;
cor_menu[2] = GREEN;
cor_menu[3] = GRAY;
```

7.3 Tamanho de um vetor

- **Na linguagem C não é possível, usando a sintaxe descrita acima, declarar um vetor com tamanho *variável*.**

Exemplo 35: Declaração ERRADA de vetor c/ tam. variável

```
...
int num;
puts("Quantos números?");
scanf("%d", &num);
float valor[num];     // declaração ERRADA
...
```

- **No entanto, é possível declarar um vetor com tamanho *parametrizado*: usando uma *constante simbólica*.**
- **Declaramos uma constante simbólica (parâmetro) com a diretiva `#define` no cabeçalho do programa e depois declaramos o vetor com esta constante simbólica como tamanho do vetor.**
- **Deste modo podemos alterar o número de elementos do vetor *antes* de qualquer *compilação* do programa. Esta é uma maneira simples de administrar o espaço de memória usado pelo programa, e também testar os limites de um vetor.**

Exemplo 36: Declaração de vetor usando parâmetro

```
...
#define MAX 5     // definicao do parametro MAX
                  // este valor pode ser alterado
void main()
{
   int valor[MAX];   // declaracao do vetor c/ MAX
   ...
```

7.4 Passando vetores para funções

- **Vetores, assim como variáveis, podem ser usados como argumentos de funções.**
- **Na *passagem* de vetores para funções usamos a seguinte sintaxe:**

```
nome_da_função(nome_do_vetor)
```

  - **onde:**
    nome_da_função **é o nome da função a ser chamada.**
    nome_do_vetor **é o nome do vetor que queremos passar.**
    Indicamos *apenas* o nome do vetor, *sem* índices.

- **Na *declaração* de funções que recebem vetores:**

```
tipo_função nome_função(tipo_vetor nome_vetor[])
{
   ...
}
```

  - **onde:**
    tipo_função **é o tipo de retorno da função.**
    nome_função **é o nome da função.**
    tipo_vetor **é o tipo de elementos do vetor.**
    nome_vetor **é o nome do vetor.**
    Depois do nome do vetor temos um índice vazio `[]` para indicar que estamos *recebendo* um vetor.

Exemplo 37: Passagem de vetor para funções

```
// Na declaração da função:
float media(float vetor[],float N)
{
  ...
}

// Na chamada da função:
void main()
{
   float valor[MAX]; // declaração do vetor
   ...
   med = media(valor, n); // vetor passado p/ a função
   ...
}
```

- **Atenção: Ao contrário das variáveis comuns, o conteúdo de um vetor *pode ser modificado* pela função chamada. Isto significa que podemos passar um vetor para uma função e alterar os valores de seus elementos. Isto ocorre porque a passagem de *vetores* para funções é feita de modo especial dito *passagem por referência*.**

7.5 Vetores multidimensionais

- **Vetores podem ter mais de uma dimensão, isto é, mais de um índice de referência.**
- **Podemos ter vetores de duas, três, ou mais dimensões.**
- **Podemos entender um vetor de duas dimensões (por exemplo) associando-o aos dados de um tabela.**

Exemplo 38: Representação de vetor bidimensional c/ tabela.

- **Na tabela a seguir representamos as notas de 5 alunos em 3 provas diferentes (matemática, física e química, por exemplo). O nome `nota` é o nome do conjunto, assim podemos dizer que a nota do 3º aluno na 2ª prova é `6.4` ou representar `nota[2,1] = 6.4`**

nota	0	1	2
0	8.4	7.4	5.7
1	6.9	2.7	4.9
2	4.5	6.4	8.6
3	4.6	8.9	6.3
4	7.2	3.6	7.7

7.5.1 Declaração e inicialização

- **A declaração e inicialização de vetores de mais de uma dimensão é feita de modo semelhante aos vetores unidimensionais.**

- **A sintaxe para declaração de vetores multidimensionais é:**

```
tipo nome[tam_1][tam_2]...[tam_N]={{lista},{lista},...{lista}};
```

- **onde:**
  `tipo` é o tipo dos elementos do vetor.
  `nome` é o nome do vetor.
  `[tam_1][tam_2]...[tam_N]` é o tamanho de cada dimensão do vetor.
  `{{lista},{lista},...{lista}}` são as listas de elementos.

Exemplo 39: Declarar e inicializar vetores multidimensionais

- **No exemplo a seguir, `nota` é um vetor** duas **dimensões `([][])` composto de** 5 **vetores de** 3 **elementos cada.**
- **`tabela` é um vetor de três dimensões `([][][])` composto de 2 vetores de 3 sub-vetores de 2 elementos cada.**

```
float nota[5][3] = {{8.4,7.4,5.7},
                    {6.9,2.7,4.9},
                    {4.5,6.4,8.6},
                    {4.6,8.9,6.3},
                    {7.2,3.6,7.7}};

int tabela[2][3][2] = {{{10,15}, {20,25}, {30,35}},
                       {{40,45}, {50,55}, {60,65}};
```

7.5.2 Passando vetores multidimensionais para funções

- **A sintaxe para *passagem* de vetores multidimensionais para funções é semelhante a passagem de vetores unidimensionais: chamamos a função e passamos o *nome* do vetor, *sem* índices. A única mudança ocorre na declaração de funções que recebem vetores:**
- **Na *declaração* de funções que recebem vetores a sintaxe é:**

```
tipo_f função(tipo_v vetor[tam_1][tam_2]...[tam_n])
{
  ...
}
```

- **Observe que depois do nome do vetor temos os índices com contendo os tamanhos de cada dimensão do vetor.**

Exemplo 40: Passando vetores multimensionais para funções

```
// Na declaração da função:
int max(int vetor[5][7],int N, int M)
{
   ...
}

// Na chamada da função:
void main()
{
     int valor[5][7]; // declaração do vetor
     ...
     med = media(valor, n); // passa vetor p/ função
     ...
}
```

- **Observações:**
  - **Do mesmo modo que vetores unidimensionais, os vetores multidimensionais podem ter seus elementos modificados pela função chamada.**
  - **Os índices dos vetores multidimensionais, também começam em `0`. Por exemplo: `vet[0][0]`, é o primeiro elemento do vetor.**
  - **Embora uma tabela não seja a única maneira de visualizar um vetor bidimensional, podemos entender o *primeiro* índice do vetor como o índice de *linhas* da tabela e o *segundo* índice do vetor como índice das *colunas*.**

8. ENDEREÇOS E PONTEIROS

8.1 CONTEÚDO:

- **O que é um ponteiro.**
- **Como os ponteiros são utilizados.**
- **Como declarar e inicializar ponteiros.**
- **Como usar ponteiros para passar matrizes para funções.**

8.2 VANTAGENS:

- **Coisas que os ponteiros ajudam a fazer melhor;**
- **Coisas que só podem ser feitas usando ponteiros.**

8.3 O QUE É UM PONTEIRO?:

- **Vamos entender o conceito de ponteiros, lembrando como funciona a memória do computador.**

8.4 A MEMÓRIA DO SEU COMPUTADOR:

- **A memória RAM de um PC consiste de milhares de localizações seqüenciais para armazenamento de dados;**
- **Cada localização é identificada por um endereço único. Os endereços enquadram-se numa faixa que varia de 0 até o valor máximo suportado pela memória instalada no computador;**
- **A memória é distribuída entre SO, programas em execução (códigos e dados do programa);**

8.4.1 ARMAZENAMENTO DE VARIÁVEIS:

⇒ **Declaração de uma variável → Compilador reserva uma localização de memória com um endereço único para armazenar essa variável.**

⇒ **Compilador** $\xrightarrow{\text{associa}}$ **endereço com o nome da variável;**

⇒ **Programa** $\xrightarrow{\text{referencia}}$ **a variável pelo nome** $\xrightarrow{\text{acessa}}$ **a localização correspondente na memória.**

⇒ **O endereço da variável na memória está sendo usado, mas para o usuário final é imperceptível.**

Exemplo 41: Armazenamento de variáveis

1000	1001	1002	1003	1004	1005
				100	

↑

rate

⇒ **A variável** *rate* **foi inicializada com o valor 100;**

⇒ **Compilador reservou o endereço 1004 da memória para armazenar a variável e associou o nome rale ao endereço 1004.**

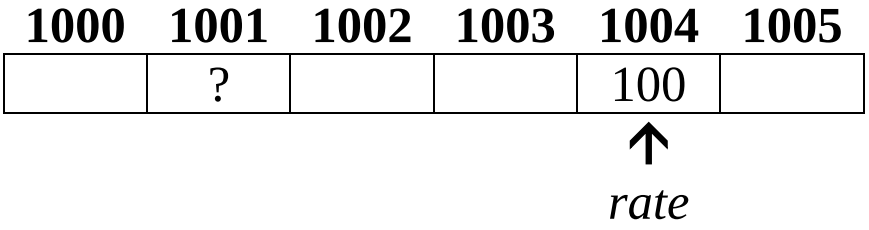
8.5 CRIANDO UM PONTEIRO:

Observe: **o endereço da variável** *rate* **é um número e pode ser tratado como qualquer outro número em C.**

⇒ **Sabendo o endereço de uma variável, pode-se criar uma segunda variável para armazenar o endereço da primeira, basta seguir os seguintes passos:**

1º Passo: declarar uma variável que será usada para conter o endereço de *rate;*
2º Passo: Variável → *p_rate;*

Exemplo 42: Ilustrando a declaração de ponteiros

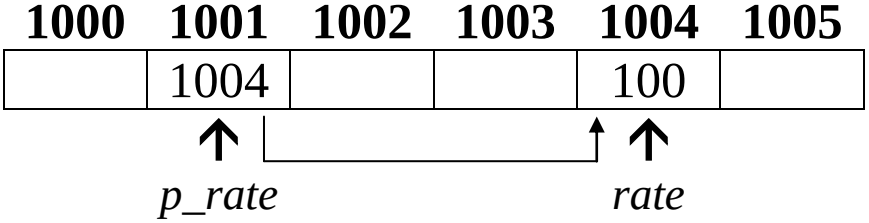


Como armazenar o endereço da variável *rate* **em** *p_rate***?**

8.6 CRIANDO UM PONTEIRO

1º Passo: armazenar endereço de *rate* → *p_rate*

Exemplo 43: Ilustrando a criação de ponteiros



2º Passo: *p_rate* **contém endereço de** *rate,* **podendo indicar a localização na memória onde são armazenados seus dados; ou seja** *p_rate* ***aponta*** **para** *rate,* **ou é um** ***ponteiro*** **para** *rate***.**

Um ponteiro é uma variável que contém o endereço de outra variável.

8.7 PONTEIROS E VARIÁVEIS SIMPLES

⇒A variável-ponteiro do exemplo apontava para uma variável simples (não-matriz).

8.7.1 DECLARANDO PONTEIROS

⇒Um ponteiro deve ser declarado antes **de ser usado;**
⇒A nomenclatura das variáveis-ponteiro segue _as mesmas convenções adotadas para outras varíáveis (seus nomes deiem serúnicos).

Exemplo 44: Como declarar um ponteiro

nometipo *nomeponteiro

⇒*nometipo:* tipo de variável em C que indica o tipo da variável para a qual o ponteiro está apontando.
⇒o asterisco (*) é o operador de indireção, indicando que *nomeponteiro* é um ponteiro para uma variável do tipo *nometipo* e não uma variável desse tipo.
⇒o asterisco (*), diz ao compilador que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado.

Exemplo 45: Mais declarações de ponteiros

char *chl, *ch2 → *ch1* e *ch2* chl e ch2 são ponteiros para variáveis do tipo char;

float *valor, porcentagem → *valor* é um ponteiro para um tipo float, *porcentagem* é uma variável normal do tipo float;

LEMBRETE:

asterisco (*) ≠ multiplicação (*)

8.8 INICIALIZANDO PONTEIROS

- **Nós já vimos como declarar ponteiros, mas o que podemos fazer com eles?**
- **Nada... até agora, a não ser que você não se importe de ter resultados desastrosos em seu programa.**
- **Por exemplo a declaração: char *ch1, declara um ponteiro para um tipo *char*, mas ele ainda aponta para um lugar indefinido;**
- **Este lugar, na pior da hipóteses pode ser uma porção de memória reservada ao SO, podendo levar até ao travamento do micro.**

"Um ponteiro só é útil depois que passa a conter o endereço de uma variável".

Exemplo 46: Operador "endereço de" (&)

ponteiro = &variável;

quando é colocado antes- de um nome de variável,r O operador *endereço de,* retorna o endereço dessa variável.

p_rate = &rate;

atribui o endereço de *rate* a *p_rate*.

8.9 USANDO PONTEIROS

⇒ **Analisando o ponteiro pjate:**

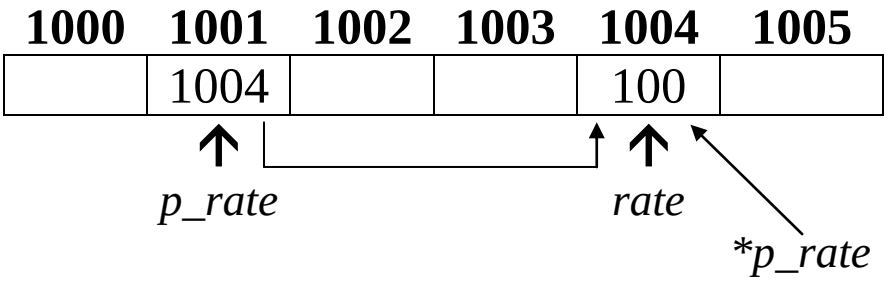
```
.
.
.
int rate;
int *prate;

main()
{
/* inicializa p_rate para apontar para rate */
prate = &rate  /* atribui o endereço de rate a p_rate */

printf(%d, rate);      /* ou */
printf(%d,*p_rate); /* Imprimirão o mesmo valor (100) */
```

OBSERVAÇÃO:

- **Em C, estas duas instruções são equivalentes:**
- **Acessar o conteúdo de uma variável usando seu nome *(acesso direto);***
- **Acessar usando um ponteiro *(acesso indireto)* ou indireção.**



- **nome de um ponteiro precedido pelo operador de indireção (*) refere-se ao valor da variável para a qual ele está apontando.**

```
1: /* Programa p/ demostrar o uso básico de ponteiros */
2:
3: #include <stdio.h>
4: int var = 1;
5: int *ptr;
6:
7: main()
8: {
9:    /* inicializa ptr para apontar para var */
10:    ptr = &var;
11:    printf("\nAcesso direto, var = %d", var);
12:    printf("\nAcesso indireto, var = %d", *ptr);
13:    /* Exibe o endereço da variável de duas maneiras*/
14:    printf("\n\nEndereço de var = %d", &var);
15:    printf("\nEndereço de var = %d", ptr);
16: }
```

Saída do programa:

Acesso direto, var =1
Acesso indireto, var = 1
Endereço de var = endereço para esta variável na memória;
Endereço de var = endereço para esta variável na memória;

9. PONTEIROS – Uso Com Matrizes

9.1 TIPOS DE VARIÁVEIS:

- **Até agora ignoramos o fato de que diferentes tipos de variáveis ocupam direrentes quantidades de memória;**

- **Num PC, uma variável *int* ocupa 2 bytes, uma variável *float* ocupa 4 bytes, e assim por diante;**

- **Cada byte individual da memória tem seu próprio endereço, assim uma variável que use múltiplos bytes também ocupará vários endereços.**

9.1.1 COMO UM PONTEIRO PODE APONTAR PARA UMA VARIÁVEL QUE OCUPE MÚLTIPLOS ENDEREÇOS?

- **O endereço de uma variável é apenas o endereço do seu byte de número mais baixo.**

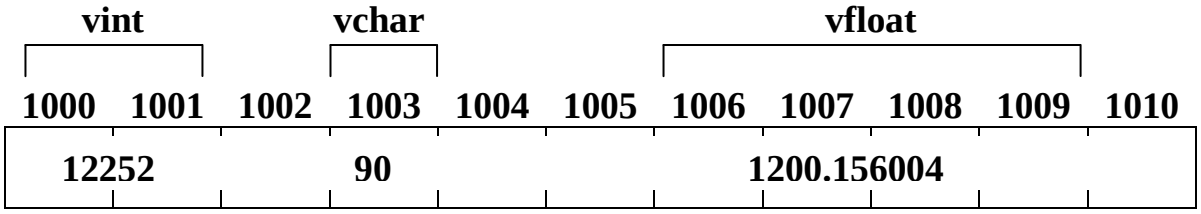
Exemplo 47: Armazenamento de variáveis na memória

- **Observe a declaração e inicialização das variáveis abaixo:**

```
int   vint = 12252;
char  vchar = 90;
float vfloat = 1200.156004;
```

- **Armazenamento na memória dessas variáveis corresponde:**

***int*: 2 bytes**
***char*: 1 bytes**
***float*: 4 bytes**



Diferentes tipos de variáveis numéricas ocupam diferentes quantidades de espaço na memória.

9.1.2 INICIALIZANDO E DECLARANDO OS PONTEIROS PARA AS VARIÁVEIS:

```
/* declarando */
int   *p_vint;
char  *p_vchar;
float *p_vfloat;

/* inicializando */
p_int   = &vint;
p_char  = &vchar;
p_float = &vfloat;
```

- **Cada ponteiro conterá o endereço do primeiro byte ocupado pela variável para o qual está apontando:**

- **Assim, com base nos exemplos anteriores, os ponteiros declarado e inicializados acima conteriam os seguintes endereços:**

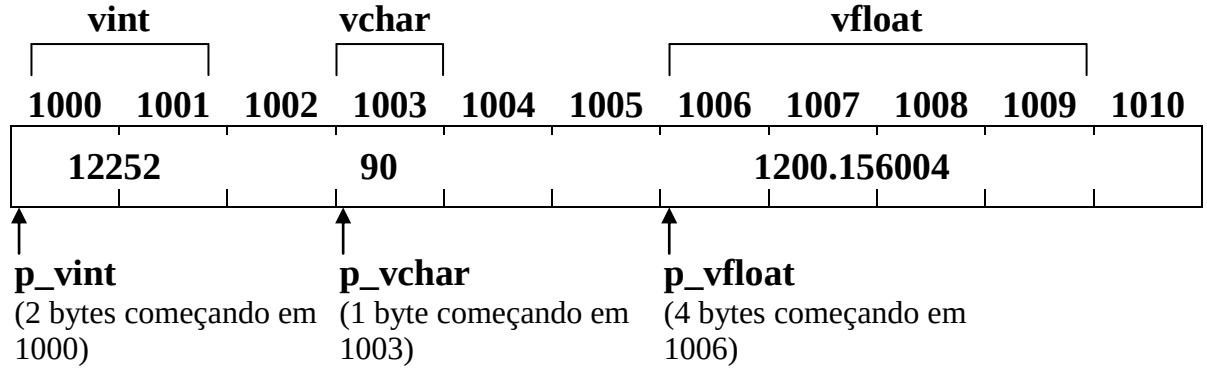
p_vint	p_vchar	p_vfloat
1000	1003	1006

- **o compilador "sabe" que um ponteiro para um tipo *int* aponta para o primeiro de *2 bytes*, um ponteiro para o tipo *float* aponta para o primeiro de *4 bytes*, etc.**

- **Na figura anterior observamos algumas localizações vazias entre as três variáveis. Isso é só para facilitar a visualização; na prática o compilador C armazenaria as três variáveis em localizações adjacentes da memória, sem deixar bytes vazios entre elas.**

9.2 PONTEIROS E MATRIZES:

- **Ponteiros são úteis quando trabalhamos com variáveis simples, mas são ainda mais úteis quando trabalhamos com matrizes.**



9.2.1 O NOME DA MATRIZ COMO UM PONTEIRO:

- **O nome de uma matriz sem colchetes é um ponteiro para o primeiro elemento dessa matriz;**

Exemplo 48: Uso do nome da matriz como ponteiro

- **Se você declarar uma matriz chamada *dados[ ]*, o endereço do primeiro elemento dessa matriz é:**

```
dados
```

ou

```
&dados[0]
```

logo,

`dados == &dados[0]` **é verdadeiro.**

LEMBRETE:

"O nome de uma matriz é um ponteiro para essa matriz."

- **É possível entretanto, declarar uma variável-ponteiro e inicializá-la para apontar para uma matriz. Exemplo:**

```
int array[100], *p_array;
/* intruções adicionais...*/
p_array = array;
```

- **A variável-ponteiro *p_array* é inicializada com o endereço do primeiro elemento da matriz *array[ ]*;**
- **p_array é um ponteiro e pode ser modificado para apontar para outros endereços.**

VANTAGEM DESSE TIPO DE DECLARACÃO:

- **Ao contrário de *array*, *p_array* não precisa necessariamente ficar sempre apontando para o 1º elemento de *array[ ]*;**
- **p_array pode apontar para outros elementos de *array[ ]*.**

9.2.2 ARMAZENAGEM DOS ELEMENTOS DE MATRIZES:

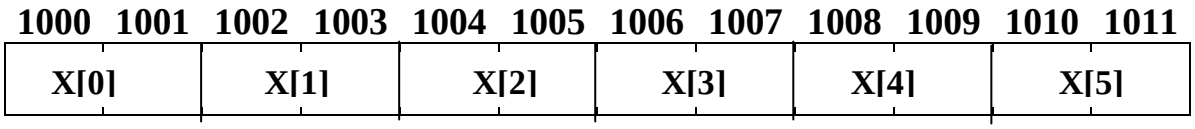
- **Os elementos de uma matriz são armazenados em localizações seqüenciais da memória;**

- **Com o primeiro elemento no endereço mais baixo, os elementos subseqüentes da matriz (aqueles cujo índice é maior que 0), são armazenados em endereços mais altos;**

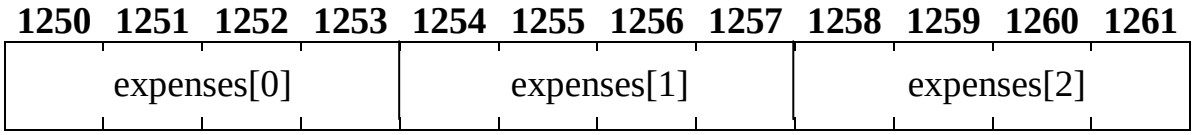
- **O tamanho do intervalo entre cada elemento depende do tipo de dados da matriz (*char*, *int*, *float*, etc.).**

Exemplo 49: Armazenamento elementos de matrizes

int x[6]:



float expenses[3]:



- **Na matriz do tipo *int* cada elemento da matriz está localizado *dois bytes* acima do elemento anterior, e seu endereço também será duas unidades maior que o endereço do elemento anterior.**

- **Na matriz do tipo *float*, cada elemento da matriz está situado *quatro bytes* acima do elemento anterior, e o endereço de cada elemento da matriz será quatro unidades maior que o endereço do elemento que o precede.**

Exemplo 50: Ilustração da armazenagem de matrizes

```
1: x==1000
2: &x[0]==1000
3: &x[1]==1002
4: expenses==1250
5: &expenses[0]==1250
6: &expenses[1]==1254
```

- Quando é usado sem colchetes, **x** aponta para o endereço do primeiro elemento;
- X[0] está no endereço 1000;
- X[1] está no endereço 1002;
  {Na matriz do tipo ***int*** a diferença é de ***dois bytes***}
- As linhas 4, 5 e 6 demonstram que para a matriz do tipo ***float*** (expenses) a diferença é de ***4 bytes***.

9.2.3 COMO PODEMOS ACESSAR OS ELEMENTOS SUCESSIVOS DE UMA MATRIZ USANDO PONTEIROS?

OBSERVE:

- **Para uma matriz do tipo *int*, um ponteiro teria que sofrer um incremento de 2 unidades;**
- **Para uma matriz do tipo *float*, um ponteiro teria que sofrer um incremento de 4 unidades;**
- **Ou seja:**
  - **Para um ponteiro acessar elementos sucessivos de uma matriz de um determinado tipo de dados, deverá, ser incrementado em *sizeof*(tipodedados).**

Exemplo 51: Relação entre endereços e elementos da matriz

```
1:  /* Demonstra a relação entre endereços e elementos */
2:  /* de matrizes de diferentes tipos de dados */
3:  #include <stdio.h>
4:  /* Declara três matrizes e uma variável de contagem */
5:  int i[10], x;
6:  float f[10];
7:  double d[10];
8:  main()
9:  {
10:    /* imprime. o cabeçalho da tabela */
11:    printf("\t\tInteira\tFloat\tDouble");
12:    printf("\n=======================================");
13:    printf("\n=======================================");
14:    /* Imprime endereços de cada elemento da matriz */
15:    for(x=0; x<10; x++)
16:       printf("\nElemento %d:\t%d\t\t%d\t\t%d",x,&i[x],
                  &f[x],&d[x]);
17:    printf("\n=======================================");
18:    printf("\n=======================================");
19:    fflush(stdin);
20:    getchar();
21: }
```

Comentários

- **Nas linhas 5,6,7, as matrizes são criadas;**

- **As linhas 15 e 16 formam um loop *for* que imprime as fileiras da tabela. O número do elemento *x*, é impresso em primeiro lugar, seguido do endereço desse elemento em cada linha das três matrizes.**

	Inteiro	Float	Double
Elemento 0:	**1392**	**1414**	**1454**
Elemento 1:	**1394**	**1418**	**1462**

10. ARITMÉTICA COM PONTEIROS

- **Como acessar os elementos da matriz usando a notação de ponteiros?**

10.1 Operações Aritméticas com Ponteiros:

- **Duas operações aritméticas são mais importantes quando tratamos com ponteiros: *incremento* e *decremento*.**

10.2 Incrementando Ponteiros:

- **Incrementando um ponteiro com o valor 1, a aritmética de ponteiro estabelece que aumentará o valor no ponteiro para que ele passe a apontar para o próximo elemento da matriz;**
- **Aumentará o endereço armazenado na declaração do ponteiro, em um valor correspondente ao tamanho desse tipo de dados.**

Exemplo 52: Incremento de ponteiros

- *ptr_para_int*: **ponteiro que referencia algum elemento do tipo** *int*.
- *ptr_para_int*++: **aumenta o valor de** *ptr_para_int* **com o valor correspondente ao tamanho do tipo** *int* (2 bytes);
- *ptr_para_int*: **passa a apontar para o próximo elemento da matriz do tipo** *int*.

- *ptr_para_float*: **ponteiro que referencia algum elemento do tipo** *float*.
- *ptr_para_float*++: **aumenta o valor de ptr_para_float em** *4 bytes*.
- *ptr_para_float*: **passa a apontar para o próximo elemento da matriz do tipo** *float*.

10.3 Incrementando valores maiores que 1:

- **Somando-se o valor *n* a um ponteiro, a linguagem C incrementa o ponteiro com um valor correspondente a *n* elementos da matriz.**

Exemplo 53: Incremento de valores maiores que 1

```
ptr_para_int += 4;
```

- **aumenta o valor armazenado em** *ptr_para_int* **em** 8 unidades**, para que ele passe a apontar** 4 elementos **mais adiante.**

```
ptr_para_float += 10;
```

- **aumenta o valor armazenado em** *ptr_para_float* **em** 40 unidades**,** *ptr_para_float* **apontará para o elemento situado** 10 posições **à frente.**

10.4 Decrementando Ponteiros:

- **Os conceitos aplicados ao incremento, aplicam-se ao decremento, com o acréscimo de um valor negativo;**
- **Usando-se os operadores + ou -=, a aritmética com ponteiros automaticamente fará os ajustes relativos ao tamanho da matriz.**

Exemplo 54: Decrementando ponteiros

```
ptr_para_int--
```

- **retrocede o valor de** *ptr_para_int* **com o valor correspondente ao tamanho do tipo** *int* **(**2 bytes**);**

Exemplo 55: Usando aritmética de ponteiros c/ matrizes

```
1: /* Demonstra uso da aritmética c/ ponteiros para acessar */
2: /* elementos de uma matriz usando a notação de ponteiros */
3: #include <stdio.h>
4: #define MAX 10
5: /* Declara e inicializa uma matriz do tipo int */
6: int i_array[MAX] = {0,1,2,3,4,5,6,7,8,9};
7: /* Declara e inicializa uma matriz do tipo float */
8: float f_array[MAX] = {.O,.1,.2,.3,.4,.5,.6,.7,.8,.9};
9:
10: /* Declara ponteiro p/ uma matriz int e variável inteira */
11: int *i_ptr,count;
12: /* Declara um ponteiro para uma matriz do tipo float */
13: float *f_ptr;
14:
15: main()
16: {
17:    /* lnicializa os ponteiros */
18:    i_ptr = i_array;
19:    f_ptr = f_array;
20:    /* Imprime os elementos da matriz */
21:    for (count 0; count < MAX; count++)  {
22:       printf("\n%d\t%f",*i_ptr++,*f_ptr++);
23:    }
24:    fflush(stdin);
25:    getchar;
26: }
```

- **nas linhas** 18 **e** 19 **o programa atribui o endereço inicial das duas matrizes aos respectivos ponteiros;**
- **na linha** 21**, a instrução** *for* **usa a variável** count **para contar de** 0 **até** MAX**;**
- **na linha** 22**, a cada ciclo da contagem os dois ponteiros são referenciados indiretamente, e os valores para os quais eles estão apontando são impressos.**
- **o operador de incremento (++) faz com que cada ponteiro aponte para o próximo elemento da matriz.**

Saída do programa

```
0   0.000000
1   0.100000
```

10.5 Outros Tipos de Operações Com Ponteiros:

- **Se *prt1* e *ptr2* estiverem apontando para elementos de uma mesma matriz:**

ptr1 _ ptr2	**Distância entre estes dois elementos**
prt1 < prt2	**Se ptr1 estiver apontando para um elemento de subscrito mais baixo que o de prt2**
==, !=, >, <, >=	**Se ambos estiverem apontando para a mesma matriz**
ptr *= 2;	**"Erro", multiplicação e divisão não fazem sentido quando tratamos com ponteiros**