

Revisão – Linguagem C

Prof. Rafael Fernandes Lopes
rafaelf@lsdi.ufma.br



Introdução

Introdução

- **C** - *Dennis Ritchie* (Laboratórios Bell)
- Inicialmente para máquinas do tipo PDP-1 (com UNIX).
- Depois para os IBM PC e compatíveis (ambientes MS DOS, e MS Windows)

O Standard ANSI-C

- Versão C que segue a norma da *American National Standard Institute (ANSI)*, e da *International Standards Organization (ISO)*.
- Os Compiladores da Borland foram os primeiros a oferecer compatibilidade com esta norma (os compiladores de Turbo C a partir da versão 2.0).

Características

- A linguagem C é muito famosa e muito utilizada:
 - pela concisão de suas instruções;
 - pela facilidade de desenvolvimento de Compiladores C;
 - pela rapidez de execução de programas;
 - e principalmente pelo fato que o poderoso sistema operacional Unix foi escrito em C.

Um Programa Simples

```
#include <stdio.h>

/*
    um programa bem simples
    que imprima: Ate logo.
*/

void main ()
{
    printf("Ate logo. ");
}
```

Exemplo 1

```
#include <stdio.h>                                /* 1 */
#define B 20                                       /* 2 */

int a,s;                                           /* 3 */

int  somar (int x, int y)                          /* 4 */
{
    return (x+y);
}

void  main ()                                       /* 5 */
{
    scanf ("%d", &a) ;                             /* 6 */
    s=somar (a,B) ;                                /* 7 */
    printf ("%d", s) ;                             /* 8 */
}
```

Comentários – Ex1

<code>/* 1 */</code>	inclusão de um arquivo da biblioteca de C que implementa as funções <code>printf</code>, <code>scanf</code>...
<code>/* 2 */</code>	<code>b</code> deve ser substituído por 20
<code>/* 3 */</code>	declaração de duas variáveis inteiras: <code>a</code> e <code>s</code>
<code>/* 4 */</code>	definição de uma função <code>somar</code>
<code>/* 5 */</code>	definição da função principal: <code>main</code>
<code>/* 6 */</code>	leitura do valor de <code>a</code> entrado pelo usuário
<code>/* 7 */</code>	chamada da função <code>somar</code> com argumentos <code>a</code> e <code>b</code>
<code>/* 8 */</code>	impressão do resultado <code>s</code> da soma de <code>a</code> e <code>b</code>

Estrutura de um Programa C

- Um conjunto de funções
- A função principal `main` é obrigatória.
- A função `main` é o ponto de entrada principal do programa.

main

```
void main ()  
{  
    declarações  
  
    instruções 1  
    instruções 2  
    ...  
    instruções n  
}
```

As outras funções

```
Tipo nome (declaração-de-parâmetros)
{
    declarações;

    instruções 1;
    instruções 2;
    ...
    instruções n;
}
```

Considerações sobre as funções

- O tipo da função = tipo do valor que a função retorna
 - Ele pode ser predefinido, ou definido pelo usuário.
- Por *default* o tipo de uma função é `int`.

Compilação

- **Pre-Processadores:**

Transformação lexical do texto do programa.

- **Compiladores:**

Tradução do texto gerado pelo pre-processador e sua transformação em instruções da máquina.

Observação

Geralmente, o pre-processador é considerado como fazendo parte integrante do compilador.

Tipos Básicos

Principais tipos de Dados

- `int` } família dos números inteiros
- `float` }
- `double` } família dos números ponto-flutuantes (pseudo-reais)
- `char` } família dos caracteres

Observação: Não tem o tipo Boolean!

Inteiros

- Podem receber dois atributos:
 - de precisão (para o tamanho)
 - de representação (para o sinal)
- Atributos de precisão:
 - `short int` : representação sobre 1 byte
 - `int` : representação sobre 1 ou 2 palavra(s)
 - `long int` : representação sobre 2 palavras

Inteiros

- **Atributos de representação**
 - **unsigned** : **somente os positivos**
 - **signed** : **positivos e negativos**

Combinação de Atributos - Inteiros

- `unsigned short int` : rep. sobre 8 bits [0, 255]
- `signed short int` : rep. sobre 7 bits [-128, 127]
- `unsigned int` : rep. sobre 16 bits [0, 65535]
- `signed int` : rep. sobre 15 bits [-32768, 32767]
- `unsigned long int` : rep. sobre 32 bits [0, 4294967295]
- `signed long int` : rep. sobre 31 bits [-2147483648, 2147483647]

Os Inteiros

- Em Resumo, temos seis tipos de inteiros:

- `int;`
- `short int;`
- `long int;`

} todos signed

e

- `unsigned int;`
- `unsigned short int;`
- `unsigned long int;`

Pseudo-Reais

(representação da forma: $M * 10^{\text{EXP}}$)

- Os flutuantes podem ser:
 - `float` : representação sobre 7 algarismos
 $[-3.4 * 10^{-38}, 3.4 * 10^{38}]$
 - `double` : representação sobre 15 algarismos
 $[-1.7 * 10^{-308}, 1.7 * 10^{308}]$
 - `long double` : representação sobre 19 algarismos
 $[-3.4 * 10^{-4932}, 3.4 * 10^{4932}]$

Caracteres

- Um caracter é representado por seu código ASCII (código numérico).
- Ele pode ser manipulado como um **inteiro**.
- Um caracter é codificado sobre **um byte**
⇒ podemos representar até 256 caracteres.

Caracteres

- O tipo é: `char`

```
char c,b;
```

```
c = '\65';
```

```
b = 'c';
```

E o tipo String?

- Não existe em C o tipo string propriamente dito.

- Um substituo:

os vetores de caracteres:

```
char nome[20]
```

E o tipo Boolean?

- **Atenção:**

O Boolean não existe em C!

- **Isto é geralmente substituído pelo**

tipo int:

0	:	false
1	:	true

Em sistemas 32 bits

• char	1	-128 a 127
• signed char	1	-128 a 127
• unsigned char	1	0 a 255
• short	2	-32,768 a 32,767
• unsigned short	2	0 a 65,535
• int	4	-2,147,483,648 a 2,147,483,647
• unsigned int	4	0 a 4,294,967,295
• long	4	-2,147,483,648 a 2,147,483,647
• unsigned long	4	0 a 4,294,967,295
• float	4	3.4E+/-38 (7 dígitos)
• double	8	1.7E+/-308 (15 dígitos)
• long double	10	1.2E+/-4932 (19 dígitos)

DECLARAÇÃO DE VARIÁVEIS

Identificadores

- Um identificador é um meio para manipulação da informação
- Um nome que indica uma **variável**, **função**, **tipo de dados**, etc.

Identificadores

- Formado por uma combinação de caracteres alfanuméricos
- Começa por uma letra do alfabeto ou um *sublinhado*, e o resto pode ser qualquer letra do alfabeto, ou qualquer dígito numérico (algarismo), ou um sublinhado.

Exemplos de identificadores

- *Exemplo:*

```
nome_1; _a; Nota; nota;  
  
imposto_de_renda;
```

~~!x;~~ ~~4_nota;~~

- *Atenção:*

Os identificadores Nota e nota por exemplo representam duas variáveis **diferentes.**

Regras para a nomeação

- Como os compiladores ANSI usam variáveis que começam por um sublinhado, melhor então não usá-las.
- De acordo com o standard ANSI-C:
 - somente 32 caracteres podem ser considerados pelos compiladores, os outros são ignorados.
- Não usar as palavras chaves da linguagem.

Palavras Chaves

(são 32 palavras)

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Declarações

`tipo lista-de-nomes-de-variáveis`

- *Exemplo:*

`int i, a;`

`double d1, d2;`

`char c;`

Inicialização de Variáveis

- C não inicializa automaticamente suas variáveis.
- Isto pode ser feito no momento da declaração:

```
int    i=1, j=2, k=somar(4,5);
```

```
short int  l=1; char c='\0';
```

Entradas e Saídas

Saídas

- `printf("Bom dia");`
- `puts("Bom dia");` /* para imprimir um string */
- `printf("Meu nome é: %s\nne tenho %d anos.", nome, idade)`

Entradas

```
gets (s) ;
```

/* leitura de um string */

```
char c;
```

```
c = getchar();
```

```
scanf ("%d", &nome) ;
```

/* para a leitura da var. nome */

Constantes utilizados no `printf`

<code>\n</code>	nova linha
<code>\t</code>	tabulação (horizontal)
<code>\v</code>	tabulação vertical
<code>\b</code>	um espaço para trás
<code>\r</code>	um <i>return</i>
<code>\a</code>	um <i>bip</i>
<code>\\</code>	backslash
<code>\%</code>	o caracter %
<code>\{ ' ' }</code>	um apostrofo
<code>\?</code>	um ponto de interrogação

Conversão de Tipos

- d** notação de decimal
- o** notação octal
- x** notação hexadecimal
- e** notação matemática
- u** notação sem sinal
- c** notação de character
- s** notação de string
- f** notação de flutuante

Formatação

- Cada um desses caracteres de conversão é utilizado depois do % para indicar o formato da saída (da conversão)
- Mas, entre os dois podemos entrar outros argumentos:

Formatação

- justificar a esquerda
- + o sinal sempre aparente
- n** o comprimento mínimo da saída (senão brancos)
- 0n** o comprimento mínimo da saída (senão **0**s esquerda)
- n.m** para separar as partes antes e depois da virgula total de n dígitos, cujo m são depois da virgula.
- l** para indicar um long

Exemplo

```
#include <stdio.h>

int main()
{
    int a; long int b; short int c;
    unsigned int d; char e; float f; double
    g;
    a  = 1023;    b = 2222;    c = 123;
    d = 1234;     e = 'X';
    f = 3.14159;
    g = 3.1415926535898;
    ...
}
```

Exemplo

```
{    printf("a = %d\n", a); a = 1023
    printf("a = %o\n", a); a = 1777
    printf("a = %x\n", a); a = 3ff
    printf("b = %ld\n", b); b = 2222
    printf("c = %d\n", c); c = 123
    printf("d = %u\n", d); d = 1234
    printf("e = %c\n", e); e = X
    printf("f = %f\n", f); f = 3.141590
    printf("g = %f\n", g); g = 3.141593
}
```

Exemplo

```
{  
    printf("\n") ;  
    printf("a = %d\n", a) ;  
    printf("a = %7d\n", a) ;  
    printf("a = %-7d\n", a) ;  
    c = 5 ;  
    d = 8 ;  
    printf("a = %*d\n", c, a) ;  
    printf("a = %*d\n", d, a) ;  
}
```

Exemplo

```
{  
    printf("\n") ;  
    printf("f = %f\n", f) ;  
    printf("f = %12f\n", f) ;  
    printf("f = %12.3f\n", f) ;  
  
    printf("f = %12.5f\n", f) ;  
    printf("f = %12.5f\n", f) ;  
}
```

Exemplo

As saídas são:

`f = 3.141590`

`f = 3.141590`

`f = 3.142`

`f = 3.14159`

`f = 3.14159`

Operações

Atribuição

- ***Exemplo:***

```
i=4 ;
```

```
a=5*2 ;
```

```
d=-b/a ;
```

```
raiz=sqrt (d) ;
```

© 2015 Pearson Education, Inc. or its affiliate(s). All rights reserved. Pearson Education, Inc., publishing as Pearson Benjamin Cummings, 101 Philip Drive, Assinippi Park, New York, NY 10984-2148

$i = (j = k) - 2;$ (caso $k=5$)

$j=5$

$i=3$

- Atribuição múltipla:

a=b=c=8 ;

Conversão de Tipos (1)

- O compilador faz uma conversão automática

```
float r1, r2=3.5; int i1, i2=5;
```

```
char c1, c2='A';
```

```
r1=i2;
```

```
/* r1 ← 5.0 */
```

```
i1=r2;
```

```
/* i1 ← 3 */
```

```
i1=c2;
```

```
/* i1 ← 65 */
```

```
c1=i2;
```

```
/* c1 ← 5 */
```

```
r1=c2;
```

```
/* r1 ← 65.0 */
```

```
c1=r2;
```

```
/* c1 ← 3 */
```

Conversão de Tipos (2)

```
int a = 2;  
float x = 17.1, y = 8.95, z;  
char c;
```

```
c = (char)a + (char)x;  
c = (char)(a + (int)x);  
c = (char)(a + x);  
c = a + x;
```

```
z = (float)((int)x * (int)y);  
z = (float)((int)(x * y));  
z = x * y;
```

Adição/subtração/multiplicação

$\text{Expressão}_1 + \text{expressão}_2$

$\text{Expressão}_1 - \text{expressão}_2$

$\text{Expressão}_1 * \text{expressão}_2$

- As duas expressões são avaliadas, depois a adição/subtração/multiplicação é realizada, e o valor obtido é o valor da expressão.
- Pode ter uma conversão de tipos, depois da avaliação das expressões:

```
float x,y;    int z; x=y+z;
```

Divisão (1)

$\text{Expressão}_1 / \text{expressão}_2$

- Em C, o / designa no mesmo tempo a divisão dos inteiros e dos reais.
- Isto depende então dos valores retornados pelas expressões:
 - Se os dois são inteiros então a divisão é inteira, mas se um deles é real então a divisão é real.

Divisão (2)

- Caso os dois operandos inteiros são positivos, o sistema arredonda o resultado da divisão a zero:

$7 / 2$ retorna 3

- Caso os dois operandos inteiros são de sinais diferentes, o arredondamento depende da implementação mas é geralmente feito a zero:

$-7 / 2$ ou $7 / -2$

retornam -3 ou -4 (mas geralmente -4)

O Módulo %

$\text{Expressão}_1 \% \text{Expressão}_2$

- Retorna o resto da divisão inteira.

`7 % 2 retorna 1`

- Caso um dos operadores é negativo, o sinal do módulo depende da implementação, mas

é geralmente o sinal do primeiro

`7 % 2 retornam 1`

`-7 % 2 retornam -1`

`7 % -2 retornam 1`

Operadores Relacionais

Expressão₁ op-rel Expressão₂

- Onde op-rel é um dos seguintes símbolos:

Operador	Semântica
= =	igual
!=	diferentes
>	superior
>=	superior ou igual
<	inferior
<=	inferior ou igual

O resultado da comparação é um valor lógico:
1 (se a comparação é verificada) e 0 (senão)

Operadores Lógicos

- Para combinar expressões lógicas

Operador	Semântica
&&	e
 	ou
!	negação

- *Exemplos:*

a >= b

(a == 0) || (b != 0)

! ((a == 0) && (b<3))

Operador &

- É o operador de endereçamento (ponteiro):
- **&a** é o endereço da variável **a**
⇒ acesso ao conteúdo da variável **a**.

Incremento/decremento

`x = x + 1;` */* isto incrementa x */*

`x++;` */* isto incrementa x */*

`++x;` */* isto incrementa x */*

`s+=i;` */* s=s+i*/*

`z = y++;` */* z = y e depois y++ */*

`z = ++y;` */* y++ e depois z = y (novo) */*

- ***O decremento é de igual modo***

Regra

$$\begin{array}{c} v = v \text{ operador expressão} \\ \cong \\ v \text{ operador} = \text{expressão} \end{array}$$

Condicional

```
a = (b >= 3.0 ? 2.0 : 10.5) ;
```

⇔

```
if (b >= 3.0)
    a = 2.0;
else
    a = 10.5;
```

```
c = (a > b ? a : b) ;      /* c=maior(a,b) */
c = (a > b ? b : a) ;      /* c=menor(a,b) */
```

Diretivas de Compilação

Diretivas de compilação

- São instruções para o pre-processador.
- Elas não são instruções C, portanto elas não terminam por uma virgula.
- Para diferenciá-las das instruções C, elas são precedidas pelo símbolo especial #

Temos principalmente as seguintes diretivas:

- `#include`
- `#define`
- `#if`, `#else`, `#endif`

Inclusão de Fontes: `#include`

`#include <nome de arquivo>`

- Para a inclusão de um arquivo.
- A busca do arquivo é feita em primeiro no diretório atual e depois no local das bibliotecas:

Em Unix: `/usr/lib/include`

Em DOS: `APPEND` (~ `PATH`, mas para os arquivos de dados e não para os executáveis)

Em Windows: isto é especificado no ambiente

Inclusão de Fontes: `#include`

`#include "nome-de-arquivo"`

- Para a inclusão de um arquivo usuário.
- Neste caso, a busca do arquivo é feita apenas no diretório atual. Senão, pode se especificar o caminho completo do arquivo

Exemplo de inclusão

```
...  
#include "f2.c"  
...
```

f1.c

f2.c

```
for (i=0;i<5;i++)  
    printf("Oi");
```

f1.c

```
...  
for (i=0;i<5;i++)  
    printf("Oi");  
...
```

Arquivos headers (.h)

- Arquivos *headers standards*:
`stdlib`, `stdio.h`, etc.
- Um arquivo header contém um conjunto de declarações de variáveis e funções.
- *Observação*:
Os headers incluídos podem também incluir outros arquivos (mas não pode existir uma referencia mútua)

Substituições: #define

- `#define MAX 60`
- `#define TRUE 1`
- `#define FALSE 0`
- `#define BOOLEAN int`

{

`BOOLEAN a=FALSE;`

`if (a == TRUE) ...; else ...;`

}

(uma convenção: usar os maiúsculos)

Macro Substituições: #define

```
#define maior(A,B)  ( (A) > (B) ? (A) : (B) )
```

```
f()  
{  
    ...  
    maior(i+1,j-1);  
    ...  
}
```

A instrução `maior(i+1,j-1)` será compilada como se nos escrevemos:

```
( (i+1) > (j-1) ? (i+1) : (j-1) )
```

O `const`

- Para declarar variáveis constantes podemos usar a palavra chave `const`

`const int N 20`

Isto proíbe que `N` seja modificado no programa (toda tentativa de modificação vai dar erro)

- *Diferença para o `#define`:*
 - Usando o `const` teremos uma reserva de um espaço na memória. Ele se aplica sobre qualquer tipo de var.
 - O `#define` serve somente para o pre-processador que faz as substituições necessárias antes da compilação.

Estruturas de Controle

Blocos de instruções

```
{  
  declarações  
  instrução_1  
  .....  
  /*um bloco de instruções  
    é delimitado por duas chaves*/  
  instrução_2  
}
```

**Em um bloco pode-se declarar variáveis,
que ficam visíveis somente no bloco.**

Instrução `if...else` (1)

```
if (condição)
    instrução_1;
[else
    instrução_2;]
```

Exemplos:

```
if (a>=b)
    max=a;
else
    max=b;
```

```
if (a!=0)
    x=-b/a;
```


Instrução `if...else` (2)

- Qualquer instrução simples C pode ser substituída por uma instrução composta:

```
if (d>0)
{
    x1=(-b-sqrt(d))/2*a;
    x2=(-b+sqrt(d))/2*a;
}
```

Instrução `if...else` (3)

Observação: o `else` refere-se ao último `if`

```
if (condição_1)
    instrução_1;
else if (condição_2)
    instrução_2;
    else if (condição_3)
        instrução_3;
        else
            instrução_4;
```


Instrução `if...else` (4)

```
int  a=3, b=2, c=1, d=0;
```

```
if (a>b) if (c<d) b=2*a; else a=2*b;
```



senão, tem que escrever:



```
if (a>b)
{
    if (c<d) b=2*a;
}
else a=2*b;
```

Instrução `if...else` (5)

Observação: o `else` refere ao último `if`
senão tem que usar chaves (bloco)

```
if (condição_1)
{
    instrução_1;
    if (condição_2)
        instrução_2;
}
else if (condição_3)
    instrução_3;
```

Observação

- A condição não é obrigatoriamente uma comparação. Pode ser qualquer expressão retornando um valor que pode ser comparado a zero.

```
int b=0;  
if (b)      /* equiv. a: if (b!=0) */  
{  
    ...  
}
```

Instrução switch (1)

escolha múltipla

```
switch (expressão)
{
    case constante_1:
        instruções
        break;
    case constante_2:
        instruções
        break;

    default:
        instruções
}
```

Instrução switch (2)

- Quando as mesmas instruções devem ser executadas no caso de um conjunto de valores:

```
switch (expressão)
{
    case val_1:
    case val_n:      instruções
                    break;
    case val_3:      instruções
                    break;
    default:         instruções
}
}
```

Instruções de Repetição

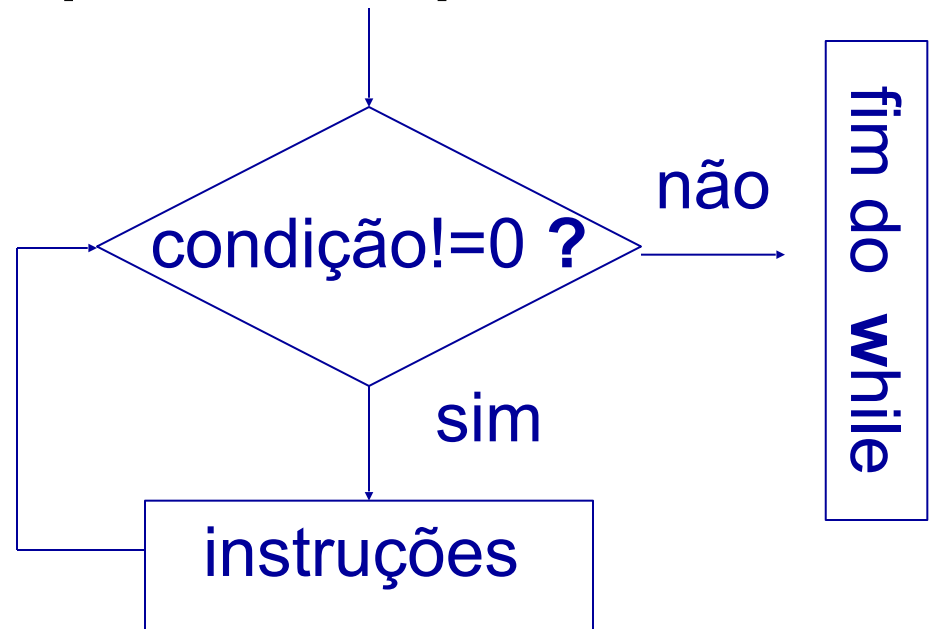
while

```
while (condição)  
    instrução
```

instrução pode ser simples ou composta.

Exemplo:

```
c='s' ;  
while (c!='f')  
{  
    c=getch() ;  
}
```



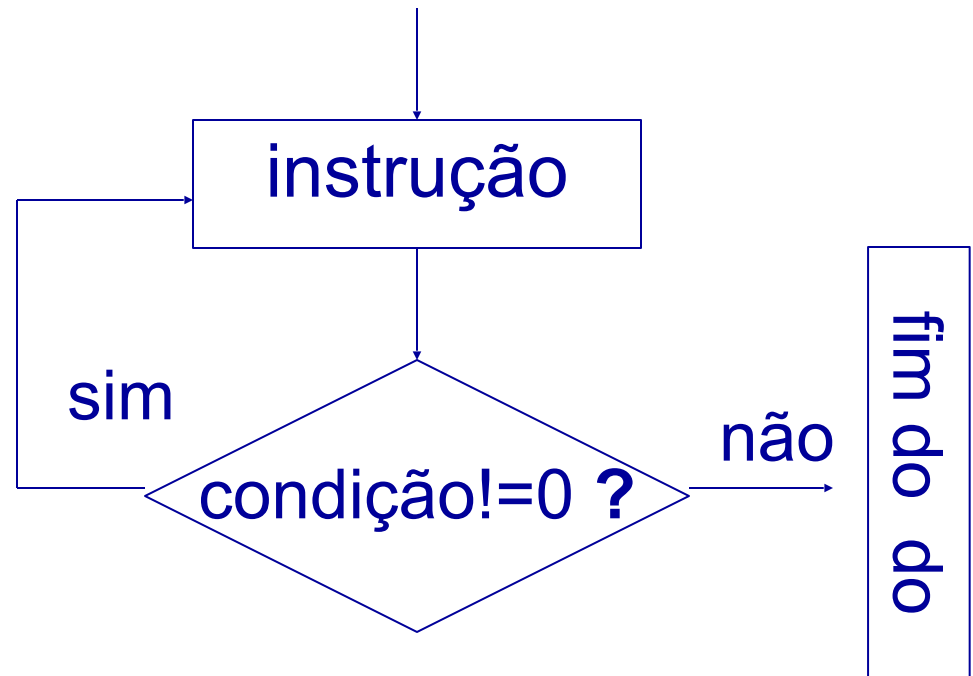
do...while (1)

```
do  
    instrução (ões)  
while (condição)
```

- É o equivalente de `repeat` do Pascal.
- As instruções do laço de repetição são executadas pelo menos uma vez.

do...while (2)

```
do  
    c=getch();  
while (c=='s')
```



for (1)

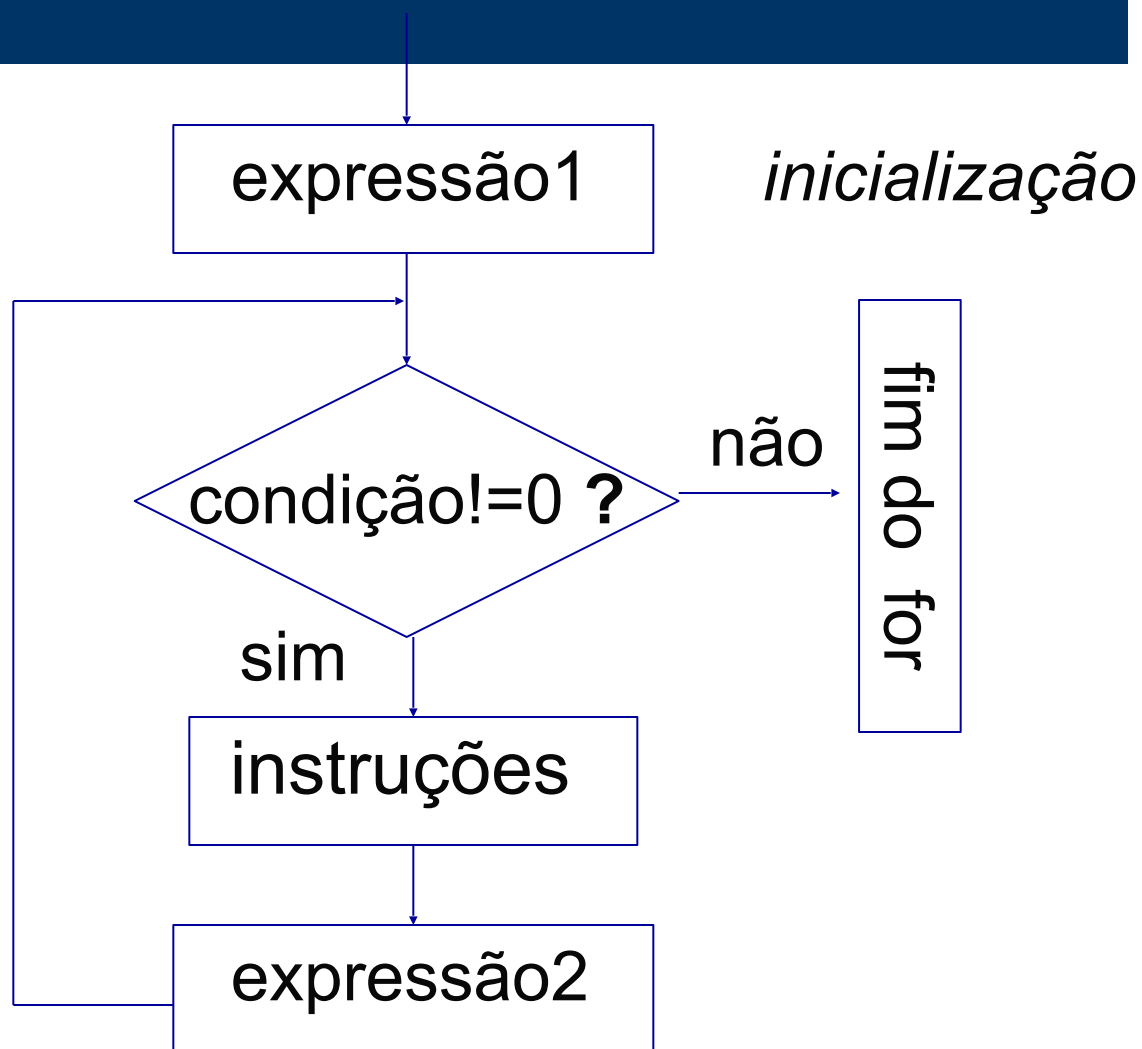
```
for (expressão1, condição, expressão2)
    instrução(ões)
```

- expressão1 é avaliada só uma vez
- depois a condição
 - se condição OK, as instruções são executadas
- depois a expressão2, antes de voltar a avaliar novamente a condição

Exemplo:

```
for (i=1; i<=10; i++)
{
    printf("Vasco da Gama");
}
```

for (2)



for (3)

- A novidade é que as expressões podem ser instruções múltiplas (separadas por virgula)

Exemplo:

```
for (i=1,j=10; i<=j; i++,j--)  
{  
    instruções;  
}
```

break

O **break** permite parar a instrução de repetição (**for**, **do** ou **while**)

Se temos vários níveis, o controle volta à penúltima estrutura de repetição.

```
for (i=0;i<20;i++)  
    if (vet[i]==10) break;
```

continue

- Utilizada dentro de uma instrução `for`, `while` ou `do`
- O `continue` permite parar a iteração atual e passar à iteração seguinte.

Exemplo:

```
for (i=1;i<20;i++)  
{  
    if (vet[i]<0) continue;  
    ...  
}
```


Funções

Definição de uma função

- Além das funções predefinidas nos arquivos *header* da biblioteca (no diretório LIB, usando o `#include`)
- O usuário pode definir novas funções.
- Exemplos de funções predefinidas da biblioteca:
`stdio.h`, `math.h`, `conio.h`,
`stdlib.h` e `dos.h`

Ex. de funções: (stdio.h)

Principalmente :

`puts`

`gets`

`printf`

`scanf`

Ex. de funções: (math.h)

abs	<i>módulo de int.</i>	int	abs(int)
fabs	<i>módulo de real</i>	double	fabs(double)
exp	<i>exponencial</i>	double	exp(double)
ceil	<i>arredondar ao max</i>	double	ceil(double)
floor	<i>arredondar ao min</i>	double	floor(double)
pow	xy	double	pow(double x, double y)
pow10	10^x	double	pow10(double)
hypot	<i>hipotenusa</i>	double	hypot(double, double)
sqrt	<i>raiz quadrada</i>	double	sqrt(double)

Ex. de funções: (stdlib.h)

min	retorna o min	type min(type, type)
max	retorna o max	type max(type, type)
rand	ret. num. alea.	int rand()
exit	ret. nível de erro	void exit(int)
random	ret. num. alea de 0 a n-1	int random(int n)
randomize	inicializa o gen. al.	int randomize()

precisa também de <time.h>

Definição de uma função

```
tipo identificador ([lista de id.])  
[lista de declarações 1]  
  
{  
    [lista de declarações 2]  
    lista de instruções  
  
}
```

Semântica

tipo: é o tipo do valor devolvido pela função

identificador: nome da função

lista dos identificadores: os parâmetros formais

lista de declarações 1: é a lista de declaração dos parâmetros formais

lista de declarações 2: declaração das variáveis locais a função

lista de instruções: são as instruções a executar quando a função é chamada.

Exemplo

```
float media (a,b)
float a,b;      /* declaração dos
                  parâmetros formais */
{
    float resultado; /* var. locais */
    resultado=(a+b)/2;

    return (resultado); /* retornar o
                          res. ao remetente */
}
```


Os parâmetros formais - opcionais

```
double pi () /* não temos param.  
    formais */  
  
{      /* não temos var. locais */  
  
    return (3.14159);  
  
}
```

Observação

- A prototipagem da função pode ser feita de uma vez:

```
int min (int a, int b)
{
    if (a<b)
        return (a) ;
    else
        return (b) ;
}
```

Chamada de uma função

`identificador (lista de expressões)`

As expressões são avaliadas, depois passadas como parâmetros efetivos à função de nome `identificador`.

```
{  
  
  int m_1, m_2;    float p, d=4.5;  
  
  m_1 = max (4,3) ;      m_2 = max (6*2-3,10) ;  
  p = d * pi () ;  
}
```

Procedimentos

- Funções que não retornam valores: usando o tipo especial `void`

```
void linha ()  
{  
    printf("-----\n");  
}
```

Observação: não temos aqui a instrução `return`
(o tipo especial `void` não existia nas primeiras versões da linguagem)

Omissão do tipo da função!

- C considera por padrão o tipo `int`.

```
somar (int a, int b)
{
    return (a+b) ;
}
```

- Melhor não usar esta possibilidade.

Passagem dos Parâmetros

```
int somar (int x, int y)
{
    return (x+y) ;
}

void main ()
{
    int a=8, b=5, s;
    s = somar(a,b) ;    /*os parâmetros efetivos*/
}
```

Passagem por Valor

Na chamada temos passagem de parâmetros.

Mas, as modificações dos parâmetros *formais*

não afetam os parâmetros *efetivos*.

Passagem por Referência (ou por endereço)

- Usar o operador de endereçamento &

```
void somar (int x, int y, int * som)
{
    *som = x+y;
}

void main ()
{
    int a=5, b=6, s;
    somar(a,b, &s);
    printf("%d + %d = %d",a,b,s);
}
```


Outro uso do `void`

```
void f (void)
{
    ...
}
```

é para indicar que a função não tem parâmetros

Declaração de Funções

- Uma função F é conhecida implicitamente por uma outra função G se elas são definidas no mesmo arquivo, e que F é definida antes de G .
- Fora desse caso, e para um controle de tipo e um bom *link*, é preciso **declarar** as funções antes de usar.

Exemplo

```
void    main    (void)
{
    int maior (int, int);
    maior (2,8);
}

...

int maior (int x, int y)
{
    return (x>y?x:y);
}

/*aí main pode ser definida antes*/
```

Dois formatos para a declaração

```
int maior ();
```

} 1ª

```
int maior (int x, int y);
```

} 2ª

**Mas, melhor usar o segundo formato
que é mais completo**

Funções Iterativas

exemplo do fatorial

```
long int fat (long int n)
{
    long int i,res=1;

    for (i=1;i<=n;i++)
        res=res*i;

    return (res) ;
}
```

Funções Recursivas

exemplo do fatorial

```
long int fat (long int n)
{
    if (n == 1)
        return 1;
    else
        return (n*fat(n-1)) ;
}
```

Exercícios

- Escreva as funções `real_dólar` e `dólar_real` de conversões Real-Dólar e vice versa.
- Escreva as versões recursiva e iterativa da função `soma` que retorna a soma dos n primeiros termos:

$$S(n) = 1 + 1/2 + 1/3 + \dots + 1/(n-1) + 1/n$$

- Escreva a função $S(n)$ tal que:

$$S(n) = 1/n - 2/(n-1) + 3/(n-2) + \dots - (n-1)/2 + n/1$$

Matrizes

Matrizes

- O objetivo da estrutura de matriz é **agrupar** um conjunto de **informações** de mesmo tipo em um **mesmo nome**.
- Uma tabela pode ser **multidimensional** ou **unidimensional** (**vetor**).

Declaração

```
float  vet[10];
```

```
long int  v1[8], v2[15];
```

```
/* v1 é um vetor de 8 long int  
   e v2 é um vetor de 15 long int */
```

- **Os elementos são indexados de 0 a N-1**

Dimensão

- Na prática, é recomendado definir sempre uma constante que indica o número de elementos:

```
#define N 60  
short int v[N] ;  
  
/* declara um vetor de 60  
   inteiros indexado de 0 a 59 */
```

Acesso aos elementos

Sintaxe:

nome-variável [expressão]

expressão deve retornar um inteiro (o indexador)

Exemplos:

```
vet[0]=6;
```

```
vet[4+1]=-2;
```

```
x=3*vet[2*a-b];
```

Inicialização

```
#define N 4  
int v[N]={1,2,3,4};
```

- Inicialização de somente uma parte do vetor:

```
#define N 10  
int v[N]={1,2}; //o resto será zerado
```

- Inicialização pelo mesmo valor:

```
for (i=0;i<=9;i++) v[i]=2;
```

- Inicialização pelo usuário:

```
for (i=0;i<=9;i++)  
    scanf ("%d", &v[i]);
```

Matrizes

Declaração:

```
int mat [3][4];    /* matriz bidimensional  
de 3 linhas 4 colunas */
```

Inicialização:

```
int mat [3][4] =  
  
    {  
        {5, 6, 8, 1},  
        {4, 3, 0, 9},  
        {12, 7, 4, 8},  
    }
```

Exemplo

/ Declaração: */*

```
#define L 4;  
#define C 3;  
int mat[L][C];
```

/ leitura: */*

```
for (i=0;i<=L;i++)  
    for (j=0;j<=C;j++)  
    {  
        printf("digite o elemento [%d,%d]: ",i,j);  
        scanf("%d",&mat[i][j]);  
    }
```

Observação 1

```
for (i=0,j=0;i<L && j<C;i++,j++)
{
    printf("digite o elemento [%d,%d]: ",i,j);
    scanf("%d",&mat[i][j]);
}
```

não é a mesma coisa que:

```
for (i=0;i<=L;i++)
    for (j=0;j<=C;j++)
    {
        printf("digite o elemento [%d,%d]: ",i,j);
        scanf("%d",&mat[i][j]);
    }
```


Observação 2

```
int  sum (int n)
{
    int res=0;
    for (;n>0;n--) /*n é inicializada na chamada*/
        res=res+n;
    return (res);
}
```

chamada: sum(5)

O que faz este código?

Exercícios

- **A.1** Escreva o procedimento `ini_num_dias` que inicializa um vetor `num_dias[12]` que indica para cada mês do ano seu número de dias: (`num_dias[i]` indica o número de dias do mês `i` do ano), sabendo que:
Se `i=2` então `num_dias=28`;
Se (`i` par e `i<=7`) ou (se `i` impar e `i>7`) então `num_dias=30` Senão `num_dias=31`.
- **A.2** Escreva o procedimento `imp_num_dias` que imprima os números de dias de todos os meses do ano.
- **B.** Escreva a função `ordenar` que ordena um vetor.
- **C.** Escreva a função `palindromo` que determina se um vetor é um palindromo.

Tipos Enumerados

enum

- A enumeração permite de agrupar um conjunto de constantes (compartilhando a mesma semântica)

- *exemplos:*

```
enum dias {Domingo, Segunda, Terça,  
Quarta, Quinta, Sexta, Sábado};
```

- *declaração:*

```
dias d1,d2=Quinta;
```

- enum defini um novo tipo cujo os elementos são numerados automaticamente de pelo compilador: 0 1 2 ...

Exemplo

```
#include <stdio.h>

enum dias {Segunda, Terça, Quarta, Quinta,
           Sexta, Sábado, Domingo} d;

    // d é uma variável declarada de tipo dias

void main (void)
{
    // dias d; é uma outra maneira de declarar
    for(d = Segunda ; d < Domingo ; d++)
        printf("O código do dia é: %d\n", d);
}
```

Vai imprimir os valores dos dias: 0, 1 até 6.

enum

- Essa numeração permite comparar os elementos do tipo: `if (d1<=d2) ...`

- Portanto podemos mudar essa numeração:

```
enum boolean {true=1,false=0};
```

- Quando um item não é numerado ele pega o valor de seu precedente:

```
enum temperatura  
    {baixa=2,media=4,alta};
```

Strings

String ~ Vetor

- Um string é um conjunto de caracteres.
- Em C, um string é uma estrutura equivalente à estrutura de vetor,
- A única diferença é que o string termina sempre pelo caractere `'\0'`
Isto para facilitar o tratamento dos *strings*
(para poder detectar o fim do string)

Declaração

- O tipo *string* não existe em C,
- Portanto existe duas maneiras para simular este tipo de variáveis:
 - Como um **vetor de char**, ou
 - Como um **ponteiro sobre uma zona de chars**

Como Vetor de Caracteres

- *Declaração:*

```
#define N 20
```

```
char ch [N];
```

- Os *strings* declarados como vetor de *chars* têm um tamanho limite fixo (o tamanho do vetor).
- Se o tamanho do string é menor do que o tamanho do vetor, o compilador C completa pelo caractere especial '`\0`' para indicar o fim do string.

Inicialização de Vetor de Caracteres

- A inicialização de um vetor de char pode ser feita, no momento da declaração, de duas maneiras:

1. Atribuindo um conjunto de caracteres:

```
char ch [20]={ 'e' , 'x' , 'e' , 'm' , 'p' , 'l' , 'o' } ;
```

(como é normalmente feito para inicializar qualquer tipo de vetor).

2. Atribuindo um string (é mais prática):

```
char ch[20]= "exemplo";
```

O tamanho pode ou não estar especificado:

```
char nome[]="este tem 23 caracteres";
```

Acesso aos Elementos

- É feito de uma maneira normal, como para qualquer tipo de vetor:

```
#define N 3
```

```
char ch [N]={ 'O' , 'i' } ;
```

```
ch[0]= 'H' /* refere ao 1º caractere */
```

```
ch[1]      /* refere ao 2º caractere */
```

```
...
```

```
ch[N-1]    /* refere ao Nésimo caractere */
```

O	i	\0
---	---	----



H	i	\0
---	---	----

Como Ponteiro sobre Caracteres

- A manipulação dos *strings* como vetores de caracteres pode aparecer como pouco prática.

Portanto, podemos criar *strings* de tamanho dinâmico usando um ponteiro sobre um char:

```
char * ch = "exemplo";
```

- Contrariamente à outra maneira, a reserva do espaço memória não é feita no momento da declaração, mas dinamicamente no momento da atribuição.

Inicialização e Atribuição

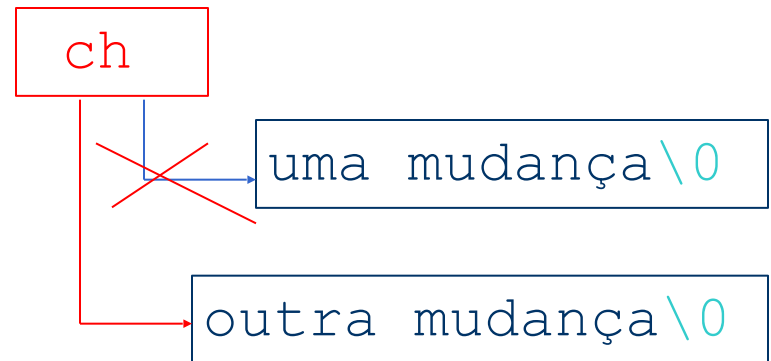
A *Inicialização* é feita diretamente por uma string:

```
char * ch = "exemplo";
```

C completa o string pelo caracter '`\0`' indicando o fim do *string*.

A *Atribuição* também é direta (contrariamente ao vetor de char):

```
ch = "uma mudança";  
ch = "outra mudança";
```



Isto não é uma copia mas uma atribuição de ponteiros.

Manipulação de *Strings*

As funções de manipulação de *strings* são definidas no arquivo da biblioteca `string.h`


Temos principalmente as seguintes funções:

```
strcpy,  strncpy  
strlen  
strcat,  strncat  
strcmp,  strncmp  
strchr,  strrchr
```

Strcpy/strncpy

- A função especial `strcpy` permite atribuir um valor texto a uma variável de tipo texto:
- O `strcpy` apresenta dois formatos de uso:

```
strcpy(string1, string2);  
strncpy(string1, string2, N);
```



exemplo:

```
char *ch1="boa", *ch2="noite";  
strcpy(ch1, "isto é um exemplo");  
strncpy(ch2, ch1, 4); /* ch2 vai pegar "isto" */
```


strlen

strlen permite retornar o tamanho de um string: número de chars que compõem o string (o ' \0' não faz parte)

exemplo:

```
int a;  
char * nome;  
strcpy(nome, "brasil");  
a= strlen(nome); /* ⇒ a=6 */
```

strcat/strncat

`strcat` se aplica sobre dois *strings* e retorna um ponteiro sobre a concatenação dos dois.

exemplo:

```
char      *ch1="boa", *ch2="noite",
          *ch3, *ch4;
ch3=strcat(ch1,ch2);

ch4=strncat(ch1,ch2,3);
printf("%s  %s",ch3,ch4);

/* vai imprimir boanoite boanoi*/
```

strcmp/strncmp

Lembramos que as letras são ordenadas dando seu código: 'a' < ... 'z' < 'A' ... < 'Z'

strcmp compara dois *strings* **s1** e **s2** e

retorna um valor:

negativo	se $s1 < s2$
0	se $s1 == s2$
positivo	se $s1 > s2$

exemplo:

```
char    *ch1="boa tarde",
        *ch2="boa noite";
int a,b;
a=strcmp(ch1,ch2);
b=strncmp(ch1,ch2,4);

/* a vai receber um valor >0 e b 0*/
```

strchr/strrchr

`strchr` procura por um caractere em um *string* e retorna um ponteiro sobre sua última ocorrência, senão retorna `null`.

exemplo:

```
char ch[]="informática";
char *pc, c='f';
pc=strchr(ch,c);
if (pc) /* i.e. if pc!=null */
    printf("%d", *pc);
else
    printf("Caractere inexistente");
```

○ `strrchr` faz a busca no senso inverso.

toupper/tolower

`toupper` converte um caractere minúsculo em maiúsculo.

`tolower` faz o contrário.

```
# include <ctype.h>
```

```
char c='a';
```

```
c=toupper(c);      /* c= 'A'          */
```

```
c=toupper(c);      /* c já esta = 'A'  */
```

```
c=tolower(c);      /* c volta a ser 'a' */
```

Exercícios 1

A. Defina a função `ocorrência` que retorna o número de ocorrências de um caractere em um *string*.

B.1 Defina a função `tamanho1` que pega como parâmetro um vetor de caracteres e retorna seu comprimento.

B.2 Defina `tamanho2` que implementa a mesma função mas que pega como parâmetro um ponteiro sobre uma zona de caracteres.

B.3 Defina o `main` que chama essas duas funções.

```
int tamanho1 (char s[])    /* com um vetor */
{
    int i=0;
    while (s[i])    /* equiv. while (s[i] != '\0') */
        i++;
    return (i);
}
```

```
int tamanho2 (char * s) /* com os ponteiros */
{
    int i=0;
    while (*s)    /* equiv. while (*s != '\0') */
        {i++; s++;}
    return (i);
}
```

```
void main (void)
{
    char ch[]="São Luis";
    int a,b;

    a=tamanho1(ch);           /* a= 8*/
    b=tamanho2(ch);           /* b= 8*/

    printf("O tamanho de %s é: %d\n",ch,a);
    printf("O tamanho de %s é: %d\n",ch,b);
}
```


Exercícios 2: *Criptografia Simples*

1. Defina as funções `Criptar` e `Decriptar` que codificam e decodificam um caractere aplicando o seguinte algoritmo de criptografia:
 - Um caractere é substituído por um outro aplicando um shift de 3 (por exemplo a seria substituído por D).
 - Apenas os caracteres do alfabeto são codificados.
 - Os maiúsculos passam a ser minúsculos e vice-versa.
2. Defina a função `main` que leia e codifica ou decodifica uma mensagem usando as funções definidas acima.

Tipos usuários

typedef

Podemos definir novos tipos usando o typedef:

```
typedef <tipo> <sinônimo>
```

- *exemplo:*

```
typedef float largura;  
typedef float comprimento;  
  
largura l;  
comprimento c=2.5;  
l=c;          /* ⇒ warning */
```

`typedef e struct`

**O `typedef` se usa mais com o tipo `struct`
(as estruturas)**

Estruturas

Declaração: método 1

```
struct cliente {  
    int cpf;  
    char nome [20];  
    char endereco[60];  
};
```

```
struct cliente c1,c2,c3;
```

Declaração: método 2

Podemos criar estruturas sem nome:

```
struct {  
    int    cpf;  
    char  nome  [20];  
    char  endereco[60];  
}  c1, c2;
```

Problema:

para criar uma outra variável de mesmo tipo em outro lugar do programa é preciso rescrever tudo.

Declaração: método 3

Podemos, no mesmo tempo, dar um nome à estrutura e declarar as variáveis:

```
struct cliente {  
    int    cpf;  
    char  nome [20];  
    char  endereco[60];  
}    c1, c2;
```

```
struct cliente c3;
```


Declaração: método 4

Definindo um tipo estrutura

```
typedef struct {  
    int cpf;  
    char nome [20];  
    char endereco[60];  
} Cliente;
```

```
Cliente    c1,c2={28400,"Maria","Rua  
Liberdade, N. 140"};
```

Acesso aos Campos

Usando o operador de seleção: . (o ponto)

`nome-estrutura.nome-do-campo`

exemplo:

`c2.nome` retorna o campo nome da
estruturas c1: Maria

`c2.nome="Jeane"` Muda o conteúdo de nome

Exemplo

```
void imprimir_cliente (Cliente c)
{
    printf("cpf: %d\n%s\n%s",
           c.cpf, c.nome, c.endereco);
}
```

chamada da função:

```
imprimir_cliente (c2);
```

Combinação de Estruturas

- Podemos definir estruturas de estruturas, estruturas de vetores, vetores de estruturas....
- *exemplo*

```
typedef struct {int num; char * rua;  
               char * bairro; int cep} End;
```

```
typedef struct {int cpf; char * nome;  
               End endereco} Pessoa;
```

Vetores de Estruturas

- Podemos criar um vetor de estruturas (dois métodos):

exemplo:

```
struct cliente vet[100]; /* seguindo o mét. 1*/  
Cliente vet[100];      /*seguindo o mét. 4*/  
declara um vetor de 100 clientes.
```

Referencia aos elementos:

Para referenciar o nome do *i*ésimo cliente do vetor:

```
vet[i].nome
```

Exercício

Escreve um programa C que:

- Declara um vetor de alunos: *turma* (um aluno é definido pelo *cpd*, *nome*, *três notas*, *média*, e *conceito*)
- Preenche esse vetor (campos: *cpd*, *nome*, e *notas*).
- Preenche os campos *média* e *conceito* (*bom* se $média \leq 8$, *regular* se ≥ 7 e *ruim* senão);
- Define a função que imprime todos o elementos do vetor.
- Define a função que pesquisa um elemento do vetor (pesquisa pelo *nome*, pelo *cpd*, ou pelo *conceito*).

Unões de Tipos

Objetivo

- Todas variáveis que nos vimos até agora possuam um único tipo.
- As vezes é interessante atribuir vários tipos a uma variável (uma mesma zona memória).
- Isto pode ser feito através do mecanismo de uniões de tipos usando a palavra chave `union`.
- Portanto, uma variável teria, a um dado instante, um único tipo, porém pode mudar.

Declaração

Exemplo:

/ declaração de um tipo que une os inteiros e os reais */*

```
typedef union {  
    int i;  
    float f;  
} número;
```

numero n;

- Podemos então escrever:

n.i=20; / para atribuir um inteiro */*

n.f=3.14159; / para atribuir um real */*

Observação

- A declaração é parecida às estruturas, mas nas uniões somente um campo é atribuído um valor.
- O problema é que nos não podemos saber a um dado instante do programa, qual é o tipo do atual valor da variável.
- Por isso que na prática a união é associada a um indicador de tipo (o tipo atual) e os dois são combinados em uma estrutura:

Utilização Prática das Uniões

```
#define INTEIRO 0
#define REAL 1

typedef struct
{
    int tipo_var;
    union
    {
        int i;
        float f;
    } número;
} aritmética;
```

Utilização Prática das Uniões

```
/* declaração */  
aritmética  a1, a2;  
  
a1.tipo_var=INTEIRO;  
a1.tipo_var=REAL  
  
a1.número.i=10;  
a1.número.i=10;
```

Facilitar o acesso aos campos

O acesso aos campos da união dentro da estrutura não é muito prático: `a1.número.i=10;`

Isto pode ser resolvido usando as substituições e o define:

```
#define I número.i;
```

```
#define R número.f;
```

Agora podemos escrever:

```
a1.I=10; ou a2.R=8.5;
```