UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

BREVE INTRODUÇÃO À LINGUAGEM C

(PARA PROGRAMADORES MÓDULA-2 OU PASCAL)

Francisco Rapchan
rapchan@writeme.com

Capítulo 1 INTRODUÇÃO

O objetivo deste texto é dar informações gerais sobre a linguagem C para pessoas que já conhecem alguma linguagem procedural de programação tal como Modula-2 (preferencialmente) ou Pascal.

Originalmente este texto foi escrito para servir a um curso de introdução à linguagem C para alunos das disciplinas Estruturas de Dados e Estruturas de Informação. A idéia original era que os alunos com formação em Modula-2 pudessem acompanhar os exemplos do livro *Desenho y Manejo de Estructuras de Dados En C* usado como texto básico daquela disciplina.

1.1 BREVE HISTÓRIA DO C

A linguagem C é uma linguagem de alto nível de uso genérico. Foi desenvolvida por programadores para programadores tendo como característica a flexibilidade e a portabilidade.

O C é uma linguagem que nasceu juntamente com o advento da teoria de linguagem estruturada e do computador pessoal. Assim tornou-se rapidamente uma linguagem popular entre os programadores.

O C foi usado para desenvolver o sistema operacional UNIX, e hoje esta sendo usada para como base para desenvolver novas linguagens, entre elas a linguagem C++ e Java.

Alguns marcos históricos são:

- 1969 - Desenvolvimento do UNIX (num PDP 7 em linguagem Assembly)
- 1969 - Desenvolvimento da linguagem BCPL (próxima do Assembly)
- 1970 - Denis Ritchie desenha uma linguagem a partir do BCPL nos laboratórios da Bell Telephones, Inc. Chama a linguagem de B.
- 1971 - Primeiro desenvolvimento da linguagem C, sucessora do B (o C é a 2ª letra de BCPL)
- 1973 - O sistema operacional UNIX é reescrito em linguagem C
- 1978 - Primeira edição do livro The C Programming Language, Kernighan & Ritchie
- 1980 - A linguagem é padronizada pelo American National Standard Institute: surge o ANSI C.
- 1992 - O C se torna ponto de concordância entre teóricos do desenvolvimento da teoria de Object Oriented Programming (programação orientada a objetos): surge o C++.

Capítulo 2 INTRODUÇÃO À LINGUAGEM C

Neste capítulo veremos uma breve introdução à linguagem C. Ao final estaremos aptos a implementar pequenos programas nesta linguagem.

2.1 COMPILANDO PROGRAMAS EM C

A criação dos programas fonte em linguagem C faz-se com o auxílio de um editor de texto genérico, ou específico de um ambiente de desenvolvimento. Em geral, os arquivos de texto produzidos deverão ter a extensão **.c**, para poderem ser reconhecidos automaticamente pelo compilador como sendo arquivos contendo código fonte em C. Obviamente o conteúdo dos arquivos deverá verificar rigorosamente a sintaxe da linguagem C.

A compilação dos programas em C faz-se através da invocação de um compilador (p. ex. no UNIX, o comando cc ou o gcc). O comando de compilação deverá ser seguido pelo nome do arquivo que contém o código fonte (geralmente com a extensão .c). É também comum colocar como parâmetros de chamada do compilador, várias opções de compilação (p. ex. no UNIX, a indicação do nome do arquivo executável com o resultado da compilação). Assim uma compilação básica poderia ser executada no UNIX através do comando:

```
cc program.c
```

onde `program.c` é o nome do arquivo contendo o código fonte.

Se existirem erros de sintaxe no código fonte, o compilador e indicará a sua localização junto com uma breve descrição do erro. Erros na lógica do programa apenas poderão ser detectados durante a execução do mesmo. Se o programa não contiver erros de sintaxe o compilador produzirá código executável. Tratando-se de um programa completo o código executável é colocado, por defeito, num arquivo chamado a.out (isto no UNIX). Se se pretender colocar o resultado da compilação noutro arquivo deverá utilizar-se a opção -o:

```
cc -o program program.c
```

Neste caso o código executável é colocado no arquivo **program** que é já criado com os necessários direitos de execução (no UNIX).

2.2 PRIMEIROS PROGRAMAS EM C

Serão mostrados abaixo alguns programas em C. O objetivo aqui é discutir alguns aspectos bem simples da linguagem C.

2.2.1 OLÁ MUNDO

Os programas abaixo imprimem o texto **`Ola Mundo!`** na tela. O primeiro é em C e o segundo, seu equivalente, em Modula-2.

```
#include <stdio.h>   /* inclui as principais funções da linguagem */
int mensagem ()
{
   printf ("Ola Mundo!\n");  /* exibe Ola Mundo! na tela */
   return 0;
}

int main ()          /* equivale ao BEGIN do MODULE ola do Modula-2 */
{
   mensagem();       /* executa o procedimento mensagem */
   printf ("Eu estou vivo!\n");
   return 0;
}
```

Em Modula-2 este programa ficaria assim:

```
MODULE ola;

FROM IO IMPORT wrStr, wrLn; (* inclui as funções wrStr e wrLn *)

PROCEDURE mensagem;
BEGIN
  WrStr (“Ola Mundo!”);     (* exibe Ola! na tela *)
  WrLn;                     (* Salta uma linha *)
END mensagem;

BEGIN             (* equivale ao main do C *)
  mensagem;                 (* executa o procedimento mensagem *)
  WrStr (“Eu estou vivo!”); (* Exibe Eu estou vivo! *)
  WrLn;                     (* Salta uma linha *)
END ola.
```

Os principais pontos entre este dois programas são:

- Os textos entre ***/**** ****/*** são comentários em C. Em Modula-2 usa-se **(*** ***)**. A maioria dos compiladores C também aceita ***//*** como indicativo de comentário.
- Não há nada parecido em C com o **MODULE** do Modula-2 para definir um nome para o módulo. Em C o nome do módulo é o próprio nome do arquivo.
- Em C usamos o **#include** para incluir bibliotecas assim como no Modula-2 usamos o **FROM**. Em C usamos o **#include** <stdio.h> para incluir a biblioteca padrão de input/output. No Modula-2 usamos o **FROM** IO.
- Em C o programador não define quais funções serão incluídas. Não há em C algo parecido com o IMPORT.
- Em Modula-2 usa-se a palavra **PROCEDURE** para definir procedimentos e funções. Em C não se usa uma palavra para definir uma função ou procedimento.
- A função **printf** é usada em C para mostrar dados na tela. É uma função muito poderosa e equivale às funções de **wr**... do Modula-2 (wrStr, wrCard, wrLn, wr...).
- Em C, o parêntese **{** equivale aproximadamente ao **BEGIN** do Modula-2 e **}** equivale aproximadamente ao **END**.
- A função **main** do C é a primeira a ser executada pelo programa. Equivale ao **BEGIN** do MODULE no Modula-2.

O programa abaixo mostra na tela o seguinte texto

Eu tenho 33 anos de idade

e pula para a linha de baixo.

```
#include <stdio.h>

main()                // programa principal
{
 int idade;           // declara a variável idade
 idade = 33;
 printf ("Eu tenho %d anos de idade\n", idade) ;
}
```

2.3 A ESTRUTURA DE UM PROGRAMA EM C

A estrutura genérica de um programa em C é a que se apresenta a seguir, podendo alguns dos elementos não existir:

- Comandos do pré-processador
- Definições de tipos
- Protótipos de funções - declaração dos tipos de retorno e dos tipos dos parâmetros das funções
- Variáveis globais
- Funções

Deverá existir sempre uma função **`main`**. Esta é a "*porta de entrada do programa*". Através da função `main` é que podemos chamar a execução de outras funções. Todo programa deve ter uma única função `main`.

Exemplo:

```
void main()
{
  printf("Eu gosto do C\n");
}
```

O programa acima contém apenas uma função (a função main(), que é obrigatória), que não retorna nada (void) e que não tem parâmetros. Como instrução da função temos apenas a chamada a printf(), uma função da biblioteca standard que escreve no vídeo. Neste caso escreve uma cadeia de caracteres (string). A combinação \n no fim da string indica uma mudança de linha (o carácter new line ou line feed). Notar que no final de cada instrução existe sempre um terminador - ;

Capítulo 3 FUNÇÕES EM C

Forma geral de uma função em C:

```
tipo_de_retorno nome_da_função (lista_de_argumentos)
{
   código_da_função
}
```

Em Pascal temos definido de forma explícita quando uma operação é uma função e quando é um procedimento (function e procedure). Em Modula-2 esta distinção é implícita ou seja, você cria uma operação e, se ela retornar um valor é uma função, caso contrário é um procedimento.

Embora em C normalmente chamemos todas as operações de função, podemos fazer "funções" que não retornam nada e portanto comportam-se como procedimentos. Estas são as funções **void**. Se uma operação for **void,** ela não precisa ter um *return* no seu código.

EXEMPLO

Um programa-exemplo com duas funções que se comportam como procedimentos.

```
#include <stdio.h>

void hello()
{
  printf("Alô\n");
}

void main()
{
 hello(); /* chama a função hello */
}
```

EXEMPLO

Este programa exibe o caracter 'D' na tela do vídeo.

```
#include <stdio.h>
void main ()
{
   char Ch;          /* define a variável Ch como um char */
   Ch = 'D';         /* atribui o caracter D a variavel Ch */
   printf ("%c",Ch); /* usa printf para imprimir Ch. Veremos adiante */
}
```

Observe que esta operação é um procedimento pois não retorna nada.

No programa acima, **%c** indica que printf deve colocar um caractere na tela (Ch então deve ser do tipo char). Observe também que a atribuição em C é feita usando-se simplesmente o igual (=). Em Modula-2 usa-se o dois pontos igual (:=). Em Modula-2 o igual (**=**) é usado para estabelecer comparação de igualdade (em C a comparação de igualdade é feita com um duplo igual (==) como veremos adiante).

EXEMPLO

Este programa imprime o produto 12 x 7.

```
#include <stdio.h>

/* Função prod com dois argumentos inteiros */
int prod (int x,int y){
  return (x*y);
}

void main (){
  int saida;
  saida = prod (12,7); /* atribui valor da função prod para saida */
  printf ("A saida e': %i\n",saida);
}
```

No programa acima, **%i** indica que printf deve colocar um inteiro na tela (saida então deve ser do tipo int). Observe que a função **prod** é do tipo **int**, portanto deve retornar um inteiro.

EXEMPLO

Uma função para calcular o valor médio de dois números, poderia ser definida da seguinte forma:

```
float media(float a, float b)
{
  float med;

  med = (a + b) / 2;
  return med;
}

void main()
{
  float a=5, b=15, result;

  result = media(a, b);
  printf("Média = %f\n",result);
}
```

Repare na instrução de return na função, que além de a terminar é também a responsável pela definição do valor de retorno da mesma.

EXEMPLO

Lendo do teclado e imprimindo na tela

```
#include <stdio.h> /* inclui funções do stdio.h */
#include <conio.h> /* inclui funções do conio.h */
main ()            /* OBS: quando não indicamos o tipo de */
                   /*      retorno, é assumido int        */
{
  char Ch;
  Ch = getch();    /* pega caracter do teclado e atribui a Ch */
```

```
   printf ("Voce pressionou a tecla %c",Ch);
   return;
}
```

A função getch lê um caracter do teclado.

3.1 ALGUMAS FUNÇÕES DE ENTRADA E SAÍDA

Abaixo são apresentadas algumas funções de entrada e saída de dados.

3.1.1 PRINTF()

A função **printf** (*print formated*) é usada para mostrar informações na tela do vídeo (saída padrão). Esta função retorna um valor inteiro representando o número de caracteres impressos. Seguem abaixo alguns exemplos de uso da função Printf.

```
printf ("Teste %% %%")          /* Teste % % */
printf ("%f",40.345)            /* 40.345 */
printf ("Um caractere %c e um inteiro %i",'D',120)  /* D e 120 */
printf ("%s e um exemplo","Este")/* Este e um exemplo */
printf ("%s%i%%","Juros de ",10) /* "Juros de 10% */
```

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

3.1.2 GETS()

A função gets() lê caracteres do teclado (entrada padrão) e coloca-os em uma string (vetor de caracteres).

EXEMPLO

O programa abaixo pergunta o nome e a idade o usuário.

```
#include <stdio.h>

void main ()
{
  char nome  [101];
  char idade [4];
```

```
  printf ("Digite seu nome: ");
  gets (nome);

  printf ("Digite sua idade: ");
  gets (idade);
}
```

3.1.3 SCANF()

A função scanf() é uma rotina de entrada de uso geral que lê informações do teclado. Ela pode ler todos os tipos de dados intrínsecos e converte-los automaticamente no formato interno apropriado. Pode-se dizer a grosso modo que scanf() é um tipo de complemento da função printf().

EXEMPLO

O programa abaixo pergunta o nome e a idade o usuário.

```
#include <stdio.h>

int main () {

   char nome[100], sobrenome[100];
   int idade;

   printf ("Digite seu nome e sobrenome:");
   scanf ("%s%s",nome, sobrenome);

   printf ("Digite sua idade: ");
   scanf ("%i",&idade);

   return 0;
}
```

Observe que scanf() delimita os campos por espaço em branco. Assim, não seria apropriado usar scanf() para obter por exemplo o nome completo. Neste caso seria adequado usar a função gets().

3.2 FUNÇÕES SEM PARÂMETROS

Em alguns compiladores a palavra void deverá ser usada no lugar dos parâmetros se a função não tiver nenhum. Um exemplo:

```
void squares(void) {
  int k = 1;              /* declara k inteiro e inicia com 1 */

  while (k<=10){          /* enquanto k for menor ou igual a 10 */
    printf("%d\n", k*k);  /* mostra k*k e pula uma linha */
    k = k + 1;            /* inclrementa k */
  }
}

void main(void) {
  squares();              /* chama a função squares */
}
```

Na chamada de funções sem argumentos é sempre obrigatório utilizar parêntesis, como se vê acima.

3.3 PROTÓTIPOS

Em C, antes chamar uma função, é necessário informar ao compilador o seu o seu nome, tipo de retorno e o número e natureza dos seus argumentos. Se a função já estiver definida antes da chamada, não haverá problemas. Caso a função seja chamada antes de sua definição, é necessário declarar, a nível global e antes da chamada, um protótipo desta função. Um protótipo não é mais do que o cabeçalho da função seguido de ponto e vírgula (**;**).

Estritamente não é necessário indicar no protótipo os nomes dos parâmetros, mas apenas os seus tipos. No entanto, considera-se boa prática indicar também os nomes dos parâmetros.

EXEMPLO

```
#include <stdio.h>

void hello();        /* declara o protótipo da função hello */

void main() {
  hello();           /* chama a função hello */
}

void hello() {       /* implementação da função hello() */
  printf("Alô\n");
}
```

Capítulo 4 ALGUNS TIPOS DE VARIÁVEIS E OPERADORES

O C tem 5 tipos básicos: *char*, *int*, *float*, *double*, *void*. Existe ainda os modificadores de tipo:: *signed*, *unsigned*, *long* e *short*. Ao float não se pode aplicar nenhum e ao double pode-se aplicar apenas o long. Os quatro podem ser aplicados a inteiros.

Declaração de variáveis

```
char ch, letra;
long count;
float pi;
```

Podemos ainda iniciar o valor das variáveis no momento de sua declaração.

```
char ch    = 'D';
int count  = 0;
float pi   = 3.141;
```

4.1 TAMANHO DOS TIPOS BÁSICOS DE DADOS DO TURBO C:

Tipo	Tamanho (em bits)	Intervalo decimal
char	8	128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long double	80	3.4E-4932 a 1.1E+4932

4.2 STRINGS

No C uma string é um vetor de caracteres terminado com um caractere nulo. O caracter nulo é um caractere com valor ASCII igual a zero. O terminador nulo pode ser escrito usando a convenção de barra invertida do C como sendo '\0'. Para declarar uma string podemos usar o seguinte formato geral:

char nome_da_string [tamanho_da_string];

EXEMPLO

```
#include <stdio.h>
main ()
{
   char texto[10];
   printf ("Digite uma string: ");
   gets (texto);
   printf ("\n\nVoce digitou %s",texto);
   return;
}
```

No programa acima, **%s** indica que printf() deve colocar uma string na tela.

Suponhamos que o usuário ao executar este programa digite a *Bom dia*. Neste caso, teríamos armazenado na variável *texto*:

0	1	2	3	4	5	6	7	8	9
66	111	109	32	100	105	97	0	???	???
B	o	m		d	i	a	\0	??	??

Onde, texto[0] contém 'B', texto[1] contém 'o' e assim por diante até texto[7] que contém o valor '\0' que corresponde ao ASCII zero. Os valores armazenados em texto[8] e texto[9] são indefinidos (lixo).

Como o C não manuseia strings directamente todas as seguintes instruções são ilegais:

```
char nome[50], apelido[50], nome_completo[100];
nome = "Arnold";                                  /* Ilegal */
apelido = "Schwarznegger";                        /* Ilegal */
nome_completo = "Mr. " + nome + ' ' + apelido;    /* Ilegal */
```

No entanto a seguinte declaração com inicialização é válida:

```
char nome[50] = "Dave";
```

EXEMPLO

```
#include <string.h> // funções para manipular strings
#include <stdio.h>

void main( void )
{
   char string[80];
   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );

   printf( "String = %s\n", string );
}
```

4.3 O TIPO BOOLEANO EM C

Em C o verdadeiro e o falso não são tipos booleanos propriamente dito. Usa-se para falso o valor inteiro 0 (zero) e para verdadeiro qualquer outro valor inteiro.

- **Verdadeiro:** qualquer valor inteiro diferente de 0 (zero).
- **Falso:** o valor 0 (zero).

Assim, em uma estrutura IF podemos fazer:

```
void main() {
  int a;
  a = 0;
  if (a)
    printf (“verdadeiro”);  /* se a for != 0 */
  else
    printf (“falso”);       /* se a for = 0 */
}
```

4.4 OPERADORES ARITMÉTICOS

Existem cinco operadores aritméticos em C. Cada operador aritméticos está relacionado ao uma operação aritmética elementar: adição, subtração, multiplicação e divisão. Existe ainda um operador (%) chamado operador de módulo cujo significado é o resto da divisão inteira. Os símbolos dos operadores aritméticos são:

Adição	Subtração	Multiplicação	Divisão	Resto	Incremento	Decremento
+	-	*	/	%	++	--

4.5 OPERADORES DE INCREMENTO E DECREMENTO

Em C pode-se otimizar o incremento ou decremento de uma variável utilizando-se os operadores de incremento e decremento.

```
x++: equivale a  x = x + 1
x--: equivale a  x = x - 1
```

Há diferença entre **++i** e **i++**. No primeiro caso o valor de **i** é primeiro incrementado e depois usado. No segundo cado, primeiro o **i** é usado e depois incrementado.

EXEMPLO

```
void main ( )
{
  int i;
  i = 10;
  i++;
  printf (“%i”,i);      /* exibe 11 no vídeo */
  printf (“%i”,i++);    /* exibe 11 no vídeo */
  printf (“%i”,++i);    /* exibe 13 no vídeo */
}
```

EXEMPLO

Observe o fragmento de código abaixo e note o valor que as variáveis recebem após a execução da instrução:

```
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 */
```

Os operadores incrementais são bastante usados para o controle de laços de repetição.

4.6 ATRIBUIÇÕES MÚLTIPLAS

A linha abaixo faz com que as 3 variáveis recebam o valor 5. Este tipo de atribuição é muito usada.

```
x = y = z = 5;
```

4.7 ALGUNS OPERADORES RELACIONAIS E LÓGICOS

Operador E	Operador Ou	Não	Igual	Diferente
&&	||	!	= =	! =

EXEMPLO

```
if (10 > 4 && !(10 < 9) || 3 <= 4)
```

Resultaria em Verdadeiro pois dez é maior que quatro E dez não é menor que nove OU três é menor ou igual a quatro.

4.8 CASTS

Algumas vezes queremos, momentaneamente, modificar o tipo de dado representado por uma variável, isto é, queremos que o dado seja apresentado em um tipo diferente do qual a variável foi inicialmente declarada. Por exemplo: declaramos uma variável como int e queremos que seu conteúdo seja apresentado como float. Este procedimento é chamado de conversão de tipo ou casting (moldagem, em inglês).

O cast é um modelador que é aplicado a uma expressão. Ele força a expressão a ser de um tipo especificado. Sua forma geral é:

(tipo) expressão

EXEMPLO

Observe o trecho de programa abaixo:

```
int num;
float valor = 13.0;
```

```
num = valor % 2;       /* inválido! */
num = (int)valor % 2;  /* válido! */
```

Observe que usamos a conversão de tipo para que o dado armazenado em valor fosse transformado no tipo int assim a operação módulo pode ser efetuada.

EXEMPLO

```
#include <stdio.h>
main ()
{
   int num;
   float f;
   num = 10;
   f   = (float) num/7;
   printf ("%f",f);
}
```

EXEMPLO:

Seja x um int e y um float.

```
float y;
int x=3;

y = (float) x/2;      /* y=1.5 */
y = (float) (x/2)     /* y=1.0 */
```

4.9 DEFINIÇÃO DE VARIÁVEIS LOCAIS

Podemos definir blocos de comandos em C. Neste blocos podemos definir variáveis e seu escopo será apenas dentro destes blocos.

EXEMPLO

No programa abaixo, as variáveis *menor* e *maior* são locais àfunção main enquanto *aux* é local ao if.

```
#include <stdio.h>

void main ()
{
   int menor, maior;

   printf ("Limite inferior e superior: ");
   scanf ("%i",&menor);
   scanf ("%i",&maior);

   if (menor > maior)
   {
      int aux;
      aux = menor;
```

```
        menor = maior;
        maior = aux;
    }
    ...
}
```

4.10 DEFINIÇÃO DE VARIÁVEIS GLOBAIS

As variáveis globais, visíveis em todas as funções de um programa, declaram-se fora e antes de todas as funções (só são visíveis a partir do local da daclaração).

EXEMPLO:

No exemplo abaixo, as variáveis number, sum e letter são globais.

```
short number, sum;
char letter;

void main(void)
{
 /* Corpo do programa ... */
}
```

Capítulo 5 ESTRUTURAS BÁSICAS DE CONTROLE

Este capítulo trata dos vários métodos de controlar o fluxo de execução de código num programa em C. Os operadores lógicos e de comparação são fundamentais para isso. Esses operadores são usados em conjunto com as instruções aqui tratadas.

5.1 ESTRUTURA IF

A instrução if tem a mesma função e estrutura de outras linguagens, como por exemplo o Pascal. Assume em C as duas formas básicas seguintes:

```
if (expressão)
   instrução;
```

ou

```
if (expressão)
   instrução_1;
else
   instrução_2;
```

EXEMPLO

```
if (num==10) {
   printf ("\n\nVoce digitou 10!\n");
}
else {
   if (num>10) {
      printf ("O numero e maior que 10.");
   }
   else {
      printf ("O numero e menor que 10.");
   }
}
```

O exemplo acima poderia ainda ser escrito suprimindo-se alguns { }.

```
if (num==10)
   printf ("\n\nVoce digitou 10!\n");
else{
   if (num>10)
      printf ("O numero e maior que 10.");
   else
      printf ("O numero e menor que 10.");
}
```

5.1.1 EXPRESSÃO CONDICIONAL

Uma estrutura muito interessante do C é a **expressão condicional**. Em C a expressão condicional é feita utilizando-se o operador ternário **?** (chama-se operador ternário porque requer 3 operandos).

Tem a forma:

```
expressão_1 ? expressão_2 : expressão_3;
```

Tem como resultado o valor de expressão_2 ou de expressão_3 consoante o valor de expressão_1 for verdadeiro (!= 0) ou falso (== 0) respectivamente.

EXEMPLO:

```
maior = (a > b) ? a : b;
```

Nesta linha, **maior** receberá o valor de **a** se **a > b**. Caso contrário, receberá o valor de **b**.

Isto equivale ao seguinte código:

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

EXEMPLO

Observe as expressões condicionais abaixo e verifique o resultado de sua avaliação.

Admita que i, j e k são variáveis tipo int com valores 1, 2 e 3, respectivamente.

Expressão	**Valor**
`i ? j : k`	2
`j > i ? ++k : --k`	4
`k == i && k != j ? i + j : i - j`	-1
`i > k ? i : k`	3

5.2 ESTRUTURA CASE

Esta estrutura possui a seguinte sintaxe:

```
switch(expressão){
  case rótulo_1:
    conjunto_1
  case rótulo_2:
    conjunto_2
  ...
  case rótulo_n:
    conjunto n
  [default:
    conjunto d]
}
```

onde:

expressão é uma expressão inteira.

rótulo_1,rótulo_2,...rótulo_n e rótulo_d são constantes inteiras.

conjunto 1, ..., conjunto n e conjunto d são conjuntos de instruções.

O valor de *expressão* é avaliado e o fluxo lógico será desviado para o conjunto cujo *rótulo* é igual ao resultado da expressão e todas as instruções **abaixo** deste rótulo serão executadas. Caso o resultado

da expressão for diferente de todos os valores dos rótulos então *conjunto d* é executado. Os rótulos devem ser expressões constantes inteiras **diferentes** entre si. O rótulo `default` é opcional.

Esta estrutura é particularmente útil quando se tem um conjunto de instruções que se deve executar em ordem, porém se pode começar em pontos diferentes.

EXEMPLO

```
#include <stdio.h>
void main ()
{
   int num;
   printf ("Digite um numero: ");
   scanf ("%i",&num); /* pega um caracter do teclado */

   switch (num)
   {
         case 7:
         case 8:
         case 9:
                  printf ("\n\nO numero e 7, 8 ou 9.\n");
                  break;
        case 11: break;
         case 10:
                  printf ("\n\nO numero e igual a 10.\n");
                  break;
         case 12:
         case 13:
                  printf ("\n\nO numero e 12 ou 13.\n");
                  break;
         default:
                  printf ("\n\nO numero não está entre 7 e 13.\n");
   }
}
```

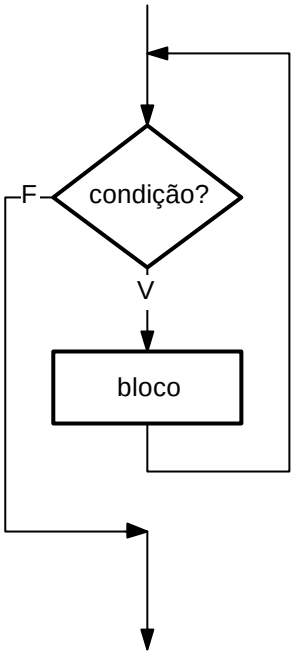
No exemplo acima, scanf é equivalente ao getch já mostrado em outros exemplos. O símbolo & em frente da variável num indica que esta está sendo passada por referência. Equivale aproximadamente ao VAR do Modula-2 ou do Pascal.

5.3 LAÇO WHILE

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

```
while(condição){
  bloco
}
```

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. Observe que nesta estrutura, o bloco de instruções pode não ser executado nenhuma vez, basta que a condição seja inicialmente falsa.



EXEMPLO

```
#include <stdio.h>
main ()
{
   char Ch;
   Ch='\0';
   while (Ch != 'q')    /* enquanto o Ch não for igual a q */
   {
        scanf("%c",&Ch); /* pega um caracter do teclado */
   }
}
```

5.4 LAÇO DO - WHILE

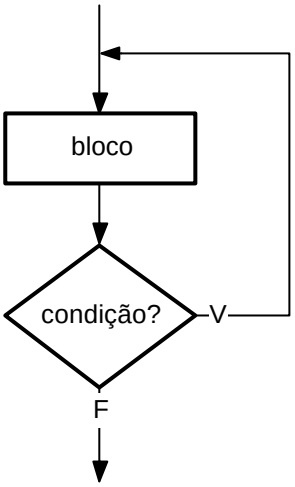
Esta é uma estrutura básica de repetição condicional. Permite a execução de um bloco de instruções repetidamente. Sua sintaxe é a seguinte:

```
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. O fluxograma desta estrutura é mostrado abaixo.



EXEMPLO

No trecho abaixo, a leitura de um número é feita dentro de um laço de repetição condicional. A leitura é repetida caso o número lido seja negativo.

```
do{
  puts("Digite um número positivo:");  /* mostra uma string na tela */
  scanf("%f",&num);                    /* coloca valor em num */
} while(num <= 0.0);
```

EXEMPLO

O exemplo abaixo controla a escolha de um menu.

```
char Ch;
do
{
   printf ("\n\nEscolha um:\n\n");
   printf ("\t(1)...Mamao\n");
   printf ("\t(2)...Abacaxi\n");
   printf ("\t(3)...Laranja\n\n");
   scanf("%c",&Ch);

} while ((Ch!='1')&&|(Ch!='2')&&(Ch!='3'));
```

5.5 O LAÇO FOR

O laço FOR no C é bastante poderoso. Sua forma mais simples é:

```
for (<início>;<condição>;<incremento>)
   comando;
```

Observação: Muitas vezes a tradução de um laço FOR do C para uma outra linguagem como Modula-2, por exemplo, é feita utilizando-se um laço WHILE.

EXEMPLO

O exemplo abaixo imprime os números 1 a 100 na tela.

```
#include <stdio.h>
void main ()
{
  int count;
  for (count=1; count <= 100; count++)
  {
    printf ("%i ",count);
  }
}
```

A forma geral do laço for é a seguinte:

```
for (inicialização1, inicialização2... ; condição ; incremento1,
incremento2...)
{ /* comandos */   }
```

Pode existir mais de uma expressão de *inicialização* e de *incremento* na estrutura `for`. Estas expressões devem ser separadas por vírgula (,). Mas **não pode** haver mais de uma expressão de *condição*. Exemplo: `for(i=0, j=10; i<10; i++, j--){...}`

EXEMPLO

Cálculo da amplitude de um conjunto de valores.

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

void main(){

  int i;            // contador de iteracao
  int num;          // numero de valores lidos
  float val;        // valor lido
  float max,min;    // valor maximo, valor minim

  puts("Digite numeros reais...");
  puts("Quantos valores? ");
  scanf("%d",&num);             // leitura do numero de valores

  for(i = 1; i <= num; i++){    // laco iterativo para i de 1 a num

    printf("%d§ valor: ",i);    // leitura dos valores
    scanf("%f",&val);
    if(i == 1){                 // se 1a leitura...
      max = min = val;          // inicializa valores
    }

    max = val > max ? val : max;// calcula maximo
    min = val < min ? val : min;// calcula minimo
```

```
    }                                   // fim do laco

    printf("\nAmplitude: [%.3f , %.3f]",min,max); // imprime min,max
}
```

Não é necessário especificar todos os parâmetros do laço **for**.

EXEMPLO

No trecho abaixo, a ausência de condições de inicialização, continuidade e terminação, causarão repetição infinita (loop infinito).

```
for(;;)
{
   printf(“Este loop rodará eternamente!\n”);
}
```

Capítulo 6 VETORES E MATRIZES

6.1 VETORES OU ARRAYS UNIDIMENSIONAIS

Em C, um vetor é um conjunto de variáveis de um **mesmo tipo** que possuem 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.
- *tam* é o tamanho do vetor, isto é, o número de elementos que o vetor pode armazenar.

EXEMPLO:

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

EXEMPLO:

```
float vetor [20];
```

O C irá reservar 4x20=80 bytes (supondo que float é armazenado em 4 bytes). Estes bytes são reservados de maneira contígua. Na linguagem C a numeração dos vetores começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

```
vetor [0]
vetor [1]
   ...
vetor [19]
```

O C não verifica se o índice que você usou está dentro dos limites válidos. Ninguém o impede de escrever:

```
vetor [30]
vetor [103]
```

EXEMPLO

O exemplo abaixo coloca valores em uma matriz bidimensional de 20 linhas e 10 colunas.

```
#include <stdio.h>
main ()
{
  int mtrx [20][10];
  int i,j,count;
  count=1;

  for (i=0;i<20;i++)
    for (j=0;j<10;j++)
    {
      mtrx[i][j]=count;
      count++;
    }
}
```

6.1.1 PASSANDO VETORES PARA FUNÇÕES

Vetores, assim como variáveis, podem ser usados como argumentos de funções.

EXEMPLO

O programa abaixo calcula a média de um conjunto de elementos.

```
#include <stdio.h>

#define MAX  10   // define uma constate MAX com valor 10

float media (int vet_num[], int limite); // protótipo da função

void main ()
{
   int num, i, vet[MAX];

   printf ("Número de elementos: ");
   scanf ("%d",&num);

   for (i = 0; i< num; i++)
   {
      printf ("Número %d: ",i+1);
      scanf ("%d",&vet[i]);
   }

   printf ("A média dos valores é: %f", media (vet, num));
}

float media (int vet_num[], int limite)
{
   int i, soma = 0;
```

```
   for (i = 0; i< limite; i++)
      soma = soma + vet_num[i];

   return ((float)soma / limite); // observe o uso do cast
}
```

Atenção: Ao contrário das variáveis comuns, o conteúdo de um vetor **pode ser modificado** pela função chamada. Isto ocorre porque a passagem de **vetores** para funções é feita *por referência*.

6.1.2 ALGUMAS INICIALIZAÇÕES DE VETORES

As inicializações abaixo são válidas.

```
float vect [6]   = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
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'};
int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Porém, **NÃO SÃO VÁLIDAS** as atribuições:

```
vect  = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
dia   = {12,30,14,7,13,15,6};
nota  = {8.4,6.9,4.5,4.6,7.2};
vogal = {'a', 'e', 'i', 'o', 'u'};
matrx = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

6.2 MATRIZES (OU ARRAYS)

As matrizes são também chamadas de vetores multidimensionais ou arrays. A declaração e inicialização de matrizes é feita de modo semelhante aos vetores. A sintaxe para declaração de matrizes é:

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

onde:

- `tipo` é o tipo dos elementos da matriz.
- `nome` é o nome da matriz.
- `[tam_1][tam_2]...[tam_N]` é o tamanho de cada dimensão da matriz.
- `{{lista},{lista},...{lista}}` são as listas de elementos.

EXEMPLO

Veja algumas declarações e inicializações de matrizes.

```
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}};
```

Neste exemplo, `nota` é matriz de **duas** dimensões (`[][]`). Esta matriz é composta de 5 vetores de 3 elementos cada. A variável `tabela` é uma matriz de três dimensões (`[][][]`).

6.2.1 PASSAGEM DE MATRIZES PARA FUNÇÕES

A sintaxe para *passagem* de matrizes para funções é semelhante a passagem de vetores unidimensionais: chamamos a função e passamos o **nome** da matriz, **sem** índices. A única mudança ocorre na declaração de funções que recebem matrizes:

Na *declaração* de funções que recebem vetores:

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

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

EXEMPLO

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);   // passagem do vetor para a função
   ...
}
```

Capítulo 7 STRUCTURES (RESGISTROS)

As estruturas em C são muito semelhantes aos registos em Pascal. Agrupam num único tipo vários valores (campos), que podem ser de tipos diferentes.

EXEMPLO:

```
struct people
{
   char name[50];
   int age;
   float salary;
};

struct people John;
```

As declarações anteriores definem uma estrutura chamada *people* e uma variável (John) desse tipo.

A estrutura definida tem um nome (tag) que é opcional. Quando as estruturas são definas com tag podemos declarar posteriormente variáveis, argumentos de funções, e também tipos de retorno, usando esse tag, como se mostrou no exemplo anterior. É também possível definir estruturas sem tag (anónimas). Neste último caso as variáveis terão de ser nomeadas entre o último } e ; da forma habitual. Por exemplo:

```
struct {
   char name[50];
   int age;
   float salary;
} John;
```

Podemos ainda iniciar as variáveis do tipo estrutura quando da sua declaração, colocando os valores pela ordem dos campos definidos na estrutura entre chavetas:

```
struct people John = { "John Smith", 26, 124.5 };
```

Para acessar um campo, ou membro, de uma estrutura utiliza-se, à semelhança do Pascal, o operador ponto. Para aumentar o salário do Sr. John Smith:

```
John.salary = 150.0;
```

EXEMPLO

Segue um exemplo de uma struct usada dentro de outra.

```
struct tipo_endereco
{
  char rua [50];
  int numero;
  char bairro [20];
  char cidade [30];
  char sigla_estado [3];
```

```
   long int CEP;
};


struct ficha_pessoal
{
   char nome [50];
   long int telefone;
   struct tipo_endereco endereco;
};

struct ficha_pessoal  Ficha;
```

7.1 O COMANDO TYPEDEF

O comando typedef permite ao programador definir um novo tipo ou um novo nome para um determinado tipo. Sua forma geral é:

```
typedef tipo novo_tipo;
```

Como exemplo vamos dar o nome de inteiro para o tipo int:

```
typedef int inteiro;
```

A definição de novos tipos com typedef pode também ser efetuada com estruturas. A declaração seguinte define um novo tipo person, podendo posteriormente declarar-se e inicializar-se variáveis desse tipo:

```
typedef struct people {
   char name[50];
   int age;
   float salary;
} person;

person John = { "John Smith", 26, 124.5 };
```

Neste exemplo o nome people também funciona como tag da estrutura e também é opcional. Nesta situação a sua utilidade é reduzida.

Arrays e estruturas podem ser misturados (aliás como quaisquer outros tipos):

```
typedef struct people {
   char name[50];
   int age;
   float salary;
} person;

person team[1000];
```

Aqui a variável team é composta por 1000 person's.

Um acesso poderia ser, por exemplo:

```
team[41].salary = 150.0;
```

ou,

```
total_salaries += team[501].salary;
```

7.2 UNIÕES

Uma variável do tipo união pode conter (em instantes diferentes) valores de diferentes tipos e tamanhos. Na linguagem C a declaração de uniões é em tudo semelhante à declaração de estruturas, empregando-se a palavra union em vez de struct. Por exemplo, podemos ter:

```
union number {
   short  sort_number;
   long   long_number;
   double real_number;
} a_number;
```

Aqui declara-se uma união chamada number e uma variável desse tipo - a_number. Esta variável poderá conter, em instantes diferentes, um short, um long ou um double. A distinção faz-se pelo nome do campo respectivo. Quando se preenche um determinado campo, o que estiver noutro qualquer é destruído. Os vários campos têm todos o mesmo endereço na memória.

Os campos ou membros são acessados da mesma forma do que nas estruturas:

```
printf("%ld\n", a_number.long_number);
```

Quando o compilador reserva memória para armazenar uma união apenas reserva o espaço suficiente para o membro maior (no exemplo de cima serão 8 bytes para o double). Os outros membros ficam sobrepostos, com o mesmo endereço inicial.

Capítulo 8 PONTEIROS

Neste capítulo veremos a definição e os principais usos de ponteiros. Veremos as operações fundamentais como ponteiros , a estreita relação de ponteiros vetores e strings e ainda a alocação dinâmica de memória e a passagem de funções para funções com o uso de ponteiros..

Para declarar um ponteiro temos a seguinte forma geral:

```
tipo_do_ponteiro * nome_da_variável;
```

É o asterisco (*) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp,*pt2;
```

Para saber o endereço de uma variável basta usar o operador &. Veja o exemplo:

```
int count=10;
int *pt;
pt = &count;
*pt = 11;          // count e *pt passaram a ser 11
count = 12;        // agora count e *pt passaram a ser 12
```

EXEMPLO

```
#include <stdio.h>
void main ()
{
  int num,valor;
  int *p;
  num = 55;
  p = &num;     /* Pega o endereco de num */
  valor=*p;     /* Valor e igualado a num de uma maneira indireta */
  printf ("\n\n%i\n",valor);
  printf ("Endereco para onde o ponteiro aponta: %p\n",p);
  printf ("Valor da variavel apontada: %i\n",*p);
}
```

```
#include <stdio.h>
void main ()
{
  int num,*p;
  num = 55;
  p = &num;     /* Pega o endereco de num */
  printf ("\nValor inicial: %i\n",num);
  *p = 100;     /* Muda o valor de num de uma maneira indireta */
  printf ("\nValor final: %i\n",num);
}
```

Explique a diferença entre: p++; (*p)++; *(p++);

EXEMPLO

```
Ponteiro como argumento de funcao.

#include <conio.h>
#include <stdio.h>
#include <math.h>

void main(){

  void round(float *); // prototipo de funcao
  float num;           // declarando uma variavel real
  float *p = &num;     // declarando um ponteiro real

  clrscr();
  printf("Digite um numero real para se arredondado: ");
  scanf ("%f",p);      // observe a sintaxe alternativa para

  round(p);

  printf("\nNumero Arredondado: %.2f ",*p);

  getch();
}

void round(float *q)
{
  *q = floor(*q + 0.5); // arredonda para baixo ou para cima!
}
```

8.1 PONTEIROS COMO VETORES

É muito comum manipular vetores e matrizes por meio de ponteiros.

```
#include <stdio.h>
main ()
{
  int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int *p;
  p = matrx;
  printf ("O terceiro elemento da matriz e': %i", p[2]);
}
```

Podemos ver que p[2] equivale a *(p+2).

Qual será o resultado do procedimento abaixo?

```
void main ()
{
  int i[2]={10,20,30}, *j, k;

  j = i; /* ou j = &i[0] */
```

```
  j++;
  (*j)++;
  k = *(j++);
  printf (“O valor de K e’:&d”, k);
}
```

EXEMPLO

Neste exemplo, a função StrCpy() funcionará como a função strcpy() da biblioteca do C.

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

void StrCpy (char *destino,char *origem)
{
  while (*origem)
  {
    *destino = *origem;
    origem++;
    destino++;
  }
  *destino = '\0';
}

void main ()
{
  char str1[100], str2[100], str3[100];

  printf ("Entre com uma string: ");
  gets   (str1);

  StrCpy (str2, str1);
  StrCpy (str3, "Voce digitou a string ");

  printf ("\n\n%s%s", str3, str2);
}
```

EXEMPLO

O exemplo abaixo define a função troca que inverte os valores de duas variáveis.

```
#include <stdio.h>

void troca(int *p1, int *p2)
{
  int temp;      // variável temporária

  temp = *p1;    // temp recebe o conteúdo apontado por p1
  *p1  = *p2;    // o conteúdo de p1 recebe o conteúdo de p2
  *p2  = temp;   // o conteúdo de p2 recebe o valor de temp
}

void main()
{
```

```
  int a,b;

  scanf ("%d %d", &a, &b);  // leitura das variáveis
  troca (&a, &b);           // passagem dos endereços de a e b
  printf("%d %d", a, b);    // imprime valores (trocados!)
}
```

8.2 PONTEIROS PARA ESTRUTURAS

Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados.

EXEMPLO

Usando as estruturas já definidas nos capítulos anteriores.

```
struct tipo_endereco
{
  char rua [50];
  int numero;
  char bairro [20];
  char cidade [30];
  char sigla_estado [3];
  long int CEP;
};

struct ficha_pessoal
{
  char nome [50];
  long int telefone;
  struct tipo_endereco endereco;
};

struct ficha_pessoal  Ficha, *p;
```

Se apontarmos o ponteiro p declarado acima para uma estrutura qualquer e quisermos acessar um elemento da estrutura poderíamos fazer:

```
(*p).nome
```

Este formato raramente é usado. O que é comum de se fazer é acessar o elemento nome através do operador seta (->). Assim faremos:

```
p->nome
```

A declaração acima é muito mais fácil e concisa. Para acessarmos o elemento CEP dentro de endereco faríamos:

```
p->endereco.CEP
```

8.3 ALOCAÇÃO DINÂMICA DE MEMÓRIA

A função malloc() é usada para "pedir" ao sistema operacional uma porção de memória. O seu protótipo é:

```
void *malloc(int number_of_bytes);
```

A função retorna um apontador genérico (void *) para o inicio da memória alocada. Se não for possível alocar a quantidade de memória solicitada, a função retorna o apontador NULL.

Então se quisermos alocar espaço para 100 inteiros podemos usar:

```
int *ip;
ip = (int *) malloc(100 * sizeof(int));
```

Notar o uso de um **cast** na saída da função e também o uso do operador **sizeof** para determinar, em bytes, o tamanho de um inteiro.

Quando o elemento não for mais necessário, é indispensável libertar a memória alocada. A libertação de memória alocada com malloc() (e só essa) é feita chamando a função standard **free().**

```
free(link);
```

Capítulo 9 PROGRAMA EXEMPLO

O programa abaixo é um pequeno exemplo de um programa em C que utiliza alguns dos principais conceitos apresentados neste texto.

```
/*****************************************************************
LISTA.C

DESCRICAO:
   O programa define uma estrutura "lista de registros encadeados",
   com operacoes para manipular esta lista.

   Este programa esta L O N G E  de ser um exemplo completo da
   linguagem C mas serve para exemplificar alguns dos principais
   conceitos que aprendemos sobre C.

*****************************************************************/


#include <stdlib.h> /* por causa do malloc, free... */
#include <stdio.h>  /* por causa das rotina para mostrar na tela... */
#include <string.h> /* por causa do strcpy... */

#define FALSE 0     /* defino estes dois simbolos para ficar + amigavel */
#define TRUE  1
#define NOMELEN 40  /* defino o tamanho do campo nome */

typedef struct Reg{
   char nome [NOMELEN]; /* e'o mesmo que: char nome[40] */
   int  salario;
   struct Reg  *proximo;
} T_Reg, *Reg;


typedef struct {
   Reg primeiro, ultimo;
} T_Lista, *Lista;


/* CriaLista
      Cria uma lista do tipo T_Lista e retorna um ponteiro
      para a area de memoria alocada.
      Caso hava problemas ao alocar a memoria, retorna NULL.
*/
Lista CriaLista (){
   Lista lista;

   if ((lista = (Lista) malloc(sizeof(T_Lista))) == NULL)
      return (NULL);
   lista->primeiro = NULL;
   lista->ultimo   = NULL;

   return (lista);
}


/* IncluiLista
      Inclui um elemento na lista.
      Se houver problemas com memoria, retorna FALSE.
```

```
      Se a inclusao for feita com sucesso, retorna TRUE
*/
int IncluiLista (Lista lista, char * nome, int salario){
   Reg reg;

   /* Aloca espaco para um novo registro de funcionario */
   if ((reg = (Reg) malloc (sizeof(T_Reg)))== NULL)
      return FALSE;

   /* Preenche os campos do registro */
   strcpy (reg->nome, nome);
   reg->salario = salario;
   reg->proximo = NULL; /* faz apontar para "nil" */

   /* Se a lista estiver vazia, inclui o primeiro elemento */
   if (lista->ultimo == NULL){
      lista->ultimo = reg;
      lista->primeiro = reg;
   }
   else {
      lista->ultimo->proximo = reg;
      lista->ultimo = reg;
   }

   return TRUE;
}

/* ExcluiLista
      Exclui o primeiro elemento da lista.
      Se a lista estiver vazia, retorna FALSE.
      Se a exclusao for feita com sucesso, retorna TRUE.
*/
int ExcluiLista (Lista lista){

   Reg reg;

   /* Se a lista esta vazia, nao tenho nada a fazer */
   if (lista->primeiro == NULL)
      return FALSE;

   /* Salvo a posicao do primeiro da fila */
   reg = lista->primeiro;

   /* Faco o primeiro da lista apontar para o proximo da lista*/
   lista->primeiro = lista->primeiro->proximo;

   /* Elimino o primeiro da lista */
   free (reg);

   /* Se so' havia um elemento, entao o ultimo foi eliminado
      e ultimo deve apontar para NULL */
   if (lista->primeiro == NULL)
      lista->ultimo = NULL;

   return TRUE;
}

/* InfoLista
      Retorna o ponteiro para o primeiro elemento da lista.
*/
Reg InfoLista (Lista lista) {
   return (lista->primeiro);
}

/* VaziaLista
```

```
      Retorna TRUE se a lista estiver vazia e FALSE caso contrario.
*/
int VaziaLista (Lista lista) {
   return (lista->primeiro == NULL);
}


/* DestroeLista
      Desaloca toda a memoria usada pela lista
*/
void DestroeLista (Lista lista){
   Reg reg, aux;

   reg = lista->primeiro;

   /* Desaloco cada um dos elementos da lista */
   while (reg != NULL){ /* ou simplesmente while (reg){ */

      aux = reg->proximo; /* salvo o ponteiro para o proximo */
      free (reg);
      reg = aux;

   }

   /* Desaloco a estrutura lista */
   free (lista);

}


/* Funcao principal. Faz so alguns testes.
*/
void main () {

   int i;
   Lista lista;
   Reg reg;

   lista = CriaLista();

   for (i=1; i<= 10; i++)
      IncluiLista (lista,"Chico", i);


   while (!VaziaLista(lista))
   {
      reg = InfoLista (lista);
      printf ("Nome: %s  Salario: %i \n",reg->nome, reg->salario);
      ExcluiLista(lista);
   }

   DestroeLista(lista);
}
```

Capítulo 10 COMPARAÇÃO ENTRE C E MÓDULA-2

C	Modula-2
#include <stdio.h>	FROM IO IMPORT wrStr, wrLn;
/* */	(* *)
{ *comandos* }	BEGIN *comandos* *END*
main	BEGIN do MODULE

tipo_de_retorno nome_da_função (argumentos)
{
/* *comandos* */
}
	nome_da_função (argumentos) : tipo_de_retorno;
	BEGIN
	(**comandos* *)
	END nome_da_função;

tipo_da_variável nome_da_variável;	VAR nome_da_variável : tipo_da_variável;
typedef tipo nome_do_tipo;	TYPE nome_do_tipo = tipo;
char, int, float, double	CHAR, INTEGER, REAL, LONGREAL
void	não há este tipo primitivo
unsigned int	CARDINAL
FALSE é o inteiro 0 TRUE é qualquer outro valor inteiro	TRUE e FALSE são valores de um tipo especial booleano.
x++	x := x +1
a = b;	a := b;

if (a == b) {
/* *comandos* */
}
	IF (a = b) THEN
	(* *comandos* *)
	END

`maximo = (a > b) ? a : b;`	IF (a > b) THEN
	maximo := a;
	ELSE
	maximo := b;

switch	CASE

for (i = 1, j = 20**;** i < j **;** i++, j--) {
/* *comandos* */
}
	i := 1; j:= 20;
	WHILE (i < j) DO
	i := i + 1;
	j := j - 1;
	(* *comandos* *)
	END;

do { /* *comandos* */ } while *condição*;	REPEAT (* *comandos* *) UNTIL *condição*;