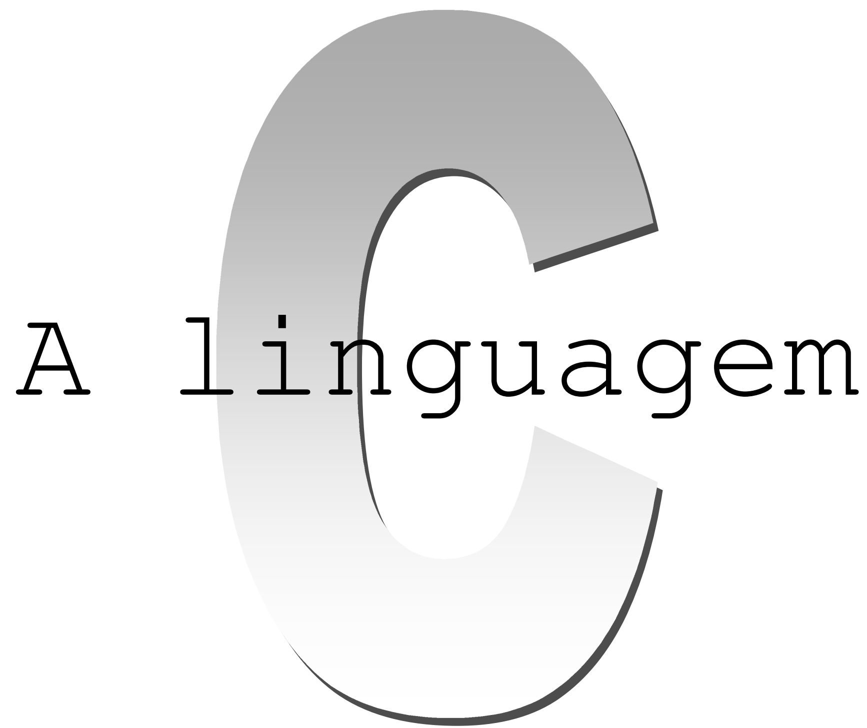


Prof. Tomás Dias Sant' Ana
Tecnólogo em Processamento de Dados pela UNIFENAS
Bacharel em Ciência da Computação pela UNIFENAS
Mestre em Ciência Computação pela Universidade de São Paulo - USP

Apostila Original desenvolvida pelo Prof. Luiz Eduardo da Silva.

ÍNDICE

1. APRESENTAÇÃO 1-4

2. A LINGUAGEM C 2-5

CARACTERÍSTICAS PRINCIPAIS DA LINGUAGEM C 2-5
BIBLIOTECA DE FUNÇÕES E LINKEDIÇÃO 2-6
ANATOMIA DE UM PROGRAMA EM C 2-6

3. TIPOS DE DADOS 3-7

OS TIPOS BÁSICOS DE DADOS EM C 3-7
OS MODIFICADORES DE TIPO 3-7
NOME DE IDENTIFICADOR 3-7
VARIÁVEL 3-8
A. Variável local: 3-8
B. Parâmetros formais 3-8
C. Variaveis Globais 3-8
Inicialização de Variáveis 3-9

4. ENTRADA E SAÍDA PELO CONSOLE 4-10

LENDO E ESCREVENDO CARACTERES 4-10
Alternativas para getchar 4-10
LENDO E ESCREVENDO STRING 4-10
ENTRADA E SAÍDA FORMATADA 4-11
Saída formatada: 4-11
Entrada formatada: 4-12

5. CONSTANTES, OPERADORES E EXPRESSÕES 5-13

CONSTANTES HEXADECIMAIS E OCTAIS 5-13
CONSTANTES DE BARRA INVERTIDA 5-13
OPERAÇÃO DE ATRIBUIÇÃO 5-14
Conversão de tipos na Atribuição 5-14
Atribuições Múltiplas 5-14
OPERADORES ARITMÉTICOS 5-14
Incremento (++) e decremento (--) 5-14
Precedência dos Operadores Aritméticos 5-15
OPERADORES RELACIONAIS E LÓGICOS 5-15
Tabela Verdade dos Operadores lógicos 5-15
Precedência dos Operadores Lógicos e Relacionais 5-16
OPERADORES BIT A BIT 5-16

6. COMANDOS DE CONTROLE DE PROGRAMA 6-18

COMANDO DE SELEÇÃO 6-18
Comando if (Seleção simples) 6-18
if's aninhados 6-19
O operador ? - Alternativo para o if-else 6-19
Comando switch - Seleção Múltipla 6-20
COMANDOS DE ITERAÇÃO 6-20
A malha (laço) de repetição FOR 6-21
O laço de repetição while 6-22
O laço do-while 6-23
COMANDOS DE DESVIO 6-24
Comando return 6-24
Comando break 6-24
Função exit 6-25
Comando continue 6-25

7. MATRIZES E STRINGS 7-26

MATRIZ 7-26
Matrizes Unidimensionais 7-26
Matrizes Multidimensionais 7-27

Matrizes passadas para função *7-27*
STRINGS - (MATRIZ DE CARACTERES) 7-28
Ponteiros e índices *7-29*
Inicialização de Matrizes *7-29*
8. PONTEIROS **8-30**
OPERADORES DE PONTEIROS 8-30
EXPRESSÕES USANDO PONTEIRO 8-30
Atribuição *8-30*
Aritmética *8-30*
Comparação *8-31*
PONTEIROS E MATRIZES 8-32
9. FUNÇÕES **9-33**
CHAMADA POR VALOR E CHAMADA POR REFERÊNCIA 9-33
Chamada por valor *9-33*
Chamada por referência *9-34*
FUNÇÕES DE BIBLIOTECA 9-34
Alguns arquivos de cabeçalho e suas funções principais: *9-34*
10. ESTRUTURAS EM C **10-35**
REFERÊNCIA A ELEMENTOS DE ESTRUTURA 10-35
ATRIBUIÇÃO DE ESTRUTURAS 10-35
MATRIZES DE ESTRUTURAS 10-36
PASSANDO ESTRUTURAS INTEIRAS PARA FUNÇÕES 10-36
PONTEIROS PARA ESTRUTURAS 10-37
A PALAVRA CHAVE TYPEDEF 10-37
11. ENTRADA E SAÍDA COM ARQUIVO EM C **11-39**
STREAMS E ARQUIVO 11-39
Funções mais comuns no sistema de E/S ANSI: *11-39*
Ponteiro para arquivo *11-39*
Abrindo arquivo *11-39*
Valores legais para MODO: *11-40*
Fechando um arquivo *11-40*
Funções fread () e fwrite () *11-41*
Programa exemplo: *11-41*
Manutenção de arquivo sequencial em C *11-43*
12. ALOCAÇÃO DINÂMICA DE MEMÓRIA **12-47**
13. BIBLIOGRAFIA **13-51**
LIVROS: 13-51
WEB SITES: 13-51
ANEXO 1 - O PRÉ-PROCESSADOR C **13-52**
#DEFINE 13-52
#ERROR 13-53
#INCLUDE 13-53
#IF #ELSE #ELIF #ENDIF 13-53
#IFDEF E #IFNDEF 13-54
#UNDEF 13-55
#LINE 13-55
OS OPERADORES # E ## 13-56
#PRAGMA 13-56

1. Apresentação

São várias as linguagens de programação disponíveis no mercado de informática hoje. Cada uma com sua aplicação específica. Podemos citar duas das linguagens mais comumentes usadas nos micros: BASIC e PASCAL. BASIC tem como vantagem a facilidade de se aprender, e é assim muito utilizada por quem deseja iniciar o estudo de programação. PASCAL, por sua vez é bastante estruturada e aberta e facilita desenvolvimento de grandes softwares em algumas áreas. Muito utilizada em programação científica, a linguagem PASCAL tem como principais usuários, programadores que já tem algum conhecimento.

Estas duas linguagens têm várias descendências. Pode-se citar COMAL, que é uma linguagem híbrida que une as principais características de BASIC e PASCAL. Pode-se citar ainda MODULA2 que é uma linguagem muito parecida com pascal, mas que carrega como característica, a facilidade de se desenvolver programas concorrentes (que ocorrem em tempo compartilhado com outros programas, concorrendo a CPU). Atualmente, temos a linguagem DELPHI que é um ambiente de desenvolvimento visual, baseado na sintaxe OBJECT PASCAL, isto é, um compilador pascal orientado para objetos.

Com uma aplicação completamente diversa de PASCAL e BASIC, pode-se citar a linguagem CLIPPER, que é uma versão compilada de DBASE PLUS. Esta linguagem é mais voltada para aplicações de banco de dados. CLARION é uma linguagem que tem uma função similar a esta, mas com uma facilidade muito maior de operação.

Num outro ponto, temos linguagens como LISP e PROLOG. A primeira é muito utilizada em programação matemática, para prova de teoremas e etc... A segunda é bastante utilizada para desenvolvimento de sistemas inteligentes.

Mais recentemente, surgiram as linguagens voltadas para aplicações WEB, isto é, linguagens com facilidades para desenvolvimento de aplicações distribuídas, voltadas para internet. Podemos citar JAVA, que é uma linguagem semi compilada e semi interpretada. O resultado da compilação de um programa JAVA é um código objeto para JVM (máquina virtual JAVA), denominado bytecode. Para que este código possa ser executado, é necessário um programa interpretador desta máquina virtual. Desta forma, qualquer aplicação JAVA pode ser executada em qualquer máquina, desde que esta tenha JVM instalado e habilitado nos navegadores para internet.

Não podemos comparar todas estas linguagens entre si e dizer qual é a melhor. Todas estas linguagens são ferramentas para a tarefa de programação. Cada uma com suas características próprias satisfazem a uma determinada aplicação ao qual se destina. Não existe aquela linguagem que é a melhor, em qualquer situação todas tem vantagens e desvantagens e sua utilização e ditada pela aplicação e pela facilidade que o usuário tem em operá-la.

2. A Linguagem C

A linguagem C foi inventada e implementada por Dennis Ritchie, resultado do desenvolvimento de uma linguagem mais antiga chamada BCPL.

Linguagem BCPL --> Linguagem B --> Linguagem C

A linguagem C e o Sistema Operacional UNIX surgiram praticamente juntos e um ajudou o desenvolvimento do outro. A versão K&R (Brain Kernigan e Dennis Ritchie) já era fornecida junto com o Sistema Operacional UNIX versão 5.

Existem várias implementações de C, mas para garantir compatibilidade entre estas implementações foi criado um padrão ANSI para C. ANSI é a abreviatura de American National Standard Institute, trata-se de um instituto americano que se encarrega da formulação de normas em diversos setores técnicos, incluindo os relativos aos computadores.

Recentemente foi lançada uma versão C orientada a objeto (usando a filosofia OOP), o C++ (inicialmente chamado C com classes). Esta versão continua vinculada ao padrão ANSI mas inclui ferramentas novas que facilitam o desenvolvimento de grandes sistemas usando C.

Características Principais da Linguagem C

⊗ É uma linguagem de médio nível, ou seja, é uma linguagem que não está tão afastada da máquina.

Linguagens de Alto nível - ADA, Modula-2, Pascal, Cobol, Basic.
Linguagens de Médio nível- C, Forth, Macro-Assembler
Linguagem de Baixo nível - Assembler

⊗ C é uma linguagem portável, ou seja, um programa desenvolvido para uma máquina pode facilmente migrar para outra sem muita alteração.

⊗ código gerado para um programa C é mais resumido (otimizado)

⊗ C tem somente 32 palavras chaves
27 do padrão K&R
5 do padrão ANSI

⊗ C é uma linguagem estruturada.

1. Não admite declaração de função dentro de função.
2. Admite malhas (laços) de repetição como for.
3. Admite que você indente os comandos.
4. Admite blocos de comandos (comandos compostos).
5. Desaconselha o uso de goto.

Linguagens estruturadas - Pascal, ADA, C, Modula-2
Linguagens não estruturadas - Fortran, Basic, Cobol

⊗ Existem versões compiladas e interpretadas de C, as versões compiladas são mais rápidas mas as versões interpretadas são preferidas quando se quer depurar um programa.

⊗ As palavras chaves de C:

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

- ⊗ Todas as palavras chaves de C são minúsculas. A linguagem C faz distinção entre palavras maiúsculas e minúsculas.
  else <> ELSE

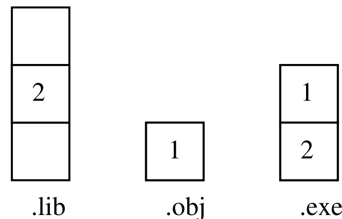
- ⊗ Um programa em C é composto de uma ou mais funções. Todo programa em C tem a função main () que é o corpo principal do programa.

Biblioteca de funções e Linkedição

Além das palavras chaves, todo compilador C vem com uma biblioteca padrão de funções que facilitam aos programadores a realização de tarefas como ler uma variável qualquer, comparar cadeia de caracteres, obter a data do sistema operacional. Existe uma biblioteca básica definida pelo padrão ANSI, mas alguns compiladores em C vem com algumas funções extras para aplicações gráficas por exemplo.

As bibliotecas devem ser incluídas nos programas que fazem uso dela e as funções usadas pelo programa serão LINKEDITADOS (ligados) ao programa.

Simplificação do processo de Linkedição:



Anatomia de um Programa em C

Todo programa em C segue uma estrutura básica, conforme o esquema apresentado abaixo:

```
[inclusão de bibliotecas]
[declaração de constantes e variáveis globais]
[declaração dos protótipos de funções]
void main (void)
{
  [corpo do programa principal]
}
[implementação das funções do programa]
```

Um programa exemplo:

```
/* Exemplo 1 - Um programa bem simples */
#include <stdio.h>
  main ()
{
 puts ("Bem vindo a linguagem C");
}
```

<u>Explicação:</u> A execução deste programa faz a apresentação do texto: <u>Bem vindo a linguagem C</u>. Na primeira linha temos o uso da macro **#include <stdio.h>** que faz a inclusão de algumas funções de biblioteca utilizadas para ler e escrever (in/out). O corpo principal do programa está organizado dentro da função **main**. Esta função deve existir em todo programa C completo e é a partir desta função que começa a ser executado o programa C. A palavra **void** antes de **main** e entre parênteses, significa que a função **main** não devolve valor, nem espera valores (não tem parâmetros). Dentro da função **main** está sendo usada a função utilizada para escrever strings: **puts** e dentro o valor string que deve ser escrito. A função **puts** tem seu protótipo definido dentro do arquivo de cabeçalho **stdio.h**, daí a necessidade de fazer a inclusão destas funções neste programa.

3. Tipos de dados

Os tipos básicos de dados em C

Há 5 tipos básicos em C:

Tamanho	Tipo	Explicação
1 byte	char	caracter
2 bytes	int	inteiro
4 bytes	float	ponto flutuante (real)
8 bytes	double	ponto flutuante precisão dupla
0 bytes	void	ausência de tipo

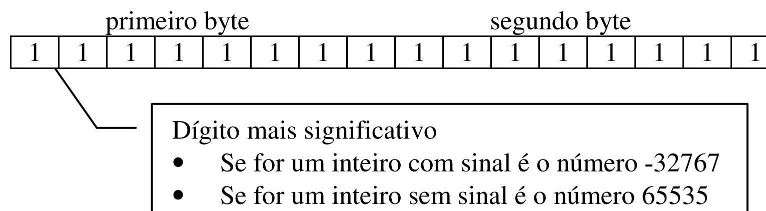
Os modificadores de tipo

Um modificador de tipo é usado para alterar o significado de um tipo: signed, short, long e unsigned.
Exemplo de uso de modificadores:

Tipo	Tamanho (em bits)	Faixa mínima
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
long int	32	-2.147.483.648 a 2.147.483.647

Obs: A diferença entre inteiros com sinal (signed) e sem sinal (unsigned) é a maneira como o bit mais significativo é interpretado.

Representação esquemática:



Nome de Identificador

Regras gerais:

- nome de um identificador deve começar com uma letra ou sublinhado e os caracteres subsequentes devem ser letras, número ou sublinhado.
- Letras maiúsculas e minúsculas são tratados diferentemente em C
- Um identificador não pode ser igual a uma palavra chave e não deve se igual a nome de função de um programa.

Exemplo de identificadores:

CORRETO INCORRETO

```
contador        1contador
teste23         ola!
balanco_alto    balanco...alto
_variavel
```

Variável

É uma posição de memória com um nome que pode ser modificada pela lógica do programa. Deve ser declarada antes de ser usada.

FORMA GERAL :

```
tipo lista_de_variáveis;
```

Exemplos :

```
int i,j,l;
double total, mediageral;
```

As variáveis são de 3 tipos dependendo do local em que foram declaradas:

A. Variável local
B. Variável formal
C. Variável global

A. Variável local:

É uma variável que só existe dentro do bloco no qual ela foi criada. Um bloco (comando composto) em C começa com { (abre chave) e termina com } (fecha chave).

Exemplo:

```
void main (void)
{
  int i;
}
```

B. Parâmetros formais

São os parâmetros passados para uma função que usa parâmetros. São tratados como variáveis locais a função e sua declaração é feita depois do nome da função e dentro dos parênteses.

```
/* Exemplo 2 - Parametro formal */

#include <stdio.h>
  void
EscreveCaracter (char c)
{
 putchar (c);
}

main ()
{
 EscreveCaracter ('a');
}
```

OBS: - Se o tipo dos parâmetros formais forem diferentes dos tipos dos argumentos passados para a função o programa em C não vai parar mais isto pode incorrer em ERRO.

- C não verifica o número de argumentos passados para a função.
- A funcão de biblioteca **putchar** é utilizada para escrever um caracter.

C. Variaveis Globais

São as variáveis que estão disponíveis ao programa inteiro, qualquer bloco do programa pode acessá-las sem erro. Estas variáveis devem ser declaradas fora de todas as funções e no começo do programa principal.

```
/* Exemplo 3 - Variavel global */

#include <stdio.h>

char c; /* Declaracao da variavel global */

main ()
{
  c = 'a';
  putchar (c);
}
```

OBS:

- O uso de variáveis globais deve ser evitado porque estas ocupam espaço na memória durante toda a execução do programa.
- Deve ser explorado o uso de funções parametrizadas para melhorar a organização e a estruturação de programas em C.

Inicialização de Variáveis

FORMA GERAL:

```
   tipo nome-da-variável = constante;
```

Exemplo:

```
        char ch = 'a';
        int first = 0;
        float valor = 12.56;
```

As variáveis ch, first e valor são declarados e já recebem um valor inicial.

4. Entrada e Saída pelo Console

Existe uma série de funções de entrada e saída nas bibliotecas da linguagem C. No entanto o enfoque das operações de entrada e saída é único, diferente de todas as linguagens de programação.

Temos em C funções para:

- ⊗ Ler e escrever caracteres - getchar () e putchar ()
- ⊗ Ler e escrever strings - gets () e puts ()
- ⊗ Entrada/Saída formatada - scanf () e printf () entre outras funções.

Lendo e Escrevendo caracteres

- ⊗ int getchar (void) - A função getchar espera até que seja digitado um caracter no teclado, então mostra o caracter e sai do processo de entrada de dados.

- ⊗ int putchar (int c) - A função putchar escreve um caracter a partir da posição corrente do cursor.

```
/* Exemplo 4 -
     Programa que le caracteres e escreve em caixa invertida.
     Le maiuscula, escreve minuscula e vice-versa */

#include <stdio.h>
#include <ctype.h>

  main ()
{
  char ch;
  puts ("\nDigite '.' e <ENTER> para terminar ");
  do
    {
      ch = getchar ();
      if (islower (ch))
      ch = toupper (ch);
      else
      ch = tolower (ch);
      putchar (ch);
    }
  while (ch != '.');
}
```

<u>Alternativas para getchar</u>

Funções:

getch () - lê um caracter do teclado sem ecoar o caracter na tela;

getche () - mesma função do getchar () para ambiente interativo.

OBS: Algumas implementações getchar foram desenvolvidas para ambiente UNIX, o qual armazena os dados digitados num buffer do terminal até que seja digitado um <ENTER> quando então transfere os dados dos buffer para o programa que espera a leitura. getche () não usa este buffer.

As funções getch () e getche () se encontra definidas no arquivo de cabeçalho **conio.h**. Este arquivo de cabeçalho não está disponível em biblioteca da C para UNIX, LINUX, etc... Esta biblioteca não é do padrão da linguagem C.

Lendo e Escrevendo string

- ⊗ char *gets (char *str) - Lê um string de caracteres digitado pelo teclado até que seja digitado <ENTER>. O caracter da tecla <ENTER> não fará parte do string, no seu lugar é colocado o caracter de fim de cadeia ('\0'). Por questão de segurança, o uso do comando *gets* é desaconselhado em ambiente UNIX.

⊗ int puts (char *s) - Escreve um string na tela seguido por uma nova linha ('\n'). É mais rápido que a função printf, mas no entanto só opera com argumentos string' s.

```
/* Exemplo 5 -
     Programa que le um nome e escreve
     uma mensagem com o nome digitado. */
#include <stdio.h>
main ()
{
  char nome[20];
  puts ("\nEscreva seu nome:");
  fgets (nome, 20, stdin);    /* gets (nome); */
  printf ("Oi, %s", nome);
}
```

Entrada e Saída formatada

Saída formatada:

printf ("série de controle", listadeargumentos) - função para impressão formatada na tela.

onde:

- ⊗ "série de controle" - É uma série de caracteres e comandos de formatação de dados que devem ser impressos.
- ⊗ listadeargumentos - Variáveis e constantes que serão trocados pelos formatos correspondentes na série de controle.

```
/* Exemplo 6 - Programa de saida formatada */
#include <stdio.h>

main ()
{
  printf ("\n----+----+");
  printf ("\n%-5.2f", 123.234);
  printf ("\n%5.2f", 3.234);
  printf ("\n%10s", "hello");
  printf ("\n%-10s", "hello");
  printf ("\n%5.7s", "1234567890");
}
```

Será apresentado na tela:

```
----+----+
123.23
 3.23
     hello
hello
1234567
```

Observações:

- A constante de barra invertida '\n' significa salto para uma nova linha (new line).
- O formato "%-5.2f" indica que o número em ponto flutuante (f) deve ser apresentado com no mínimo 5 caracteres, 2 dígitos para a parte fracionária do número e deve ser justificado a esquerda.
- O formato "%5.2f" indica a mesma coisa só que justificado a direita.
- O formato "%10s" indica que o string (s) deve ser apresentado em 10 espaços justificado a direita.
- O formato "%-10s" indica a mesma coisa só que justificado a esquerda.
- O formato "%5.7s" indica que o string (s) deve ser apresentado com pelo menos 5 caracteres e não mais que 7 caracteres.

Alguns comandos de formato do printf

código	formato
%c	caracter
%d	inteiro com sinal
%i	inteiro com sinal
%e	notação científica
%E	notação científica
%g	%e ou %f (+curto)
%G	%E ou %f (+curto)
%%	caracter %

código	formato
%o	octal
%s	string
%u	inteiro sem sinal
%x	hexadecimal minúsculo
%X	hexadecimal maiúsculo
%p	endereço
%n	ponteiro de inteiro

```
/* Exemplo 7 - Tabela de quadrado e cubo de numeros */

#include <stdio.h>
main ()
{
  int i;
  printf ("\n\n %8s %8s %8s", "x", "x^2", "x^3");
  for (i = 1; i <= 10; i++)
    printf ("\n %8d %8d %8d", i, i * i, i * i * i);
  printf ("\n");
}
```

Será apresentado:

```
        x      x^2      x^3
        1        1        1
        2        4        8
        3        9       27
        4       16       64
        5       25      125
        6       36      216
        7       49      343
        8       64      512
        9       81      729
       10      100     1000
```

Entrada formatada:

scanf ("série de controle", listadeendereçodosargumentos): função para leitura formatada.
OBS:
Os formatos usados na série de controle da função scanf é semelhante aos formatos usados na função printf.

```
/* Exemplo 8 - Uso da funcao scanf */
#include <stdio.h>
main ()
{
  char nome[20];
  printf ("\nDigite um nome: ");
  scanf ("%s", nome);
  printf ("Oi, %s!\n\n", nome);

}
```

Obs:

⊗ -Este programa pede um nome e uma idade e monta uma frase do tipo: "João, voce tem 26 anos".

⊗ -Detalhes como a declaração **char nome [20];** e da chamada da função **scanf ("%d", &idade);** serão esclarecidos quando descutirmos os assuntos de strings em C e passagem de parâmetros por referência em funções.

5. Constantes, Operadores e Expressões

São valores fixos que não podem ser alterados por um programa. Uma constante pode ser de quatro tipos:

⊗ caracter (char): ' a' , ' %' , envolvidos por aspas simples (apóstrofe);
⊗ inteiro (int): 10, -97, números sem componente fracionário.
⊗ real (float,double): 11.12, número com componente fracionário.
⊗ strings: "universidade", caracteres envolvidos por aspas dupla.

Constantes Hexadecimais e Octais

<u>Constante Hexadecimal</u> - Número na base numérica 16. Os dígitos nesta base numérica são 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F , ou seja, temos 16 dígitos. Exemplos: 1A = 26 em decimal, FF = 255 em decimal.

<u>Constante Octal</u> - Número na base numérica 8. Os dígitos nesta base numérica são 0,1,2,3,4,5,6,7 ou seja, temos 8 dígitos. Exemplos: 11 = 9 em decimal, 24 = 20 em decimal.

Para que possamos especificar ao compilador C que uma constante é Hexadecimal ou Octal usamos de dois símbolos: 0x, 0 antes do número Hexadecimal e Octal respectivamente.

`0xconstante` especifica que um número é Hexadecimal
`0constante` especifica que um número é Octal.

```
/* Exemplo 9 - Constantes Hexadecimais e Octais */

#include <stdio.h>

main ()
{
  int h = 0xff;
  int o = 024;
  printf ("HEX  : %X = DECIMAL : %d\n", h, h);
  printf ("OCTAL: %o = DECIMAL : %d\n", o, o);
}
```

OBS: Os padrões %x, %d e %o serão trocados pelos valores das variáveis h, h e o nos formatos hexadecimal, decimal e octal respectivamente. Na tela aparecerá:

HEX : FF = DECIMAL : 255
OCTAL: 24 = DECIMAL : 20

Constantes de Barra Invertida

Para descrever alguns caracteres da tabela ASCII que não podem ser definidos usando as aspas simples existem em C, alguns códigos de barra invertida que representam estes caracteres.Exemplo:

Código	Significado
\b	retrocesso (BS)
\f	alimentação de formulário (FF)
\n	nova linha
\r	retorno de carro
\t	tabulação horizontal
\"	caracter "
\'	caracter '
\0	nulo (fim de cadeia de caracteres)
\\	barra invertida
\v	tabulação vertical
\a	beep de alerta
\N	constante octal (N - número octal)
\xN	constante hexadecimal

Operadores

Existem 5 níveis de operadores em C: atribuição, aritméticos, relacionais, lógicos e bit a bit.

Operação de Atribuição

A linguagem C usa o sinal de igual (=) como operador de atribuição. O formato geral de uma atribuição em C é o seguinte:

```
     nome_da_variável = expressão;
```

A novidade em C é que o operador de atribuição pode entrar em qualquer expressão válida de C.

Conversão de tipos na Atribuição

O valor do lado direito da atribuição é convertido no tipo do lado esquerdo. Se o tamanho em bytes do tipo do lado esquerdo for menor do que o valor atribuido alguma informação será perdida (Os bits mais significativos do valor serão desprezados)

Exemplo:

```
int   X  = 10;
char  Ch = 'a';
float F  = 12.5;

void main (void)
{
   Ch = X; --> Os bit + significativos de X são ignorados
   X = F;  --> X recebe a parte inteira de F
   F = Ch; --> O valor inteiro de Ch é convertido para
                ponto flutuante
   F = X;  --> Converte o valor inteiro de X para seu
                formato em ponto flutuante
}
```

Atribuições Múltiplas

Pode-se atribuir um mesmo valor para diversas variáveis em C. Assim:

X = Y = Z = 0;

Operadores Aritméticos

Os operadores aritméticos de C são os seguintes:

Operador	Ação
-	subtração, também o menos unário
+	Adição
*	Multiplicação
/	Divisão
%	módulo (resto) da divisão inteira
++	Incremento
--	Decremento

O uso destes operadores é igual ao uso nas outras linguagens exceto os de incremento e decremento

Incremento (++) e decremento (--)

x = x + 1 e ++x são instruções equivalentes em C

x = x - 1 e --x também o são

No entanto os operadores de incremento e decremento podem preceder ou suceder o operando assim:

++x e x++

Existe uma diferença básica entre este dois usos:
- ⊗ `++x` -executa a operação de incremento/ decremento antes de usar o valor do operando.
- ⊗ `x--` -executa a operação de incremento/ decremento depois de usar o valor do operando.

```
/* Exemplo 10 - Operador de Incremento */

#include <stdio.h>

main ()
{
  int X, Y;
  X = 10;
  Y = ++X;
  printf ("X= %d Y= %d\n", X, Y);
  X = 10;
  Y = X++;
  printf ("X= %d Y= %d\n", X, Y);
}
```

Será apresentado na tela
X= 11 Y= 11
X= 11 Y= 10

Precedência dos Operadores Aritméticos

maior ↓ menor

Operador	Significado
++, --	incremento, decremento
+, -	Unários
*, /, %	multiplicação, divisão e módulo
+, -	adição e subtração

Obs:
- ⊗ Operadores de mesma precedência são tratados primeiro da esquerda para direita.
- ⊗ Para alterar a ordem de precedência deve-se usar parênteses.

Operadores Relacionais e Lógicos

Operadores relacionais tratam das relações entre valores (>, < etc...) e os operadores lógicos fazem a composição de expressões relacionais.

Obs:
- ⊗ VERDADEIRO EM C = qualquer valor diferente de zero.
- ⊗ FALSO EM C = valor zero

Tabela Verdade dos Operadores lógicos

a	b	a && b	a ll b	!a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

onde:
- ⊗ &&- é o and (e lógico) - a&&b le-se "a e b"
- ⊗ ll- é o or (ou lógico) - allb le-se "a ou b"
- ⊗ ! - é o not (negação) - !a le-se "negação de a"

```c
/* Exemplo 11 - Operadores Logicos */

#include <stdio.h>

main ()
{
  int a = 1, b = 0, c;
  c = a && !b || !a && b;
  if (c)
    puts ("1");
  else
    puts ("0");
}
```

Precedência dos Operadores Lógicos e Relacionais

maior ↓ menor

<table>
<tr><th>Operador</th><th>Significado</th></tr>
<tr><td>!</td><td>negação</td></tr>
<tr><td>>, >=, <, <=, ==, !=</td><td>menor, menor igual, maior, maior igual, igual, diferente</td></tr>
<tr><td>&&</td><td>e lógico</td></tr>
<tr><td>||</td><td>ou lógico</td></tr>
</table>

Obs:

⊗ Todo resultado de uma expressão lógica e/ou relacional é o número 0 ou diferente de 0 (zero). Então será válido em C o seguinte programa:

```c
/* Exemplo 12 - Resultado de uma operacao relacional */

#include <stdio.h>
  main ()
{
  int X = 1;
  printf ("%d", X > 10);
}
```

Operadores Bit a Bit

São operadores que possibilitam a manipulação dos bits numa palavra ou byte da linguagem C. As operações bit a bit são AND, OR, XOR, complemento, deslocamento a esquerda, deslocamento a direita. Essas operações atuam nos bits dos operandos. São eles:

<table>
<tr><th>Operador</th><th>Ação</th></tr>
<tr><td>&</td><td>and binário</td></tr>
<tr><td>|</td><td>or binário</td></tr>
<tr><td>^</td><td>xor binário</td></tr>
<tr><td>~</td><td>complemento</td></tr>
<tr><td>>></td><td>deslocamento à direita</td></tr>
<tr><td><<</td><td>deslocamento à esquerda</td></tr>
</table>

Os operadores bit a bit são muito usados em programas "drivers" de dispositivos (modem, scanner, impressora, porta serial) para mascarar bits de informação nos bytes enviados por estes dispositivos.

```
/* Exemplo 13 - Traducao de hexadecimal para binario */

#include <stdio.h>

main ()
{
  int h = 0x7A, i, m = 0x80;
  for (i = 1; i <= 8; i++)
    {
      if (m & h)
      putchar ('1');
      else
      putchar ('0');
      m = m >> 1;
    }
}
```

Obs:

- ⊗ O número hexadecimal 7A foi vasculhado por este programa para ser traduzido para o seu equivalente binário. Operações bit a bit foram necessárias para encontrarmos os bits acesos (1) e apagados (0) do byte h.
- ⊗ Será apresentado na tela: 01111010.

Expressões em C

Uma expressão é a composição de operadores lógicos, aritméticos e relacionais, variáveis e constantes. Em C uma expressão é qualquer combinação válida destes elementos.

Observações:

- ⊗ Conversão de tipos em expressões: Numa expressão, o compilador C converte todos os operandos no tipo do maior operando (promoção de tipo).

Exemplo:

```
     char ch;         result = (ch / i) + (f * d) - (f + i)
     int i;                        \ /       \ /       \ /
     float f;                      int      double     float
     double d;                        \      /         /
                                       double      /
                                             \   /
                                             double
```

- ⊗ Pode-se forçar que o resultado de uma expressão seja de um tipo específico através do uso de um molde (CASTS) de tipo. Assim:

```
      (float) x / 2;      o resultado de  X dividido por 2 é um
                          um valor real
```

- ⊗ Espaços e parênteses podem ser usados a vontade em expressões "C" sem que isto implique em diminuição de velocidade de execução da expressão. O uso de parênteses e espaço é aconselhado no sentido de aumentar a legibilidade do programa. O parênteses pode mudar a ordem de precedência dos operadores.

- ⊗ Abreviações em C: A linguagem C admite que algumas atribuições sejam abreviadas.

Exemplo:

```
      X = X + 10; é o mesmo que X += 10
```

Forma Geral:

`variável = variável operador expressão`

é o mesmo que `variável operador= expressão.`

6. Comandos de controle de programa

O padrão ANSI divide os comandos em C nos seguintes grupos:

⊗ seleção : if , switch
⊗ iteração : while, do-while, for
⊗ desvio : break, continue, goto e return
⊗ expressão : expressões válidas em C
⊗ bloco : blocos de código entre { e }

Comando de Seleção

Comando if (Seleção simples)

Forma Geral:

```
    if (expressão)
      comando1;
    else
      comando2;
```

Obs:

⊗ A expressão que controla a seleção deve estar entre parênteses.
⊗ comando1, que pode ser um comando simples ou um comando em bloco, será executado se o resultado da expressão for diferente de zero (VERDADEIRO em C)
⊗ comando2 será executado se o resultado da expressão for igual a zero.
⊗ Diferente de PASCAL, em C temos um ponto e vírgula antes da palavra reservada ELSE.

```
/* Exemplo 14 - Comando de selecao simples - if */

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

main ()
{
  int nro, palp, stime;
  long ltime;

  /*-- obtem a hora do sistema --*/
  ltime = time (NULL);
  stime = (unsigned) ltime / 2;

  /*-- estabelece um ponto de partida   --*/
  /*-- para sequencia gerada por rand() --*/

  srand (stime);

  /*-- rand() gera numeros aleatorios --*/
  nro = rand () % 10;
  puts ("\nAdivinhe o numero magico [0..9]");
  scanf ("%d", &palp);
  if (palp == nro)
    puts ("\nVoce ACERTOU !");
  else
    printf ("\nVoce errou, %d era o numero magico!!\n", nro);
}
```

if's aninhados

As regras para aninhamento de if's é igual para outras linguagens de programação. O comando else em C se refere ao if mais próximo (acima deste) que está no mesmo bloco.

Exemplo:

```
          if (i) {
                    if (j) comando1;
                    if (k) comando2;
                    else comando3;
                 }
          else comando4;
```

Obs:

⊗ a identação de IF's e ELSE's fica a cargo e critério de cada programador. Não existe uma regra em C.

O operador ? - Alternativo para o if-else

Em C podemos usar o operador ã para substituir a construção IF-ELSE da seguinte forma:

```
          if (condição) expressão1; else expressão2;
```

pode ser substituido por

```
          condição ? expressão1 : expressão2
```

```
/* Exemplo 15 - If alternativo */

#include <stdio.h>

main ()
{
  int X = 10, Y;
  Y = X > 9 ? 100 : 200;
  printf ("\n%d", Y);
}
```

Obs:

⊗ operador ? é chamado de operador ternário (trabalha com três operandos)

⊗ expressão1 e expressão2 devem ser expressões simples e nunca um outro comando de C (tipo printf, scanf ou outro qualquer).

Obs:

⊗ A condição do comando de seleção pode ser qualquer número igual ou diferente de zero. O seguinte programa é válido em C:

```
/* Exemplo 16 - Condicao de selecao */

#include <stdio.h>

main ()
{
  int a, b;
  puts ("\nDigite dois numeros separados por espaco : ");
  scanf ("%d%d", &a, &b);
  if (b)
    printf ("\ndivisao : %d\n", a / b);
  else
    printf ("\nErro: divisao por zero\n");
}
```

Comando switch - Seleção Múltipla

Forma Geral:

```
          switch (expressão)
          {
            case constante1:  sequência de comandos1;
                              break;
            case constante2:  sequência de comandos2;
                              break;
                  . . .
            case constanteN:  sequência de comandosN;
                              break;
            default : sequência de comandos;
          }
```

Obs:
⊗ Se o resultado da expressão for igual a constante1, será executado a sequência de comandos1;
⊗ Se o resultado da expressão for igual a constante2, será executado a sequência de comandos2;
⊗ A sequência de comandos após a palavra chave default será executado se o resultado da expressão não for igual a nenhuma das constantes.
⊗ comando break após cada sequência de comandos põe fim ao comando switch. Se não for usado, todas as instruções contidas abaixo da sequência de comandos serão executadas.

```
/* Exemplo 17 - Comando de selecao multipla */

#include <stdio.h>
main ()
{
  char c;
  printf ("\nDigite um numero: ");
  c = getchar ();
  printf ("\n");
  switch (c)
    {
    case '1':
      printf ("um\n");
    case '2':
      printf ("dois\n");
    case '3':
      printf ("tres\n");
    }
}
```

Obs:
⊗ Se i = ' 1' então será apresentado na tela um dois tres
⊗ Se i = ' 2' então será apresentado na tela dois tres
⊗ Se i = ' 3' então será apresentado na tela tres
⊗ Como no caso do comando IF o comando SWITCH também pode ser aninhada em vários níveis.

Exercício: Modifique o exemplo anterior inserindo o comando break; após cada printf dentro do comando switch e verifique os resultados.

Comandos de Iteração

Comandos de iteração (malha ou laço de repetição) são usados para permitir que um conjunto de instruções sejam executadas enquanto uma determinada condição seja verdadeira. Os comandos de iteração em C são : for, while, do-while

A malha (laço) de repetição FOR

Forma geral:

```
          for (inicialização; condição; incremento)
               comando;
```

onde:

- ⊗ inicialização - É geralmente um comando de atribuição que coloca um valor inicial para as variáveis de controle do laço for.
- ⊗ condição - É uma condição relacional, lógica ou aritmética que define até quando a iteração pode continuar (condição de permanência da repetição)
- ⊗ incremento - Define como a variável de controle do laço varia cada vez que o laço é repetido.

```
/* Exemplo 18 - Contagem com o comando de repeticao for */

#include <stdio.h>

main ()
{
  int X;
  for (X = 1; X < 100; X++)
    printf ("%-4d", X);
  printf ("\n");
}
```

Explicação: Assim que o programa entra na repetição for a variável inteira X assume o valor 1. Dentro da repetição a única instrução é escrever o valor de X, que irá variar de um em um (X++) enquanto X é menor que 100.

Exercício:

Fazer este mesmo programa contar de 10 em 10, de 0 até 1000.

Obs:

- ⊗ A malha de repetição for pode repetir um conjunto de comandos (comandos em bloco).

```
/* Exemplo 19 - For com comando em bloco */

#include <stdio.h>

main ()
{
  int X, Y;
  for (X = 100; X != 65; X -= 5)
    {
      Y = X * X;
      printf ("O quadrado de %3d e' %5d\n", X, Y);
    }
}
```

- ⊗ laço de repetição for pode trabalhar com mais do que uma única variável. Pode-se inicializar e incrementar mais do que uma variável no escopo da repetição for.

```
/* Exemplo 20 -

     Repeticao for com 2 variaveis de controle */

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

main ()
{
  char s[8] = "Joaquim";
  int i, j;
  for (i = 0, j = strlen (s) - 1; i < strlen (s); i++, j--)
    {
     printf ("%c - %c\n", s[i], s[j]);
    }
}
```

Os comandos clrscr (); e gotoxy (); são particulares da versão TURBO C, não existem no padrão ANSI. Mas toda versão C tem uma instrução equivalente a estas.

⊗ Parte das definições do laço <u>for</u> não são obrigatórios. Com isto conseguimos por exemplo um laço infinito:

```
/* Exemplo 21 - Repeticao for infinita

   OBS: DIGITE CTRL-C para terminar este programa */

#include <stdio.h>

main ()
{
  for (;;)
    printf ("Laco sem fim!");
}
```

⊗ Existem também os casos em que uma malha de repetição não irá executar nada (laço for sem corpo)

```
/* Exemplo 22 - Outro exemplo de repeticao */

#include <stdio.h>

  main ()
{
  long int i;
  puts ("Espere um pouco...");
  for (i = 1; i < 500000000; i++);
  puts ("Terminei !");
}
```

<u>O laço de repetição while</u>

forma geral:

```
        while (condição) comando;
```

onde:

⊗ <u>condição:</u> É qualquer expressão onde o resultado é um valor igual ou diferente de zero.
⊗ <u>comando:</u> É um comando vazio, um comando simples ou um comando em bloco (comando composto).

```c
/* Exemplo 23 - Repeticao while */

#include <stdio.h>

main ()
{
  char ch = 0;
  while (ch != 's')
    ch = getchar ();
  puts ("\nFim");
}
```

O programa ficará executando a malha while até que o usuário digite o caracter 's' minusculo. Antes a variável ch é inicializada com nulo (zero).

Como no caso da instrução for a malha de repetição while também pode ser declarada sem corpo. Então o primeiro programa apresentado também poderia ser escrito assim:

```c
/* Exemplo 24 - Repeticao while sem corpo */

#include <stdio.h>

  main ()
{
  while (getchar () != 's');
  puts ("\nFim");
}
```

No caso do escopo da instrução while já se está fazendo a leitura de um caracter do teclado com eco (getchar) e verificando se o caracter digitado é a letra 's' minusculo.

O laço do-while

Ao contrário dos laços while e for que testam a condição de permanência no começo, o laço do-while testa a condição de permanência no final.

forma geral :

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

O comando dentro das chaves será executado enquanto a condição for satisfeita.

```c
/* Exemplo 25 - Repeticao do-while */

#include <stdio.h>

main ()
{
  int i = 1;
  do
    {
      printf ("%4d", i);
      i++;
    }
  while (i <= 100);
}
```

Exercício: Verifique o que faz este programa.

Comandos de desvio

A linguagem C tem quatro comandos de desvio que realizam desvio incondicional: return, goto, break e continue.

Comando return

Obriga o programa a retornar da execução de uma função. Se o programa está executando uma função e encontra o comando return, o programa retornará ao ponto em que foi chamada a função.

Forma Geral :

```
      return expressão
```

```
/* Exemplo 26 - Comando de desvio return */

#include <stdio.h>

int
menos (int a, int b)
{
  return (a - b);
}

main ()
{
  int x, y;
  puts ("\nDigite dois inteiros separados por espaco : ");
  scanf ("%d%d", &x, &y);
  printf ("%d menos %d e' igual a %d\n\n", x, y, menos (x, y));
}
```

Comando goto

Um comando pouco necessário e não aconselhado em programas C. Sua função é saltar para um rótulo.

```
/* Exemplo 27 - Comando de desvio goto */

#include <stdio.h>

void
main (void)
{
  int i = 1;
loop:
  printf ("%4d", i);
  i++;
  if (i <= 100)
    goto loop;
}
```

Comando break

Tem dois usos: terminar um case da instrução switch, ou terminar uma malha de repetição evitando o teste de condição de permanência.

```
/* Exemplo 28 - Comando de desvio break */

#include <stdio.h>
  main ()
{
  int t = 1;
  for (;;)
    {
```

```
      if (t > 100)
      break;
      printf ("%4d", t++);
    }
}
```

Função exit

Assim como eu posso abortar uma malha de repetição com a instrução break, eu também posso abortar um programa com a instrução exit.

Forma geral:

```
      void exit (int código_de_retorno);
```

O valor do código_de_retorno é devolvido ao processo chamador do programa que é normalmente o sistema operacional.

```
/* Exemplo 29 - Comando de desvio exit  */

#include <stdlib.h>  /* Biblioteca onde esta definido exit */
  main ()
{
 exit (10);
}
```

Comando continue

Tem efeito contrário do comando break. Obriga a próxima iteração do laço, pulando qualquer código intermediário. Para o laço for, o comando continue faz com que a condição e o incremento sejam executados.

```
/* Exemplo 30 - Comando de desvio continue */

#include <stdio.h>
#include <string.h>
  main ()
{
  char s[80];
  int i;
  printf ("\nDigite um nome completo: ");
  fgets (s, 79, stdin);
  for (i = 0; i != strlen (s); i++)
    {
      if (s[i] == ' ')
      continue;
      putchar (s[i]);
    }
}
```

⊗ **Explicação:** Será lido um string e este string será percorrido caracter a caracter. Se o caracter for o espaço a impressão do caracter será evitada pelo uso do comando continue, caso contrário putchar será executado.

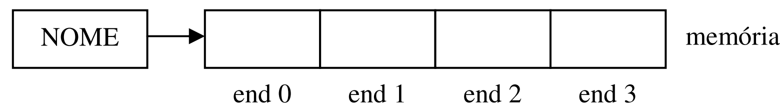
7. Matrizes e Strings

Matriz

Matriz é uma coleção homogênea de dados que ocupam posições contíguas na memória e cujos elementos podem ser acessados através da especificação de índice. O endereço mais baixo na memória corresponde ao primeiro elemento na matriz, o mais alto corresponde ao último elemento.

Exemplo:

```
main () {
   char NOME[3];
}
```



O nome da matriz é um ponteiro para a primeira posição na memória onde está alocada a matriz. No exemplo NOME é um apontador do primeiro elemento da matriz na memória.

Uma matriz pode ser unidimensional ou multidimensional.

Matrizes Unidimensionais

Forma Geral de declaração:

```
      tipo nome_de_variável [tamanho]
```

Exemplo: declaração de um vetor de nome VALOR do tipo real de 100 posições.

```
      float VALOR [100];
```

Todas as matrizes em C começam no índice 0 (zero), então quando declaramos a matriz anterior estamos definindo na memória os dados VALOR [0], VALOR [1], ..., VALOR [99].

Exercícios:

1. Declare um vetor em C de 20 posições de inteiros e de nome NUMEROS.
2. Quais serão os índices válidos para este vetor.

```
/* Exemplo 31 - Programa de usa vetor */

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

main ()
{
  int i, v[100], stime;
  long ltime;
  ltime = time (NULL);
  stime = (unsigned) ltime / 2;
  srand (stime);
  for (i = 100; i > 0; i--)
    v[i - 1] = rand () % 100;
  for (i = 1; i <= 100; i++)
    printf ("%4d", v[i - 1]);
}
```

Obs:

⊗ A linguagem C não verifica limites na matriz, então para o exemplo anterior X [2200] = 20 seria compilado sem erro, mas este programa teria um erro de lógica porque outra posição de memória será afetada de forma indesejável.

Matrizes Multidimensionais

Forma geral:

```
          tipo nome [t1][t2]... [tn];
```

Não são usadas por gastar muito espaço em memória e tempo no cálculo dos índices.

Exemplo:

```
               int matriz [10][6][9][4];
```

Ocupa 10 * 6 * 9 * 4 = 2160 posições de memória. 2160 * 2 = 4320 bytes de memória, considerando que cada inteiro (int) ocupa dois bytes.

Matrizes passadas para função

Pode-se passar uma matriz inteira como argumento de uma função. Na verdade quando fazemos isto estamos passando o endereço da primeira posição da matriz para a função. Assim:

```
               main () {
                    int i [10];
                    func (i);
                    . . .
               }
```

O parâmetro matriz da função func pode ser declarado de uma das 3 formas:

```
     func (int *X)  ou  func (int X [10]) ou func (int X[])
```

```
/* Exemplo 32 - Vetores para funcao */
#include <stdio.h>
void mostra (int *vetor, int tam);
int total (int *vetor, int tam);

main ()
{
  int i, vetA[10], stime;
  long ltime;
  ltime = time (NULL);
  stime = (unsigned) ltime / 2;
  srand (stime);
  for (i = 0; i < 10; i++)
    vetA[i] = rand () % 10;
  mostra (vetA, 10);
  printf ("%d\n", total (vetA, 10));
}

void
mostra (int *vetor, int tam)
{
  int i;
  for (i = 0; i < tam; i++)
    {
      printf ("%d", vetor[i]);
      if (i < tam - 1)
      putchar ('+');
      else
      putchar ('=');
    }
}
```

```
int
total (int *vetor, int tam)
{
  int i, t;
  for (i = 0, t = 0; i < tam; i++)
    t = t + vetor[i];
  return t;
}
```

Strings - (matriz de caracteres)

Uma string em C é uma matriz de caracteres terminada com um fim de cadeia '\0' (caracter nulo). Por isto devemos declarar um string com um caracter a mais do que necessitamos para a nossa aplicação.
Exemplo: string de 10 caracteres

```
      char s [11];   /* índices de (0 - 10) */
```

Para manipulação do tipo string existe na biblioteca C uma série de funções:

Nome	**Função**
strcpy (s1, s2)	copia o string s2 em s1
strcat (s1, s2)	concatena s2 ao final de s1
strlen (s1)	retorna o tamanho do string s1
strcmp (s1, s2)	compara o string s1 com s2
strchr (s1, ch)	retorna o endereço de memória da primeira ocorrência do caracter ch no string s1
strstr (s1, s2)	retorna o endereço de memória da primeira ocorrência do string s1 no string s2.

```
/* Exemplo 33 - Funcoes para strings em C */
#include <stdio.h>
#include <string.h>

main ()
{
  char nome1[40], nome2[40];
  printf ("\nDigite um nome    : ");
  fgets (nome1, 39, stdin);
  printf ("Digite outro nome : ");
  fgets (nome2, 39, stdin);

  puts ("\n**** TESTE DA FUNCAO strlen     ");
  printf ("Tamanho do primeiro nome : %d", strlen (nome1));
  printf ("\nTamanho do segundo  nome : %d", strlen (nome2));

  puts ("\n\n**** TESTE DA FUNCAO strcmp     ");
  if (!strcmp (nome1, nome2))
    puts ("Os nomes sao iguais");
  else
    puts ("Os nomes sao diferentes");

  puts ("\n\n**** TESTE DA FUNCAO strcpy     ");
  strcpy (nome1, "Universidade ");
  printf ("Novo nome: %s ", nome1);

  puts ("\n\n**** TESTE DA FUNCAO strcat     ");
  strcat (nome1, nome2);
  printf ("Nomes concatenados : %s ", nome1);

  puts ("\n\n**** TESTE DA FUNCAO strchr     ");
  if (strchr (nome2, 'a'))
    puts ("Existe 'a' no segundo nome");
```

```
  else
    puts ("Nao existe 'a' no segundo nome");
  puts ("\n**** TESTE DA FUNCAO strstr     ");

  if (strstr (nome2, "ana"))
    puts ("Existe \"ana\" no segundo nome");
  else
    puts ("Nao existe \"ana\" no segundo nome");
}
```

Obs:

⊗ A função strcmp retorna:

- 0 (zero) se s1 e s2 são iguais
- < 0 se s1 é alfabeticamente menor que s2
- > se s1 é alfabeticamente maior que s2

Ponteiros e índices

Matrizes e ponteiros estão intimamente relacionados em C. O nome da matriz é um ponteiro para a primeira posição na memória onde se encontra esta matriz. Então o seguinte trecho de programa é valido em C:

```
     int *p, i[10];
     p = i;
     p [5] = 100;
     *(p + 5) = 100;
```

Inicialização de Matrizes

Forma Geral:

```
          tipo nome [t1]..[tn] = { lista de valores }
```

```
/* Exemplo 34 - Inicializacao de matrizes */
#include <stdio.h>
  main ()
{
  char letra[6] = { 0x18, 0x24, 0x42, 0x7E, 0x42, 0x42 };
  int i, j;
  unsigned int m;
  for (i = 0; i < 6; i++)
    {
      m = 0x80;
      for (j = 0; j < 8; j++)
      {
        if (m & letra[i])
          printf ("9");
        else
          printf (" ");
        m = m >> 1;
      }
      printf ("\n");
    }
}
```

Exercícios:

1. Fazer um programa em C que leia uma matriz de inteiros 3X4 e some os valores de mesma coluna da matriz num vetor de 4 posições de inteiros:
2. Elaborar um programa em C para fazer a multiplicação de duas matrizes 4X4.
3. Elaborar um programa em C para somar duas matrizes 3X3 e apresentar o resultado da soma de maneira tabular.

8. Ponteiros

Ponteiros são variáveis que apontam para algum endereço de memória. São usadas para alocação dinâmica de espaço em memória, ou seja, alocação de espaço de memória que ocorre em tempo de execução dos programas.

Declaração:

```
     tipo *nome_da_variável;
```

Exemplo:

```
char *p; /* o conteúdo do endereço apontado por  p é do tipo caracter */
```

Operadores de ponteiros

& - Devolve o endereço de uma variável.

Exemplo:

```
m = &x; /* m recebe o endereço da variável x */
```

***** - Devolve o conteúdo de memória apontado por um ponteiro.
Exemplo:

```
c = *p;  /* c recebe o conteúdo de memória apontado por p */
```

Expressões usando ponteiro

Atribuição

Exemplo:

```
     void main (void)
     {
       int x;                /* uma variável inteira       */
       int *p1, *p2;         /* dois ponteiros de inteiros */
       p1 = &x;              /* p1 recebe o endereço de x  */
       p2 = p1;       /* p2 recebe o endereço apontado por p1*/
       printf ("%p", p2);    /* escreve o endereço de x    */
     }
```

Aritmética

Duas operações são válidas para ponteiros: Adição e subtração.

```
char *ch = 3000;
ch = ch + 1;
int  *i  = 3000;
i = i + 1;
```

Quando incrementamos um ponteiro estamos na verdade saltando o tamanho do tipo base na memória do computador.

No exemplo anterior o incremento de 1 na variável ch provoca um salto para o próximo byte na memória (ch aponta para char que ocupa um byte na memória). No caso de um incremento de 1 na variável i, provoca um salto de dois bytes a frente na memória (o tipo base int ocupa dois bytes). Para o decremento o mecanismo é semelhante.

Comparação

Pode-se comparar ponteiros normalmente em C. Todas os operadores relacionais são válidos (==, <, !=, etc...).

Exemplo:

```
int *p, *q;
if (p < q) printf ("o endereco apontado por p1 e < que o de q");
```

Outro exemplo:

Programa de pilha usando vetor. Duas operações são válidas com a pilha: empilhar e desempilhar elementos do topo. A função empilha é chamada toda vez que se entra um valor diferente de zero. A função desempilha é chamada quando se entra com o valor zero. O valor -1 termina o programa.

```
/* Exemplo 35 - Uma aplicacao com ponteiros */

#include <stdio.h>

void empilha (int i);
int desempilha (void);

int *TAM, *TOPO, PILHA[10];

main ()
{
  int VALOR;
  puts ("\nEmpilha   : valor diferente de zero");
  puts ("Desempilha: valor igual a zero");
  puts ("Fim       : valor igual a -1");
  TAM = PILHA + 10;
  TOPO = PILHA - 1;

  do
    {
      printf ("Digite um valor : ");
      scanf ("%d", &VALOR);
      if (VALOR != 0)
      empilha (VALOR);
      else
      printf ("Desempilhei : %d\n", desempilha ());
    }
  while (VALOR != -1);
}

void
empilha (int i)
{
  if (TOPO + 1 == TAM)
    {
      puts ("Pilha cheia");
      return;
    }
  TOPO++;
  *TOPO = i;
}

int
desempilha (void)
{
  if (TOPO == PILHA - 1)
```

```
    {
      puts ("Pilha vazia");
      return 0;
    }
  TOPO--;
  return *(TOPO + 1);
}
```

Ponteiros e matrizes

Na declaração :

```
char str [80], *p1;
```

Tanto a variável str com p1 apontam para um caracter (str aponta para o primeiro caracter do string e p1 para um caracter qualquer).

A atribuição:

```
p1 = str
```

Faz com que p1 e str apontem para a primeira posição do string. Então para acessar o quinto elemento do string poderemos proceder de duas formas:

```
str [4] ou *(p + 4)
```

Programadores em C preferem indexar os elementos da matriz usando ponteiros.

```
/* Exemplo 36 - Outro aplicacao com ponteiros */
#include <stdio.h>
#include <ctype.h>
void
main (void)
{
  char *nome = "Universidade";
  while (*nome)
    putchar (toupper (*nome++));
  printf ("\n");
}
```

Sintaxe de definição de uma função:

```
     tipo NomedaFunção(ListaParâmetros);
```

onde:

- ⊗ tipo: Pode ser um tipo válido de C. Se não for definido nenhum tipo para função, C assume que a função devolve um valor inteiro (int).
- ⊗ NomedaFunção: Nome válido de identificador em C pelo qual a função será chamada.
- ⊗ ListaParâmetros: Lista de nomes de variáveis separadas por vírgulas e seus tipos associados. Estas variáveis receberão os valores dos argumentos na chamada da função.

```
/* Exemplo 37 - Funcao esta contido */
#include <stdio.h>
#include <string.h>
contido (char c, char *s);
void
main (void)
{
  char nome[30], car;
  printf ("\nDigite um nome....: ");
  fgets (nome, 29, stdin);
  nome[strlen (nome) - 1] = '\0';
  printf ("\nDigite um caracter: ");
  scanf ("%c", &car);
  if (contido (car, nome))
    printf ("O caracter [%c] esta' no nome [%s]\n", car, nome);
  else
    printf ("O caracter [%c] nao esta' no nome [%s]\n", car, nome);
}
contido (char c, char *s)
{
  while (*s)
    if (*s == c)
      return 1;
    else
      s++;
  return 0;
}
```

Chamada por valor e chamada por referência

Existem duas maneiras de se passar argumentos para uma função:

- ⊗ Passagem por valor - o argumento não é alterado pela função.
- ⊗ Passagem por referência - o argumento é alterado.

Chamada por valor

Uso normal dos parâmetros formais da função sem alterar os valores dos argumentos (Função funcionando como função):

```
/* Exemplo 38 - Parametros por valor */
#include <stdio.h>
int quadrado (int x);
main ()
{
  int n = 8;
  printf ("\nO quadrado de %d e' %d\n", n, quadrado (n));
```

```
}
int
quadrado (int x)
{
  return x * x;
}
```
Obs:

- ⊗ Para passagem por referência devemos passar o endereço do argumento que queremos alterar dentro da função.
- ⊗ Existem funções de biblioteca C que usam deste mecanismo. Exemplo:

```
          int t;
          scanf ("%d", &t);
```

Chamada por referência

```
/* Exemplo 39 - Parametros por referencia */

#include <stdio.h>

void quadrado (int x, int *q);

main ()
{
  int n = 8, r;
  quadrado (n, &r);
  printf ("\nO quadrado de %d e' %d\n", n, r);
}

void
quadrado (int x, int *q)
{
  *q = x * x;
}
```

Funções de Biblioteca

Várias funções de biblioteca C podem ser incluídas em programas. Isto é feito com a inclusão de um arquivo de cabeçalho no início do programa. Este arquivo de cabeçalho (com extensão .h) contém uma série de definições dos protótipos de funções e de constantes.

Por exemplo, com a inclusão do arquivo de cabeçalho STDIO.H:

```
#include "stdio.h"
```

as funções de biblioteca printf, scanf entre outras estarão disponíveis ao programa.

Alguns arquivos de cabeçalho e suas funções principais:

Arquivo de cabeçalho	Principais funções
conio.h	clrscr(); gotoxy(); clreol(); kbhit(); getch();
stdio.h	gets(); puts (); printf(); scanf(); getchar(); putchar();
stdlib.h	abort(); atof(); atol(); atoi(); exit();
string.h	strcmp(); strcat(); strcpy(); strchr(); strlen(); strstr(); strlen();
graphics.h	initgraph(); rectangle(); closegraph(); getimage(); putimage(); putpixel(); floodfill(); line();
dos.h	delay(); gettime(); setdate(); getdate(); peek(); poke(); sound();
ctype.h	toupper(); tolower();

10. Estruturas em C

Estruturas são conjuntos heterogêneos de dados que são agrupados e referenciados através de um mesmo nome. Equivalente ao REGISTRO de outras linguagens de programação.

A forma geral de definição de estrutura é a seguinte:

```
struct nome {
          tipo nome_da_variável;
          tipo nome_da_variável;
          tipo nome_da_variável;
          ...
} variáveis_estruturas;
```

Exemplo:

```
     struct registro {
          char nome [30];
          char endereco [30];
          char cidade [20];
          char telefone [15];
          char estado [3];
          char cep [10];
     } func, auxiliar1, auxiliar2;
```

Obs:

⊗ Com esta declaração está sendo definida uma estrutura de nome registro e tres variáveis que são do tipo desta estrutura construída: func, auxiliar1, auxiliar2.

Referência a elementos de estrutura

Usa-se do operador ponto. Por exemplo, para se atribuir o CEP 37460 para o campo CEP da variável estrutura func usa-se o seguinte código:

```
               strcpy (func.cep, "37460");
```

OBS:

⊗ a atribuição de variáveis do tipo string em C só pode ser feita através da função de biblioteca **strcpy**.

⊗ nome da variável estrutura seguindo por um ponto e pelo nome do elemento referencia o elemento individual da estrutura: **func.cep** referencia o campo cep da estrutura func.

⊗ operador ponto pode aparecer em qualquer lugar. Exemplo:

```
      gets (func.nome);
```

Atribuição de estruturas

Uma estrutura inteira pode ser atribuída a outra, nas versões de C que são compatíveis ao padrão ANSI.

```
/* Exemplo 40 - Estruturas em C */

#include <stdio.h>

void
main (void)
{
  struct
  {
    int a;
    int b;
  }
```

```
  x, y;
  x.a = 7;
  y = x;
  printf ("\n%d", y.a);
}
```

Matrizes de estruturas

Pode-se declarar matrizes de estruturas em C. Por exemplo, para a estrutura **registro** declarada anteriormente, a seguinte matriz é válida:

```
     struct registro_matriz [100];
```

Obs:

- ⊗ Com esta declaração está sendo definida uma matriz de índices de 0 a 99 de elementos do tipo estrutura registro.
- ⊗ Para se acessar um estrutura específica, deve-se indexar o nome da estrutura. Por exemplo para se imprimir o código de CEP da estrutura 3, escreva:

```
     printf ("%s", matriz [2].cep);
```

Passando estruturas inteiras para funções

Quando uma estrutura é passada para uma função, a estrutura inteira é passada usando o método de chamada por valor.

```
/* Exemplo 41 - Estruturas e funcoes */

#include <stdio.h>

struct reg
{
  int a, b;
  char c;
};

void funcao (struct reg param);

main ()
{
  struct reg arg;
  arg.a = 1000;
  funcao (arg);
  printf ("\n%d", arg.a);
}


void
funcao (struct reg param)
{
  printf ("\n%d", param.a);
  param.a = 200;
}
```

Obs:

- ⊗ No exemplo anterior o argumento **arg** não terá seu valor alterado apesar do parâmetro **param** ter tido seu valor mudado dentro da **funcao**

Ponteiros para estruturas

Como outros ponteiros, você declara ponteiros para estruturas colocando * na frente do nome da estrutura.

Exemplo:

```
     struct conta {
                    float saldo;
                    char nome [30];
                 } cliente;
     struct conta *p;
     p = &cliente;
```

A instrução acima põe o endereço da estrutura cliente no ponteiro p.

Para acessar os campos de uma estrutura usando um ponteiro para estrutura deve-se usar o operador ->. Exemplo:

```
     p->nome
```

A palavra chave typedef

C admite que se defina explicitamente novos nomes aos tipos de dados utilizando a palavra chave **typedef**. Por exemplo:

```
/* Exemplo 42 - Estruturas e funcoes com TYPEDEF */

#include <stdio.h>

typedef struct
{
  int a, b;
  char c;
}
reg;

void funcao (reg param);

main ()
{
  reg arg;
  arg.a = 1000;
  funcao (arg);
  printf ("\n%d", arg.a);
}

void
funcao (reg param)
{
  printf ("\n%d", param.a);
  param.a = 200;
}
```

<u>Exercícios:</u>

1. Desenvolver usando vetor e aritmética de ponteiros as seguintes funções:
    - copia (char *s1, char *s2) - função que copia s2 em s1.
    - void conc (char *s1, char *s2) - função que concatena s2 ao final de s1.
    - int tam (char *s) - função que retorna o tamanho do string s.
    - int compara (char *s1, char *s2) - função que retorna 0 se s1 <> de s2 e retorna 1 se s1 = a s2.

- int pos (char *s, char c) - função que devolve a posição + 1 (POS) da primeira ocorrência de C no string S. Se o caracter C não está em S o valor 0 será retornado.

2. Definir uma estrutura de árvore binária usando vetor e implementar algumas das primitivas para manipulação deste novo tipo estruturado (contrucao de árvore, acompanhamento da árvore em pré-ordem, apresentação espacial da árvore...).

3. Desenvolver um programa em C para ler uma lista de números e calcular qual o maior dos números. OBS: desenvolver uma função que encontra o maior elemento num vetor de números que é passado para a função e fazer a chamada desta função no programa principal.

4. Desenvolver uma função em C que faz a fusão de dois vetores ordenados, dando o resultado no terceiro vetor. OBS: O terceiro vetor deve estar ordenado ao final da função.

5. Usando aritmética de ponteiros, faça uma função em C que reverta uma cadeia de caracteres. Assim: Antes: "TESTE" Depois: "ETSET"

6. Escrever um programa que use a função do questão anterior.

7. Codifique a seguinte rotina em C:

```
procedimento ORDENA (VETOR : VET [0..10] DE REAL)
     declare i, j: INT
             x   : REAL
     inicio
          i <- 2
          repita
               Se i > 10 Entao Interrompa
               x <- VETOR [i]
               VETOR [0] <- x;
               j <- i - 1
               repita
                    Se x >= VETOR [j] Entao Interrompa
                    VETOR [j+1] <- VETOR [j]
                    j <- j - 1
               fim-repita
               VETOR [j+1] <- x
               i <- i + 1
          fim-repita
     fim
```

11. Entrada e Saída com Arquivo em C

Existem dois conjuntos de funções de E/S em C: E/S ANSI (com buffer ou formatada) e a E/S UNIX (sem buffer ou não formatada). Daremos uma ênfase maior ao primeiro conjunto pela portabilidade deste sistema de entrada e saída com arquivos.

Streams e Arquivo

O sistema de arquivo C é definido para trabalhar com uma série de dispositivos: terminais, acionadores de disco, etc... Estes dispositivos são vistos como arquivos lógico em C denominados STREAM. (abstração do dispositivo).

O dispositivo real é denominado ARQUIVO (impressora, disco, console). Um STREAM é associado a um ARQUIVO por uma operação de abertura do arquivo.

Funções mais comuns no sistema de E/S ANSI:

Função	Descrição
fopen ()	abre um arquivo
fclose ()	fecha um arquivo
putc () e fputc ()	escreve um caracter num arquivo
getc () e fgetc ()	lê um caracter num arquivo
fseek ()	posiciona numa posição do arquivo
fprintf ()	impressão formatada em arquivo
fscanf ()	leitura formatada em arquivo
feof ()	verdadeiro se o fim do arquivo foi atingido
fwrite ()	escreve tipos maiores que 1 byte num arquivo
fread ()	lê tipos maiores que 1 byte num arquivo

Ponteiro para arquivo

O ponteiro de arquivo une o sistema de E/S com um buffer. O ponteiro não aponta diretamente para o arquivo em disco e sim contém informações sobre o arquivo, incluindo nome, status (aberto, fechado) e posição atual sobre o arquivo.

Para se definir uma variável ponteiro de arquivo usa-se o seguinte comando:

```
FILE *arq;
```

OBS: com este comando passa a existir uma variável de nome arq que é ponteiro de arquivo.

Abrindo arquivo

A função que abre arquivo em C é a função fopen (). Ela devolve NULL (nulo) ou um ponteiro associado ao arquivo e deve ser passado para função o nome físico do arquivo e o modo como este arquivo deve ser aberto.

Exemplo:

```
arq = fopen ("teste", "w");
```

OBS: Está sendo aberto um arquivo de nome "teste" para escrita (w - write).

Ficaria melhor assim:

```
if ((arq = fopen ("teste", "w")) == NULL)
   {
     puts ("Arquivo nao pode ser aberto");
     exit (1);
   }
```

Valores legais para MODO:

Modo	Significado
r	abre um arquivo texto para leitura
w	cria um texto para escrita
a	anexa um arquivo texto
rb	abre um arquivo binário para leitura
wb	cria um arquivo binário para escrita
ab	anexa um arquivo binário
r+	abre texto para leitura/escrita
w+	cria texto para leitura/escrita
a+	anexa texto para leitura/escrita
r+b	abre um arquivo binário para leitura/escrita
w+b	cria um arquivo binário para leitura/escrita
a+b	anexa um arquivo binário para leitura/escrita

Fechando um arquivo

A função que fecha e esvazia o buffer de um arquivo é a função fclose ()

Exemplo:

```
          fclose (arq);
```

```
/* Exemplo 45 - Le teclado, grava arquivo */

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

main ()
{
  FILE * arq;
  char c, nome[20];
  printf ("Mini editor em C - digite ESC e ENTER para terminar.\n");
  printf ("Nome do arquivo? ");
  scanf ("%s", nome);
  if ((arq = fopen (nome, "w")) == NULL)
    {
      printf ("Erro abertura\n");
      printf ("Arquivo : %s\n", nome);
      exit (1);
    }
  do
    {
      c = getchar ();
      if (c == 13)
      {
        putc ('\n', arq);
        puts ("");
      }
      else if (c != 27)
      putc (c, arq);
    }
  while (c != 27);
  fclose (arq);
}
```

```
/* Exemplo 46 - Le de arquivo, mostra na tela */

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

main ()
{
  FILE * arq;
  char nome[20];
  signed char c;
  int contalinha = 1;
  printf ("Nome do arquivo? ");
  scanf ("%s", nome);
  if ((arq = fopen (nome, "rt")) == NULL)
    {
      printf ("Erro de abertura\n");
      printf ("Arquivo: %s", nome);
      exit (1);
    }
  c = getc (arq);
  printf ("%4d:", contalinha);
  while (c != EOF)
    {
      if (c == 10)
      {
        contalinha++;
        printf ("\n%4d:", contalinha);
      }
      else
      putchar (c);
      c = getc (arq);
    }
  fclose (arq);
}
```

Funções fread () e fwrite ()

Para ler e escrever estruturas maiores que 1 byte, usa-se as funções fread () e fwrite ().
Parâmetros das funções:

```
fread (buffer, tamanhoembytes, quantidade, ponteirodearquivo)
fwrite (buffer, tamanhoembytes, quantidade, ponteirodearquivo)
```

ONDE:

- ⊗ buffer- é um endereço na memória da estrutura para onde deve ser lido ou de onde serão escritos os valores (fread () e fwrite () respectivamente).
- ⊗ tamanhoembytes - é um valor numérico que define o tamanho em bytes da estrutura que deve ser lida/escrita.
- ⊗ quantidade - número de estrutura que serão lidas ou escritas em cada processo de fread ou fwrite.
- ⊗ ponteirodearquivo - é o ponteiro do arquivo onde será lida ou escrita uma estrutura.

Programa exemplo:

Programa que lê estruturas de dois campos : nome e telefone e vai gravando num arquivo denominado "ARQUIVO". Este processo é terminado com a digitação do campo nome vazio, quando então o programa apresenta todos os registros do arquivo que acaba de ser gerado.

```
/* Exemplo 47 - fread e fwrite */

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

typedef struct
{
  char nome[30];
  char telefone[20];
}
registro;

main ()
{
  registro reg;
  FILE * arq;
  char numstr[5];
  if ((arq = fopen ("arquivo", "w+b")) == NULL)
    {
      puts ("Erro na abertura do arquivo");
      exit (1);
    }
  puts ("Digite enter no NOME para terminar");
  puts ("");

  do
    {
      printf ("\nEntre um nome..... : ");
      fgets (reg.nome, 29, stdin);
      reg.nome[strlen (reg.nome) - 1] = '\0';
      printf ("Entre com telefone : ");
      if (reg.nome[0])
      {
        fgets (reg.telefone, 19, stdin);
        reg.telefone[strlen (reg.telefone) - 1] = '\0';
        fwrite (&reg, sizeof (registro), 1, arq);
      }
    }
  while (reg.nome[0]);
  rewind (arq);
  printf ("\n\nListagem do arquivo\n");
  puts ("-------------------\n");
  printf ("%-30s  %20s\n", "NOME", "TELEFONE");
  while (!feof (arq))
    {
      fread (&reg, sizeof (registro), 1, arq);
      if (!feof (arq))
      printf ("%-30s  %20s\n", reg.nome, reg.telefone);
    }
}
```

Manutenção de arquivo sequencial em C

```
/* Exemplo 48 - Manutencao de arquivo em C */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

  typedef struct
{
  char nome[30];
  char telefone[15];
  int status;
} registro;
FILE * arq;

int insere (registro r);
void lista (void);
int retira (registro r);
void entra (registro * r);
int consulta (registro * r);
char menu (void);
void msg (char *s);
void escreve (char *s);

main ()
{
  registro reg;
  char opcao;
  if ((arq = fopen ("arquivo", "rb+")) == NULL)
    if ((arq = fopen ("arquivo", "wb+")) == NULL)
      {
      puts ("Erro na abertura do arquivo");
      exit (1);
      }
  do
    {
      opcao = menu ();
      printf ("\n\n");
      switch (opcao)
      {
      case '1':
        puts ("Insercao");
        entra (&reg);
        if (insere (reg))
          puts ("Insercao OK");
        else
          puts ("ERRO na insercao");
        break;
      case '2':
        puts ("Remocao");
        printf ("\n Entre com um nome.... :");
        fgets (reg.nome, 29, stdin);
        reg.nome[strlen (reg.nome) - 1] = '\0';
        if (retira (reg))
          puts ("Remocao OK\n");
        else
          puts ("Erro na remocao\n");
        break;
      case '3':
```

```
        puts ("Lista");
        lista ();
        break;
      case '4':
        puts ("Consulta");
        printf ("\n Entre com um nome.... :");
        fgets (reg.nome, 29, stdin);
        reg.nome[strlen (reg.nome) - 1] = '\0';
        if (consulta (&reg))
          {
            printf ("\nNOME:%s, TEL:%s\n\n", reg.nome, reg.telefone);
            msg ("");
          }
        else
          msg ("NOME nao encontrado\n");
      }
    }
  while (opcao != '5');
  puts ("fim do programa...");
  fclose (arq);
}

char
menu (void)
{
  char opcao;
  puts ("\nMENU :      ");
  puts ("1. Insere  ");
  puts ("2. Retira  ");
  puts ("3. Lista   ");
  puts ("4. Consulta");
  puts ("5. Fim");
  printf ("\nEscolha ==>");
  opcao = getchar ();
  getchar ();
  return opcao;
}

void
msg (char *s)
{
  puts (s);
  puts ("Digite qualquer tecla...");
  getchar ();
}

void
entra (registro * r)
{
  printf ("\n Entre com um nome.... :");
  fgets (r->nome, 29, stdin);
  r->nome[strlen (r->nome) - 1] = '\0';
  printf (" Entre com um telefone :");
  fgets (r->telefone, 14, stdin);
  r->telefone[strlen (r->telefone) - 1] = '\0';
  r->status = 1;
}

int
```

```
insere (registro r)
{
  if (!fseek (arq, 0, SEEK_END))
    {
     fwrite (&r, sizeof (registro), 1, arq);
     return 1;
    }
  return 0;
}

int
retira (registro r)
{
  registro reg;
  fseek (arq, 0, SEEK_SET);
  while (!feof (arq))
    {
      fread (&reg, sizeof (registro), 1, arq);
      if (!strcmp (reg.nome, r.nome) && (reg.status))
      {
        reg.status = 0;
        fseek (arq, -1L * sizeof (registro), SEEK_CUR);
        if (fwrite (&reg, sizeof (registro), 1, arq))
          return 1;
      }
    }
  return 0;
}

void
lista (void)
{
  int i;
  registro r;
  rewind (arq);
  printf ("\n%-30s %-15s\n", "Nome", "Telefone");
  for (i = 0; i < 46; i++)
    printf ("-");
  printf ("\n");
  while (!feof (arq))
    {
      fread (&r, sizeof (registro), 1, arq);
      if ((r.status) && (!feof (arq)))
        printf ("\n%-30s %-15s", r.nome, r.telefone);
    }
  printf ("\n\n");
}

int
consulta (registro * r)
{
  registro reg;
  fseek (arq, 0, SEEK_SET);
  while (!feof (arq))
    {
      fread (&reg, sizeof (registro), 1, arq);
      if (!strcmp (reg.nome, r->nome) && (reg.status))
      {
        *r = reg;
```

```
        return 1;
      }
    }
  return 0;
}
```

Exercício:
Modifique este programa para manutencao de um arquivo sequencial com os seguintes campos:
nome (string de 20 posições), endereco (string de 20 posições), cidade (string de 20 posições), estado (string de 2 posições) e cep (inteiro longo)

12. Alocação Dinâmica de Memória

Existe em C, uma forma poderosa de se gerar espaço de memória dinamicamente, durante a execução de um programa. Isto é necessário quando nos programas temos quantidades de armazenamento variáveis. Por exemplo, uma pilha dentro de um programa qualquer. A pilha deve crescer dinamicamente, conforme forem sendo empilhados elementos. Se usarmos o vetor como mecanismo para manutenção desta pilha, estaremos limitando o número de elementos que poderão ser mantidos na pilha. Se usarmos alocação dinâmica para os elementos da pilha, a única limitação que teremos será devido ao espaço de memória disponível quando da execução do programa.

São basicamente duas as funções de alocação dinâmica de memória em C:

malloc- Devolve o endereço do primeiro byte da memória alocado para uma variável tipo ponteiro qualquer. Equivalente ao comando NEW da linguagem PASCAL.

Exemplo:

```
               char *p;
               p = malloc (1000);
```

OBS: foram alocados com esta instrução 1000 bytes de memória sendo que p aponta para o primeiro destes 1000 bytes. Um string foi alocado dinamicamente. Sempre que se alocar memória deve-se testar o valor devolvido por malloc (), antes de usar o ponteiro, para garantir que não é nulo. Assim:

```
          if (p=malloc (1000)){
               puts ("sem memoria\n");
               exit (1);
          }
```

free- A função free é a função simétrica de malloc, visto que ela devolve para o sistema uma porção de memória que foi alocada dinamicamente. Equivalente ao comando DISPOSE da linguagem PASCAL.

```
/* Exemplo 49 - Alocacao dinamica de memoria */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

  main ()
{
  char *s, tamstr[6];
  int t;
  printf ("\n\nQual o tamanho do string ? ");
  fgets (tamstr, 5, stdin);
  t = atoi (tamstr);
  s = malloc (t);
  if (!s)
    {
      puts ("Sem memoria!");
      exit (1);
    }
  printf ("Entre um string qualquer : ");
  fgets (s, t, stdin);
  printf ("Ao contrario fica....... : ");
  for (t = strlen (s) - 1; t >= 0; t--)
    putchar (s[t]);
  free (s);
  printf ("\n");
}
```

Programa exemplo completo que implementa a estrutura de dados árvore binária por alocação dinâmica em linguagem C:

```
/* Exemplo 50 - Arvore Binaria em linguagem C */

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

struct arv
{
  char info[30];
  struct arv *dir;
  struct arv *esq;
};


struct arv *constroi (void);
void preordem (struct arv *no);
void posordem (struct arv *no);
void ordeminter (struct arv *no);
void destroi (struct arv *no);
void mostra (struct arv *no, int nivel);
int menu (void);
void escreve (char *s);

main ()
{
  struct arv *ARVORE = NULL;
  int escolha;
  do
    {
      escolha = menu ();
      switch (escolha)
      {
      case 0:
        puts ("Constroi arvore\n\r");
        destroi (ARVORE);
        ARVORE = constroi ();
        break;
      case 1:
        puts ("Pre'-ordem\n\r");
        preordem (ARVORE);
        break;
      case 2:
        puts ("Pos ordem\n\r");
        posordem (ARVORE);
        break;
      case 3:
        puts ("Ordem intermediaria\n\r");
        ordeminter (ARVORE);
        break;
      case 4:
        puts ("Mostra arvore\n\r");
        mostra (ARVORE, 0);
        break;
       }
       puts ("\n\nDigite qualquer tecla...");
```

```
        getchar ();
      }
    while (escolha != 5);
    destroi (ARVORE);
}

int
menu (void)
{
    int opcao;
    puts ("Opcoes:");
    puts ("-------");
    puts ("0. Constroi arvore");
    puts ("1. Mostra arvore em Pre'-ordem");
    puts ("2. Mostra arvore em Pos-ordem");
    puts ("3. Mostra arvore em Ordem-intermediaria");
    puts ("4. Desenha a arvore");
    puts ("5. Fim de operacoes\n\n");


    do
      {
        printf ("Escolha [0,1,2,3,4 ou 5]: ");
        opcao = getchar () - 48;
        getchar ();
        puts ("\n\n");
      }
    while ((opcao < 0) && (opcao > 5));
    return opcao;
}

struct arv *
constroi (void)
{
    struct arv *no;
    char auxstr[30];
    fgets (auxstr, 29, stdin);
    auxstr[strlen (auxstr) - 1] = '\0';
    if (strcmp (auxstr, ".") == 0)
      return NULL;
    else
      {
        no = malloc (sizeof (struct arv));
        strcpy (no->info, auxstr);
        no->esq = constroi ();
        no->dir = constroi ();
        return no;
      }
}

void
preordem (struct arv *no)
{
    if (no)
      {
        puts (no->info);
        preordem (no->esq);
        preordem (no->dir);
      }
```

```
}

void
posordem (struct arv *no)
{
  if (no)
    {
      posordem (no->esq);
      posordem (no->dir);
      puts (no->info);
    }
}

void
ordeminter (struct arv *no)
{
  if (no)
    {
      ordeminter (no->esq);
      puts (no->info);
      ordeminter (no->dir);
    }
}

void
destroi (struct arv *no)
{
  if (no)
    {
      destroi (no->esq);
      destroi (no->dir);
      free (no);
      no = NULL;
    }
}

void
mostra (struct arv *no, int nivel)
{
  int i;
  if (no)
    {
      mostra (no->dir, nivel + 1);
      for (i = 0; i < nivel; i++)
            printf ("   ");
      puts (no->info);
      mostra (no->esq, nivel + 1);
    }
}
```

13. Bibliografia

Livros:

SCHILDT, HERBERT, C completo e total, São Paulo, MacGrawHill, 1990.
SCHILDT, HERBERT, Linguagem C: Guia do Usuário, São Paulo, MacGrawHill, 1986.
SWAN, TOM – Aprendendo C++, São Paulo, MacGrawHill, 1993.
BERRY, JOHN THOMAS, Programando em C++, São Paulo, MacGrawHill, 1991.
TANENBAUM, AARON M., Estruturas de dados usando C, São Paulo, MAKRON BOOKS, 1995.

Web Sites:

http://ead1.eee.ufmg.br/cursos/C
http://www.portalc.na-web.net

Anexo 1 - O Pré-processador C

Vamos falar de algumas diretivas de C definidas para serem executadas e analisadas antes da execução do programa, durante o processo de compilação. Trata-se das diretivas de préprocessamento. Algumas destas diretivas podem parecer dispensáveis, mas existem fundamentalmente para manipulação de alguns casos especiais.

Como foi definido pelo padrão ANSI, o pré-processador C contém as seguintes diretivas:

```
     #if         #ifdef       #ifndef        #else
     #elif       #include     #define        #undef
     #error      #pragma
```

Todas começam com o caracter #. Além disso cada diretiva deve estar numa linha distinta.

#define

Esta diretiva é usada para efetuar macro-substituição, ou seja, substituição de um nome por um string ou um determinado valor por todo o arquivo fonte do programa.

FORMA GERAL:

```
               #define nome_macro string/valor
```

```
/* Exemplo 53 - pre processamento */

#include <stdio.h>

#define UM 1
#define DOIS UM+UM
#define TRES UM+DOIS

void main (void)
{
  printf ("%d, %d, %d\n", UM, DOIS, TRES);
  getchar ();
}
```

```
/* Exemplo 52 - Macro substituicao */

#include <STRING.H>
#include <STDIO.H>

#define INICIO {
#define FIM }
#define ESCREVA printf
#define LEIA gets
#define PROGRAMA void main (void)
#define CARACTER char
#define PARA for
#define INTEIRO int
#define TAMSTR strlen

PROGRAMAINICIO
     INTEIRO i, j;
     CARACTER s[30], c;
     ESCREVA ("Escreva um string qualquer: ");
     LEIA (s);
```

```
     PARA (i = 0, j = TAMSTR (s) - 1; i < j; i++, j--)
     INICIO
      c = s[i];
      s[i] = s[j];
      s[j] = c;
     FIM
     ESCREVA ("O string invertido fica.. : %s\n", s);
     ESCREVA ("digite <ENTER>...");
     LEIA (s);
FIM
```

Uma boa convenção é definir os nomes de macros todas com letra maiúscula para fácil identificação das macros. O uso das macros é interessante também nos casos em que se deseja definir valores constantes de uso global. Por exemplo:

```
#define MAX 20;
int v[MAX];
void main (void)
{
 int i;
 for (i = 0; i < MAX; i++)
   v[i] = i+1;
}
```

#error

Uma outra diretiva do pré-processador é a diretiva #error. Esta diretiva tem a função de parar a compilação com um erro e é usada principalmente para depuração de programas.

```
/* Exemplo 54 - Diretiva #error */

#include <stdio.h>

#define UM 1
void main (void)
{
  printf ("%d.\n", UM);
#error "Parei aqui"
  getche ();
}
```

/* Neste programa vai aparecer a seguinte mensagem :
  Error: Error directive: "Parei aqui" in function main */

#include

Instrui o compilador a ler outro arquivo fonte (arquivo de cabeçalho .h ou outro arquivo qualquer). Exemplo:

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

Todas as funções do arquivo de cabeçalho STDIO.H que estão sendo usadas num programa serão incluidas.

#if #else #elif #endif

Estas diretivas são definidas para compilação condicional, ou seja, o compilador vai compilar um bloco ou outro de um arquivo fonte dependendo de condições. Este esquema é muito usado em software houses que tem várias versões de um mesmo programa.

```
/* Exemplo 55 - Compilacao condicional */

#include <STDIO.H>
#include <CONIO.H>

#define MAX 10

void main (void)
{
#if MAX > 99
   printf ("Compilado para matriz maior que 99\n");
#else
   printf ("Compilado para matriz pequena\n");
#endif
   getch ();
}
```

```
/* Exemplo 56 - Compilacao condicional */

#include <STDIO.H>
#include <CONIO.H>

#define EUA 1
#define INGLATERRA 2
#define BRASIL 3

#define PAIS_ATIVO EUA

#if PAIS_ATIVO == BRASIL
  char moeda [10] = "dolar";
#elif PAIS_ATIVO == INGLATERRA
  char moeda [10] = "libra";
#else
  char moeda [10] = "cruzeiro";
#endif

void main (void)
{
 printf ("MOEDA : %s\n", moeda);
 getche ();
}
```

#ifdef e #ifndef

Estas diretivas condicionam a compilação de um bloco ou outro de um programa dependendo de uma macro estar ou não definida para este programa.
Exemplo:

```
/* Exemplo 57 - diretiva #ifdef */

#include <STDIO.H>
#include <CONIO.H>

#define TED 10
#define RALPH 20

void main (void)
{
 #ifdef TED
    printf ("TED esta' definido\n");
```

```
 #else
    printf ("TED nao esta' definido\n");
 #endif

 #ifndef RALPH
    printf ("Ralph nao esta' definido\n");
 #else
    printf ("Ralph esta' definido\n");
 #endif
    getche ();
}
```

#undef

Diretiva usada para remover uma definição anterior de uma macro.

```
/* Exemplo 58 - diretiva #undef */

#include <STDIO.H>
#include <CONIO.H>

#define TED 10
void main (void)
{
 #undef TED
 #ifdef TED
    printf ("TED esta' definido\n");
 #else
    printf ("TED nao esta' definido\n");
 #endif
    getche ();
}
```

#line

Diretiva usada para modificar o valor de algumas macros pré definidas em C.

```
/* Exemplo 59 -
   Este programa testa a diretiva #line
   O seguinte resultado pode ser apresentado:

17          -Posicao do fonte da primeira macro __LINE__
EXEMP059.C  -Nome do arquivo fonteMay 06 1992 -Data do arquivo fonte
13:27:13    -Hora do arquivo fonte
21          -Posicao do fonte da segunda macro __LINE__
*/

#include <STDIO.H>
#include <CONIO.H>

void main (void)
{
 printf ("%d\n", __LINE__);
 printf ("%s\n", __FILE__);
 printf ("%s\n", __DATE__);
 printf ("%s\n", __TIME__);
 printf ("%d\n", __LINE__);
 getche ();
}
```

Os operadores # e

O operador # transforma o argumento que ele precede em um string. E o operador ## concatena duas palavras.

```
/* Exemplo 51 - Uso dos operadores # e ##  */
/*  # transforma seu argumento em string   */
/* ## concatena duas palavras              */

#include <STDIO.H>

#define mkstr(s) # s
#define concat(a,b) a ## b

void main (void)
{
 int xy;
 xy = 10;
 printf (mkstr(Eu gosto de C\n));
 printf ("%d\n", concat(x,y));
 getche ();
}
```

#pragma

Esta diretiva é definida de forma diferenciada para cada compilador C. Na versão TURBO C esta diretiva diz ao compilador que o programa tem porções de código escritas em assembler.