UNIOESTE - Universidade Estadual do Oeste do Paraná
SEMINC 2003

Apostila da Linguagem C

Guilherme Galante
Raphael Laércio Zago

Cascavel
2003

Sumário

INTRODUÇÃO

Vamos, neste curso, aprender os conceitos básicos da linguagem de programação C a qual tem se tornado cada dia mais popular, devido à sua versatilidade e ao seu poder. Uma das grandes vantagens do C é que ele possui tanto características de "alto nível" quanto de "baixo nível".

Apesar de ser bom, não é pré-requisito do curso um conhecimento anterior de linguagens de programação. É importante uma familiaridade com computadores. O que é importante é que você tenha vontade de aprender, dedicação ao curso e, caso esteja em uma das turmas do curso, acompanhe atentamente as discussões que ocorrem na lista de discussões do curso.

O C nasceu na década de 70. Seu inventor, Dennis Ritchie, implementou-o pela primeira vez usando um DEC PDP-11 rodando o sistema operacional UNIX. O C é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.

O C é uma linguagem de programação genérica que é utilizada para a criação de programas diversos como processadores de texto, planilhas eletrônicas, sistemas operacionais, programas de comunicação, programas para a automação industrial, gerenciadores de bancos de dados, programas de projeto assistido por computador, programas para a solução de problemas da Engenharia, Física, Química e outras Ciências, etc ...

Estudaremos a estrutura do ANSI C, o C padronizado pela ANSI. Veremos ainda algumas funções comuns em compiladores para alguns sistemas operacionais. Quando não houver equivalentes para as funções em outros sistemas, apresentaremos formas alternativas de uso dos comandos.

1. PRIMEIROS PASSOS

1.1 O C é "Case Sensitive"

Vamos começar o nosso curso ressaltando um ponto de suma importância: o C é "Case Sensitive", isto é, *maiúsculas e minúsculas fazem diferença*. Se declararmos uma variável com o nome soma ela será diferente de **Soma**, **SOMA**, **SoMa** ou **sOmA**. Da mesma maneira, os comandos do C **if** e **for**, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

1.2 Dois Primeiros Programas

Vejamos um primeiro programa em C:

```
#include <stdio.h>
/* Um Primeiro Programa */
int main ()
{
      printf ("Ola! Eu estou vivo!\n");
      return(0);
}
```

Compilando e executando este programa você verá que ele coloca a mensagem *Ola! Eu estou vivo!* na tela.

Vamos analisar o programa por partes.

A linha **#include <stdio.h>** diz ao compilador que ele deve incluir o arquivo-cabeçalho **stdio.h**. Neste arquivo existem declarações de funções úteis para entrada e saída de dados (std = standard, padrão em inglês; io = Input/Output, entrada e saída ==> stdio = Entrada e saída padronizadas). Toda vez que você quiser usar uma destas funções deve-se incluir este comando. O C possui diversos arquivos-cabeçalhos.

Quando fazemos um programa, uma boa idéia é usar comentários que ajudem a elucidar o funcionamento do mesmo. No caso acima temos um comentário: **/* Um Primeiro Programa */**. O compilador C desconsidera qualquer coisa que esteja começando com **/*** e terminando com ***/**. Um comentário pode, inclusive, ter mais de uma linha.

A linha **int main()** indica que estamos definindo uma função de nome **main**. Todos os programas em C têm que ter uma função **main**, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves **{ }**. O código que estiver dentro das chaves será executado sequencialmente quando a função for chamada. A palavra int indica que esta função retorna um inteiro. O que significa este retorno será visto posteriormente, quando estudarmos um pouco mais detalhadamente as funções do C. A última linha do programa, **return(0)**; , indica o número inteiro que está sendo retornado pela função, no caso o número 0.

A única coisa que o programa *realmente* faz é chamar a função **printf()**, passando a string (uma string é uma seqüência de caracteres, como veremos brevemente) **"Ola! Eu estou vivo!\n"** como argumento. É por causa do uso da função **printf()** pelo programa que devemos incluir o arquivo- cabeçalho **stdio.h** . A função **printf()** neste caso irá apenas colocar a string na tela do computador. O **\n** é uma constante chamada de *constante barra invertida*. No caso, o **n** é a constante barra invertida de "new line" e ele é interpretado como um comando de mudança de linha, isto é, após imprimir *Ola! Eu estou vivo!* o cursor passará para a próxima linha. É importante observar também que os *comandos* do C terminam com **;** .

Podemos agora tentar um programa mais complicado:

```
#include <stdio.h>
int main ()
{
      int Dias; /* Declaracao de Variaveis */
      float Anos;
      printf ("Entre com o número de dias: ");/* Entrada de Dados */
      scanf ("%d",&Dias);
      Anos=Dias/365.25; /* Conversao Dias->Anos */
      printf ("\n\n%d dias equivalem a %f anos.\n",Dias,Anos);
      return(0);
}
```

Vamos entender como o programa acima funciona. São declaradas duas variáveis chamadas **Dias** e **Anos**. A primeira é um **int** (inteiro) e a segunda um **float** (ponto flutuante). As variáveis declaradas como ponto flutuante existem para armazenar números que possuem casas decimais, como 5,1497.

É feita então uma chamada à função **printf()**, que coloca uma mensagem na tela.

Queremos agora ler um dado que será fornecido pelo usuário e colocá-lo na variável inteira **Dias**. Para tanto usamos a função **scanf()**. A string **"%d"** diz à função que iremos ler um inteiro. O segundo parâmetro passado à função diz que o dado lido deverá ser armazenado na variável **Dias**. É importante ressaltar a necessidade de se colocar um **&** antes do nome da variável a ser lida quando se usa a função **scanf()**. O motivo disto só ficará claro mais tarde. Observe que, no C, quando temos mais de um parâmetro para uma função, eles serão separados por vírgula.

Temos então uma expressão matemática simples que atribui a **Anos** o valor de **Dias** dividido por 365.25 (365.25 é uma constante ponto flutuante 365,25). Como **Anos** é uma variável **float** o compilador fará uma conversão automática entre os tipos das variáveis (veremos isto com detalhes mais tarde).

A segunda chamada à função **printf()** tem três argumentos. A string **"\n\n%d dias equivalem a %f anos.\n"** diz à função para pular duas linhas, colocar um inteiro na tela, colocar a mensagem **" dias equivalem a "**, colocar um valor **float** na tela, colocar a mensagem **" anos."** e pular outra linha. Os outros parâmetros são as variáveis, **Dias** e **Anos**, das quais devem ser lidos os valores do inteiro e do **float**, respectivamente.

1.3 Palavras Reservadas do C

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser nos seus propósitos originais, isto é, não podemos declarar funções ou variáveis com os mesmos nomes. Como o C é "case sensitive" podemos declarar uma variável **For**, apesar de haver uma palavra reservada **for**, mas isto não é uma coisa recomendável de se fazer pois pode gerar confusão.

Apresentamos a seguir as palavras reservadas do ANSI C. Veremos o significado destas palavras chave à medida em que o curso for progredindo:

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

2. INTRODUÇÃO ÀS FUNÇÕES

Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O uso de funções permite que o programa fique mais legível, mais bem estruturado. Um programa em C consiste, no fundo, de várias funções colocadas juntas.

Abaixo o tipo mais simples de função:

```
#include <stdio.h>
int mensagem () /* Funcao simples: so imprime Ola! */
{
     printf ("Ola! ");
     return(0);
}

int main ()
{
     mensagem();
     printf ("Eu estou vivo!\n");
     return(0);
}
```

Este programa terá o mesmo resultado que o primeiro exemplo da seção anterior. O que ele faz é definir uma função **mensagem()** que coloca uma string na tela e retorna 0. Esta função é chamada a partir de **main()** , que, como já vimos, também é uma função. A diferença fundamental entre main e as demais funções do problema é que main é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

2.1 Argumentos

Argumentos são as entradas que a função recebe. É através dos argumentos que passamos *parâmetros* para a função. Já vimos funções com argumentos. As funções **printf()** e **scanf()** são funções que recebem argumentos. Vamos ver um outro exemplo simples de função com argumentos:

```
#include <stdio.h>

int square (int x)    /* Calcula o quadrado de x */
{
     printf ("O quadrado e %d",(x*x));
     return(0);
}

int main ()
{
     int num;
     printf ("Entre com um numero: ");
     scanf ("%d",&num);
     printf ("\n\n");
     square(num);
     return(0);
}
```

Na definição de **square()** dizemos que a função receberá um argumento inteiro **x**. Quando fazemos a chamada à função, o inteiro **num** é passado como argumento. Há alguns pontos a observar. Em primeiro lugar temos de satisfazer aos requesitos da função quanto ao tipo e à quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo, que o C faz automaticamente, é importante ficar atento. Em segundo lugar, não é importante o nome da variável que se passa como argumento, ou seja, a variável **num**, ao ser passada como argumento para **square()** é copiada para a variável **x**. Dentro de **square()** trabalha-se apenas com **x**. Se mudarmos o valor de **x** dentro de **square()** o valor de **num** na função **main()** permanece inalterado.

Vamos dar um exemplo de função de mais de uma variável. Repare que, neste caso, os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um. Note, também, que os argumentos passados para a função não necessitam ser todos variáveis porque mesmo sendo constantes serão copiados para a variável de entrada da função.

```
#include <stdio.h>

int mult (float a, float b,float c)   /* Multiplica 3 numeros */
{
printf ("%f",a*b*c);
return(0);
}

int main ()
{
      float x,y;
      x=23.5;
      y=12.9;
      mult (x,y,3.87);
      return(0);
}
```

2.2 Retornando valores

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui estavam retornando o número 0. Podemos especificar um tipo de retorno indicando-o antes do nome da função. Mas para dizer ao C *o que* vamos retornar precisamos da palavra reservada **return**. Sabendo disto fica fácil fazer uma função para multiplicar dois inteiros e que retorna o resultado da multiplicação. Veja:

```
#include <stdio.h>
int prod (int x,int y)
{
      return (x*y);
}
int main ()
{
      int saida;
      saida=prod (12,7);
      printf ("A saida e: %d\n",saida);
      return(0);
}
```

Veja que, como prod retorna o valor de 12 multiplicado por 7, este valor pode ser usado em uma expressão qualquer. No programa fizemos a atribuição deste resultado à variável saida, que posteriormente foi impressa usando o printf. Uma observação adicional: se não especificarmos o tipo de retorno de uma função, o compilador C automaticamente suporá que este tipo é inteiro. Porém, não é uma boa prática não se especificar o valor de retorno e, neste curso, este valor será sempre especificado.

Com relação à função main, o retorno sempre será inteiro. Normalmente faremos a função main retornar um zero quando ela é executada sem qualquer tipo de erro.

Mais um exemplo de função, que agora recebe dois floats e também retorna um float:

```
#include <stdio.h>
float prod (float x,float y)
{
      return (x*y);
}

int main ()
{
      float saida;
      saida=prod (45.2,0.0067);
      printf ("A saida e: %f\n",saida);
      return(0);
}
```

2.3 Forma geral

Apresentamos aqui a forma geral de uma função:

tipo_de_retorno nome_da_função (lista_de_argumentos)
{
código_da_função
}

3. INTRODUÇÃO À ENTRADA E SAÍDA

3.1 Caracteres

Os caracteres são um tipo de dado: o **char**. O C trata os caracteres ('a', 'b', 'x', etc ...) como sendo variáveis de um *byte* (8 *bit*s). Um *bit* é a menor unidade de armazenamento de informações em um computador. Os inteiros (**int**s) têm um número maior de *bytes*. Dependendo da implementação do compilador, eles podem ter 2 *bytes* (16 *bits*) ou 4 *bytes* (32 *bits*). Na linguagem C, também podemos usar um **char** para armazenar valores numéricos inteiros, além de usá-lo para armazenar caracteres de texto. Para indicar um caractere de texto usamos apóstrofes. Veja um exemplo de programa que usa caracteres:

```
#include <stdio.h>
int main ()
{
      char Ch;
      Ch='D';
      printf ("%c",Ch);
      return(0);
}
```

No programa acima, **%c** indica que **printf()** deve colocar um caractere na tela. Como vimos anteriormente, um **char** também é usado para armazenar um número inteiro. Este número é conhecido como o código ASCII correspondente ao caractere. Veja o programa abaixo:

```
#include <stdio.h>
int main ()
{
      char Ch;
      Ch='D';
      printf ("%d",Ch); /* Imprime o caracter como inteiro */
      return(0);
}
```

Este programa vai imprimir o número 68 na tela, que é o código ASCII correspondente ao caractere 'D' (d maiúsculo).

Muitas vezes queremos ler um caractere fornecido pelo usuário. Para isto as funções mais usadas, quando se está trabalhando em ambiente DOS ou Windows, são **getch()** e **getche()**. Ambas retornam o caractere pressionado. **getche()** imprime o caractere na tela antes de retorná-lo e **getch()** apenas retorna o caractere pressionado sem imprimí-lo na tela. Ambas as funções podem ser encontradas no arquivo de cabeçalho **conio.h**. Geralmente estas funções não estão disponíveis em ambiente Unix (compiladores cc e gcc) e podem ser substituídas pela função scanf(), porém sem as mesmas funcionalidades. Eis um exemplo que usa a função **getch(),** e seu correspondente em ambiente Unix:

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

/* Não funcionará no Unix por causa da conio.h */

int main ()
{
```

```
      char Ch;
      Ch=getch();
      printf ("Voce pressionou a tecla %c",Ch);
      return(0);
}
```

Equivalente para o ambiente Unix do programa acima, sem usar getch():

```
#include <stdio.h>
int main ()
{
      char Ch;
      scanf("%c", &Ch);
      printf ("Voce pressionou a tecla %c",Ch);
      return(0);
}
```

A principal diferença da versão que utiliza getch() para a versão que não utiliza getch() é que no primeiro caso o usuário simplesmente aperta a tecla e o sistema lê diretamente a tecla pressionada. No segundo caso, é necessário apertar também a tecla <ENTER>.

3.2 Strings

No C uma string é um vetor de caracteres terminado com um caractere nulo. O caracter nulo é um caractere com valor inteiro igual a zero (código ASCII igual a 0). O terminador nulo também pode ser escrito usando a convenção de barra invertida do C como sendo '\0'. Embora o assunto vetores seja discutido posteriormente, veremos aqui os fundamentos necessários para que possamos utilizar as strings. Para declarar uma string, podemos usar o seguinte formato geral:

char nome_da_string[tamanho];

Isto declara um vetor de caracteres (uma string) com número de posições igual a *tamanho*. Note que, como temos que reservar um caractere para ser o terminador nulo, temos que declarar o comprimento da string como sendo, no mínimo, um caractere maior que a maior string que pretendemos armazenar. Vamos supor que declaremos uma string de 7 posições e coloquemos a palavra João nela. Teremos:

J	o	a	o	\0	...	...

No caso acima, as duas células não usadas têm valores indeterminados. Isto acontece porque o C *não* inicializa variáveis, cabendo ao programador esta tarefa. Portanto as únicas células que são inicializadas são as que contêm os caracteres 'J', 'o', 'a', 'o' e '\0' .

Se quisermos ler uma string fornecida pelo usuário podemos usar a função **gets()**. Um exemplo do uso desta função é apresentado abaixo. A função **gets()** coloca o terminador nulo na string, quando você aperta a tecla "Enter".

```
#include <stdio.h>
int main ()
{
      char string[100];
      printf ("Digite uma string: ");
      gets (string);
```

```
      printf ("\n\nVoce digitou %s",string);
      return(0);
}
```

Neste programa, o tamanho máximo da string que você pode entrar é uma string de 99 caracteres. Se você entrar com uma string de comprimento maior, o programa irá aceitar, mas os resultados podem ser desastrosos. Veremos porque posteriormente.

Como as strings são vetores de caracteres, para se acessar um determinado caracter de uma string, basta "indexarmos", ou seja, usarmos um índice para acessarmos o caracter desejado dentro da string. Suponha uma string chamada *str*. Podemos acessar a **segunda** letra de *str* da seguinte forma:

str[1] = 'a';

Por quê se está acessando a segunda letra e não a primeira? Na linguagem C, o índice ***começa em zero.*** Assim, a primeira letra da string sempre estará na posição 0. A segunda letra sempre estará na posição 1 e assim sucessivamente. Segue um exemplo que imprimirá a segunda letra da string "Joao", apresentada acima. Em seguida, ele mudará esta letra e apresentará a string no final.

```
#include <stdio.h>
int main()
{
      char str[10] = "Joao";
      printf("\n\nString: %s", str);
      printf("\nSegunda letra: %c", str[1]);
      str[1] = 'U';
      printf("\nAgora a segunda letra eh: %c", str[1]);
      printf("\n\nString resultante: %s", str);
      return(0);
}
```

Nesta string, o terminador nulo está na posição 4. Das posições 0 a 4, sabemos que temos caracteres válidos, e portanto podemos escrevê-los. Note a forma como inicializamos a string **str** com os caracteres 'J' 'o' 'a' 'o' e '\0' simplesmente declarando char str[10] = "Joao". Veremos, posteriormente que "Joao" (uma cadeia de caracteres entre aspas) é o que chamamos de string constante, isto é, uma cadeia de caracteres que está pré-carregada com valores que não podem ser modificados. Já a string str é uma string variável, pois podemos modificar o que nela está armazenado, como de fato fizemos.

No programa acima, **%s** indica que **printf()** deve colocar uma string na tela. Vamos agora fazer uma abordagem inicial às duas funções que já temos usado para fazer a entrada e saída.

3.3 printf

A função **printf()** tem a seguinte forma geral:

printf (string_de_controle,lista_de_argumentos);

Teremos, na string de controle, uma descrição de tudo que a função vai colocar na tela. A string de controle mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os códigos de controle, que usam a notação **%**. Na string de controle indicamos quais, de qual tipo e em que posição estão as variáveis a serem apresentadas. É muito importante que, para cada código de controle, tenhamos um argumento na lista de argumentos. Apresentamos agora alguns dos códigos **%**:

Código	Significado
%d	Inteiro
%f	Float
%c	Caractere
%s	String

Vamos ver alguns exemplos de **printf()** e o que eles exibem:

printf ("Teste %% %%") -> "Teste % %"

printf ("%f",40.345) -> "40.345"

printf ("Um caractere %c e um inteiro %d",'D',120) -> "Um caractere D e um inteiro 120"

printf ("%s e um exemplo","Este") -> "Este e um exemplo"

printf ("%s%d%%","Juros de ",10) -> "Juros de 10%"

Maiores detalhes sobre a função **printf()** (incluindo outros códigos de controle) serão vistos posteriormente, mas podem ser consultados de antemão pelos interessados.

3.4 scanf

O formato geral da função **scanf()** é:

scanf (string-de-controle,lista-de-argumentos);

Usando a função **scanf()** podemos pedir dados ao usuário. Um exemplo de uso, pode ser visto acima. Mais uma vez, devemos ficar atentos a fim de colocar o mesmo número de argumentos que o de códigos de controle na string de controle. Outra coisa importante é lembrarmos de colocar o **&** antes das variáveis da lista de argumentos. É impossível justificar isto agora, mas veremos depois a razão para este procedimento. Maiores detalhes sobre a função **scanf()** serão vistos posteriormente, mas podem ser consultados de antemão pelos interessados.

4. COMANDOS DE CONTROLE DE FLUXO

Os comandos de controle de fluxo são aqueles que permitem ao programador alterar a sequência de execução do programa. Vamos dar uma breve introdução a dois comandos de controle de fluxo. Outros comandos serão estudados posteriormente.

4.1 if

O comando **if** representa uma tomada de decisão do tipo "SE isto ENTÃO aquilo". A sua forma geral é:

if (condição) declaração;

A condição do comando **if** é uma expressão que será avaliada. Se o resultado for zero a declaração não será executada. Se o resultado for qualquer coisa diferente de zero a declaração será executada. A declaração pode ser um bloco de código ou apenas um comando. É interessante notar que, no caso da declaração ser um bloco de código, não é necessário (e nem permitido) o uso do **;** no final do bloco. Isto é uma regra geral para blocos de código. Abaixo apresentamos um exemplo:

```
#include <stdio.h>
int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
      if (num>10)
          printf ("\n\nO numero e maior que 10");
      if (num==10)
      {
            printf ("\n\nVoce acertou!\n");
            printf ("O numero e igual a 10.");
      }
      if (num<10)
          printf ("\n\nO numero e menor que 10");
      return (0);
}
```

No programa acima a expressão **num>10** é avaliada e retorna um valor diferente de zero, se verdadeira, e zero, se falsa. No exemplo, se num for maior que 10, será impressa a frase: "O número e maior que 10". Repare que, se o número for igual a 10, estamos executando dois comandos. Para que isto fosse possível, tivemos que agrupa-los em um bloco que se inicia logo após a comparação e termina após o segundo printf. Repare também que quando queremos testar igualdades usamos o operador **==** e não **=**. Isto porque o operador **=** representa *apenas* uma atribuição. Pode parecer estranho à primeira vista, mas se escrevêssemos

if (num=10) ... /* Isto esta errado */

o compilador iria *atribuir* o valor 10 à variável **num** e a expressão **num=10** iria retornar 10, fazendo com que o nosso valor de **num** fosse modificado e fazendo com que a declaração fosse executada sempre. Este problema gera erros frequentes entre iniciantes e, portanto, muita atenção deve ser tomada.

Os operadores de comparação são: **==** (igual)**, !=** (diferente de), **>** (maior que)**, <** (menor que)**, >=** (maior ou igual**), <=** (menor ou igual)**.**

4.2 switch

O comando if-else e o comando switch são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o if, mas o comando switch tem aplicações valiosas. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando switch é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos. Sua forma geral é:

```
switch (variável)
    {

    case constante_1:
    declaração_1;
    break;
    case constante_2:
    declaração_2;
    break;
    .
    .
    .
    case constante_n:
    declaração_n;
    break;
    default
    declaração_default;
    }
```

Podemos fazer uma analogia entre o switch e a estrutura if-else-if apresentada anteriormente. A diferença fundamental é que a estrutura switch não aceita expressões. Aceita apenas constantes. O switch testa a variável e executa a declaração cujo case corresponda ao valor atual da variável. A declaração default é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.

O comando break, faz com que o switch seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando switch. Se após a execução da declaração não houver um break, o programa continuará executando. Isto pode ser útil em algumas situações, mas eu recomendo cuidado. Veremos agora um exemplo do comando switch:

```
#include <stdio.h>

int main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);

      switch (num)
      {
            case 9:
                  printf ("\n\nO numero e igual a 9.\n");
                  break;
```

```
            case 10:
                 printf ("\n\nO numero e igual a 10.\n");
                break;
            case 11:
                 printf ("\n\nO numero e igual a 11.\n");
                 break;
            default:
                 printf ("\n\nO numero nao e nem 9 nem 10 nem 11.\n");
         }
      return(0);
}
```

4.3 for

O loop (laço) **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

for (inicialização;condição;incremento) declaração;

A declaração no comando for também pode ser um bloco ({ }) e neste caso o **;** é omitido. O melhor modo de se entender o loop **for** é ver de que maneira ele funciona "por dentro". O loop **for** é equivalente a se fazer o seguinte:

inicialização;
if (condição)
{
declaração;
incremento;
"Volte para o comando if"
}

Podemos ver que o **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Abaixo vemos um programa que coloca os primeiros 100 números na tela:

```
#include <stdio.h>
int main ()
{
      int count;
      for (count=1;count<=100;count=count+1)
           printf ("%d ",count);
      return(0);
}
```

Outro exemplo interessante é mostrado a seguir: o programa lê uma string e conta quantos dos caracteres desta string são iguais à letra 'c'

```
#include <stdio.h>
int main ()
{
      char string[100]; /* String, ate' 99 caracteres */
      int i, cont;
```

```
     printf("\n\nDigite uma frase: ");
     gets(string); /* Le a string */
     printf("\n\nFrase digitada:\n%s", string);
     cont = 0;
     for (i=0; string[i] != '\0'; i=i+1)
     {
          if ( string[i] == 'c' ) /* Se for a letra 'c' */
              cont = cont +1; /* Incrementa o contador de caracteres */
     }
     printf("\nNumero de caracteres c = %d", cont);
     return(0);
}
```

Note o teste que está sendo feito no for: o caractere armazenado em string[i] é comparado com '\0' (caractere final da string). Caso o caractere seja diferente de '\0', a condição é verdadeira e o bloco do for é executado. Dentro do bloco existe um if que testa se o caractere é igual a 'c'. Caso seja, o contador de caracteres c é incrementado.

Mais um exemplo, agora envolvendo caracteres:

```
/* Este programa imprime o alfabeto: letras maiúsculas  */

#include <stdio.h>
int main()
{
    char letra;
    for(letra = 'A' ; letra <= 'Z' ; letra =letra+1)
    printf("%c ", letra);
}
```

Este programa funciona porque as letras maiúsculas de A a Z possuem código inteiro sequencial.

4.4 while

O comando while tem a seguinte forma geral:

while (condição) declaração;

Assim como fizemos para o comando for, vamos tentar mostrar como o while funciona fazendo uma analogia. Então o while seria equivalente a:

```
if (condição)
   {
   declaração;
   "Volte para o comando if"
   }
```

Podemos ver que a estrutura while testa uma condição. Se esta for verdadeira a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do for, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo.

Vamos ver um exemplo do uso do while. O programa abaixo é executado enquanto i for menor que 100. Veja que ele seria implementado mais naturalmente com um for ...

```
#include <stdio.h>
int main ()
{
      int i = 0;
      while ( i < 100)
      {
            printf(" %d", i);
            i++;
       }
       return(0);
}
```

O programa abaixo espera o usuário digitar a tecla 'q' e só depois finaliza:

```
#include <stdio.h>
int main ()
{
      char Ch;
      Ch='\0';
      while (Ch!='q')
      {
            scanf("%c", &Ch);
      }
      return(0);
}
```

4.5 do-while

A terceira estrutura de repetição que veremos é o do-while de forma geral:

do
{
declaração;
} while (condição);

Mesmo que a declaração seja apenas um comando é uma boa prática deixar as chaves. O ponto-e- vírgula final é obrigatório. Vamos, como anteriormente, ver o funcionamento da estrutura do-while "por dentro":

Declaração:

if (condição) "Volta para a declaração"

Vemos pela análise do bloco acima que a estrutura do-while executa a declaração, testa a condição e, se esta for verdadeira, volta para a declaração. A grande novidade no comando do-while é que ele, ao contrário do for e do while, garante que a declaração será executada pelo menos uma vez.

Um dos usos da extrutura do-while é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme apresentado a seguir:

```
#include <stdio.h>
int main ()
{
      int  i;
      do
      {
            printf ("\n\nEscolha a fruta pelo numero:\n\n");
            printf ("\t(1)...Mamao\n");
            printf ("\t(2)...Abacaxi\n");
            printf ("\t(3)...Laranja\n\n");
            scanf("%d", &i);
      }
      while ((i<1)||(i>3));
      switch (i)
      {
            case 1:
                  printf ("\t\tVoce escolheu Mamao.\n");
                  break;
            case 2:
                  printf ("\t\tVoce escolheu Abacaxi.\n");
                  break;
            case 3:
                  printf ("\t\tVoce escolheu Laranja.\n");
                  break;
      }
      return(0);
}
```

4.6 break

Nós já vimos dois usos para o comando break: interrompendo os comandos switch e for. Na verdade, estes são os dois usos do comando break: ele pode quebrar a execução de um comando (como no caso do switch) ou interromper a execução de qualquer loop (como no caso do for, do while ou do do while). O break faz com que a execução do programa continue na primeira linha seguinte ao loop ou bloco que está sendo interrompido.

Observe que um break causará uma saída somente do laço mais interno. Por exemplo:

```
for(t=0; t<100; ++t)
{
      count=1;
      for(;;)
      {
            printf("%d", count);
            count++;
            if(count==10) break;
      }
}
```

O código acima imprimirá os números de 1 a 10 cem vezes na tela. Toda vez que o break é encontrado, o controle é devolvido para o laço for externo.

Outra observaçao é o fato que um break usado dentro de uma declaraçao switch afetará somente os dados relacionados com o switch e nao qualquer outro laço em que o switch estiver.

4.7 continue

O comando continue pode ser visto como sendo o oposto do break. Ele só funciona dentro de um loop. Quando o comando continue é encontrado, o loop pula para a próxima iteração, sem o abandono do loop, ao contrário do que acontecia no comando break.

O programa abaixo exemplifica o uso do continue:

```
#include <stdio.h>
int main()
{
      int opcao;
      while (opcao != 5)
      {
            printf("\n\n Escolha uma opcao entre 1 e 5: ");
            scanf("%d", &opcao);
            if ((opcao > 5)||(opcao <1))
               continue;
               /* Opcao invalida: volta ao inicio do loop */
            switch (opcao)
            {
                  case 1:
                              printf("\n --> Primeira opcao..");
                  break;
                  case 2:
                              printf("\n --> Segunda opcao..");
                  break;
                  case 3:
                              printf("\n --> Terceira opcao..");
                  break;
                  case 4:
                              printf("\n --> Quarta opcao..");
                  break;
                  case 5:
                              printf("\n --> Abandonando..");
                  break;
            }
        }
        return(0);
}
```

O programa acima ilustra uma aplicação simples para o continue. Ele recebe uma opção do usuario. Se esta opção for inválida, o continue faz com que o fluxo seja desviado de volta ao início do loop. Caso a opção escolhida seja válida o programa segue normalmente.

4.8 goto

Vamos mencionar o goto apenas para que você saiba que ele existe. O goto é o último comando de controle de fluxo. Ele pertence a uma classe à parte: a dos comandos de salto incondicional. O goto realiza um salto para um local especificado. Este local é determinado por um rótulo. Um rótulo, na linguagem C, é uma marca no programa. Você dá o nome que quiser a esta marca. Podemos tentar escrever uma forma geral:

```
nome_do_rótulo:
        ....
        goto nome_do_rótulo;
        ....
```

Devemos declarar o nome do rótulo na posição para a qual vamos dar o salto seguido de :. O goto pode saltar para um rótulo que esteja mais à frente ou para trás no programa. Uma observação importante é que o rótulo e o goto devem estar dentro da mesma função. Como exemplo do uso do goto vamos reescrever o equivalente ao comando for apresentado na seção equivalente ao mesmo:

```
inicialização;
início_do_loop:

if (condição)
    {
    declaração;
    incremento;
    goto início_do_loop;
    }
```

O comando goto deve ser utilizado com parcimônia, pois o abuso no seu uso tende a tornar o código confuso. O goto não é um comando necessário, podendo sempre ser substituído por outras estruturas de controle. Recomendamos que o goto nunca seja usado.

Existem algumas situações muito específicas onde o comando goto pode tornar um código mais fácil de se entender se ele for bem empregado. Um caso em que ele pode ser útil é quando temos vários loops e ifs aninhados e se queira, por algum motivo, sair destes loops e ifs todos de uma vez. Neste caso um goto resolve o problema mais elegantemente que vários breaks, sem contar que os breaks exigiriam muito mais testes. Ou seja, neste caso o goto é mais elegante e mais rápido.

O exemplo da página anterior pode ser reescrito usando-se o goto:

```
#include <stdio.h>

int main()
{
      int opcao;

      while (opcao != 5)
      {
            REFAZ: printf("\n\n Escolha uma opcao entre 1 e 5: ");
                  scanf("%d", &opcao);
             if ((opcao > 5)||(opcao <1))
                 goto REFAZ;
```

```
            switch (opcao)
            {
                  case 1:
                              printf("\n --> Primeira opcao..");
                              break;
                  case 2:
                              printf("\n --> Segunda opcao..");
                              break;
                  case 3:
                              printf("\n --> Terceira opcao..");
                              break;
                  case 4:
                              printf("\n --> Quarta opcao..");
                              break;
                  case 5:
                              printf("\n --> Abandonando..");
                              break;
              }
       }
      return(0);
}
```

4.9 exit

O protótipo da função **exit()** cujo é:

```
void exit (int codigo_de_retorno);
```

Para utilizá-la deve-se colocar um include para o arquivo de cabeçalho stdlib.h. Esta função aborta a execução do programa. Pode ser chamada de qualquer ponto no programa e faz com que o programa termine e retorne, para o sistema operacional, o código_de_retorno. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um número não nulo no caso de ter ocorrido um problema.

5. VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

5.1 Nomes de Variáveis

As variáveis no C podem ter qualquer nome se duas condições forem satisfeitas: o nome deve começar com uma letra ou sublinhado (_) e os caracteres subsequentes devem ser letras, números ou sublinhado (_). Há apenas mais duas restrições: o nome de uma variável não pode ser igual a uma palavra reservada, nem igual ao nome de uma função declarada pelo programador, ou pelas bibliotecas do C. Variáveis de até 32 caracteres são aceitas. Mais uma coisa: é bom sempre lembrar que o C é "case sensitive" e portanto deve-se prestar atenção às maiúsculas e minúsculas.
Quanto aos nomes de variáveis:

- É uma prática tradicional do C, usar letras minúsculas para nomes de variáveis e maiúsculas para nomes de constantes. Isto facilita na hora da leitura do código;
- Quando se escreve código usando nomes de variáveis em português, evita-se possíveis conflitos com nomes de rotinas encontrados nas diversas bibliotecas, que são em sua maioria absoluta, palavras em inglês.

5.2 Os Tipos do C

O C tem 5 tipos básicos: **char**, **int**, **float**, **void, double**. Destes não vimos ainda os dois últimos: O **double** é o ponto flutuante duplo e pode ser visto como um ponto flutuante com muito mais precisão. O void é o tipo vazio, ou um "tipo sem tipo". A aplicação deste "tipo" será vista posteriormente.

Para cada um dos tipos de variáveis existem os modificadores de tipo. Os modificadores de tipo do C são quatro: **signed**, **unsigned**, **long** e **short**. Ao **float** não se pode aplicar nenhum e ao **double** pode-se aplicar apenas o **long**. Os quatro modificadores podem ser aplicados a inteiros. A intenção é que **short** e **long** devam prover tamanhos diferentes de inteiros onde isto for prático. Inteiros menores (**short**) ou maiores (**long**). **int** normalmente terá o tamanho natural para uma determinada máquina. Assim, numa máquina de 16 bits, **int** provavelmente terá 16 bits. Numa máquina de 32, **int** deverá ter 32 bits. Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que **shorts ints** e **ints** devem ocupar pelo menos 16 bits, **longs ints** pelo menos 32 bits, e **short int** não pode ser maior que **int**, que não pode ser maior que **long int**. O modificador **unsigned** serve para especificar variáveis sem sinal. Um **unsigned int** será um inteiro que assumirá apenas valores positivos. A seguir estão listados os tipos de dados permitidos e seu valores máximos e mínimos em um compilador típico para um hardware de 16 bits. Também nesta tabela está especificado o formato que deve ser utilizado para ler os tipos de dados com a função scanf():

Tipo	Num de bits	Formato para leitura com scanf	Intervalo	
			Inicio	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	16	%i	-32.768	32.767
unsigned int	16	%u	0	65.535
signed int	16	%i	-32.768	32.767
short int	16	%hi	-32.768	32.767
unsigned short int	16	%hu	0	65.535
signed short int	16	%hi	-32.768	32.767
long int	32	%li	-2.147.483.648	2.147.483.647

signed long int	32	%li	-2.147.483.648	2.147.483.647
unsigned long int	32	%lu	0	4.294.967.295
float	32	%f	3,4E-38	3.4E+38
double	64	%lf	1,7E-308	1,7E+308
long double	80	%Lf	3,4E-4932	3,4E+4932

O tipo **long double** é o tipo de ponto flutuante com maior precisão. É importante observar que os intervalos de ponto flutuante, na tabela acima, estão indicados em faixa de *expoente*, mas os números podem assumir valores tanto positivos quanto negativos.

5.3 Declaração e Inicialização de Variáveis

As variáveis no C devem ser declaradas antes de serem usadas. A forma geral da declaração de variáveis é:

tipo_da_variável lista_de_variáveis;

As variáveis da lista de variáveis terão todas o mesmo tipo e deverão ser separadas por vírgula. Como o tipo default do C é o **int**, quando vamos declarar variáveis **int** com algum dos modificadores de tipo, basta colocar o nome do modificador de tipo. Assim um **long** basta para declarar um **long int**.

Por exemplo, as declarações

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

declaram duas variáveis do tipo **char** (ch e letra), uma variavel **long int** (count) e um **float** pi.

Há três lugares nos quais podemos declarar variáveis. O primeiro é fora de todas as funções do programa. Estas variáveis são chamadas **variáveis globais** e podem ser usadas a partir de qualquer lugar no programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as vêem. O segundo lugar no qual se pode declarar variáveis é **no início** de um bloco de código. Estas variáveis são chamadas **locais** e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável, dentro do bloco no qual foram declaradas. O terceiro lugar onde se pode declarar variáveis é na **lista de parâmetros** de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas.

Veja o programa abaixo:

```
#include <stdio.h>

int contador;
int func1(int j)
{
/* aqui viria o código da funcao ...  */
}
int main()
{
      char condicao;
      int i;

      for (i=0; i<100; i=i+1)
      {
            float f2;
```

```
            func1(i);
      }
      return(0);
}
```

A variável *contador* é uma variável global, e é acessível de qualquer parte do programa. As variáveis *condição* e *i,* só existem dentro de main(), isto é são variáveis locais de main. A variável float *f2* é um exemplo de uma variável de bloco, isto é, ela somente é conhecida dentro do bloco do for, pertencente à função main. A variável inteira *j* é um exemplo de declaração na lista de parâmetros de uma função (a função *func1*).

As regras que regem *onde* uma variável é válida chamam-se regras de *escopo* da variável. Há mais dois detalhes que devem ser ressaltados. Duas variáveis globais não podem ter o mesmo nome. O mesmo vale para duas variáveis locais de uma mesma função. Já duas variáveis locais, de funções diferentes, podem ter o mesmo nome sem perigo algum de conflito.

Podemos inicializar variáveis no momento de sua declaração. Para fazer isto podemos usar a forma geral

tipo_da_variável nome_da_variável = constante;

Isto é importante pois quando o C cria uma variável ele *não* a inicializa. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor *indefinido* e que não pode ser utilizado para nada. *Nunca* presuma que uma variável declarada vale zero ou qualquer outro valor. Exemplos de inicialização são dados abaixo:

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

Ressalte-se novamente que, em C, uma variável tem que ser declarada no início de um bloco de código. Assim, o programa a seguir não é válido em C (embora seja válido em C++).

```
int  main()
{
        int i;
        int j;
        j = 10;
        int k = 20; /* Esta declaração de variável não é válida*/
        return(0);
}
```

5.4 Constantes

Constantes são valores que são mantidos fixos pelo compilador. Já usamos constantes neste curso. São consideradas constantes, por exemplo, os números e caracteres como 45.65 ou 'n', etc...

5.4.1 Constantes dos tipos básicos

Abaixo vemos as constantes relativas aos tipos básicos do C:

Tipo de Dado	Exemplos de Constantes
char	'b' '\n' '\0'
int	2 32000 -130
long int	100000 -467

short int	100 -30
unsigned int	50000 35678
float	0.0 23.7 -12.3e-10
double	12546354334.0 -0.0000034236556

5.4.2 Constantes hexadecimais e octais

Muitas vezes precisamos inserir constantes hexadecimais (base dezesseis) ou octais (base oito) no nosso programa. O C permite que se faça isto. As constantes hexadecimais começam com 0x. As constantes octais começam em 0. Alguns exemplos:

Constante	Tipo
0xEF	Constante Hexadecimal (8 bits)
0x12A4	Constante Hexadecimal (16 bits)
03212	Constante Octal (12 bits)
034215432	Constante Octal (24 bits)

Nunca escreva portanto 013 achando que o C vai compilar isto como se fosse 13. Na linguagem C 013 é diferente de 13!

5.4.3 Constantes strings

Já mostramos como o C trata strings. Vamos agora alertar para o fato de que uma string **"Joao"** é na realidade uma constante string. Isto implica, por exemplo, no fato de que **'t'** é diferente de **"t"**, pois **'t'** é um **char** enquanto que **"t"** é uma constante string com dois **chars** onde o primeiro é **'t'** e o segundo é **'\0'**.

5.4.4 Constantes de barra invertida

O C utiliza, para nos facilitar a tarefa de programar, vários códigos chamados códigos de barra invertida. Estes são caracteres que podem ser usados como qualquer outro. Uma lista com alguns dos códigos de barra invertida é dada a seguir:

Código	Significado
\b	Retrocesso ("back")
\f	Alimentação de formulário ("form feed")
\n	Nova linha ("new line")
\t	Tabulação horizontal ("tab")
\"	Aspas
\'	Apóstrofo
\0	Nulo (0 em decimal)
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro ("beep")
\N	Constante octal (N é o valor da constante)
\xN	Constante hexadecimal (N é o valor da constante)

5.5 Operadores Aritméticos e de Atribuição

Os operadores aritméticos são usados para desenvolver operações matemáticas. A seguir apresentamos a lista dos operadores aritméticos do C:

Operador	Ação
+	Soma (inteira e ponto flutuante)
-	Subtração ou Troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
--	Decremento (inteiro e ponto flutuante)

O C possui operadores unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma é um operador binário pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores **-** (subtração), *****, **/** e **%**. O operador **-** como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por -1.

O operador / (divisão) quando aplicado a variáveis inteiras, nos fornece o resultado da divisão inteira; quando aplicado a variáveis em ponto flutuante nos fornece o resultado da divisão "real". O operador % fornece o resto da divisão de dois inteiros. Assim seja o seguinte trecho de código:

```
int a = 17, b = 3;
int x, y;
float z = 17. , z1, z2;
x = a / b;
y = a % b;
z1 = z / b;
z2 = a/b;
```

Ao final da execução destas linhas, os valores calculados seriam x = 5, y = 2, z1 = 5.666666 e z2 = 5.0 . Note que, na linha correspondente a z2, primeiramente é feita uma divisão inteira (pois os dois operandos são inteiros). Somente após efetuada a divisão é que o resultado é atribuído a uma variável float.

Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar, a variável sobre a qual estão aplicados, de 1. Então

```
x++;
x--;
```

são equivalentes a

```
x=x+1;
x=x-1;
```

Estes operadores podem ser pré-fixados ou pós- fixados. A diferença é que quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Então, em

```
x=23;
y=x++;
```

teremos, no final, **y=23** e **x=24**. Em

```
x=23;
y=++x;
```

teremos, no final, **y=24** e **x=24**. Uma curiosidade: a linguagem de programação C++ tem este nome pois ela seria um "incremento" da linguagem C padrão. A linguagem C++ é igual a linguagem C só que com extensões que permitem a programação orientada a objeto, o que é um recurso extra.

O operador de atribuição do C é o **=**. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto ele retorna o valor que ele atribuiu. Isto faz com que as seguintes expressões sejam válidas:

```
x=y=z=1.5;          /* Expressao 1 */

if (k=w) ...        /* Expressao 2 */
```

A expressão 1 é válida, pois quando fazemos **z=1.5** ela retorna 1.5, que é passado adiante, fazendo y = 1.5 e posteriormente x = 1.5. A expressão 2 será verdadeira se **w** for diferente de zero, pois este será o valor retornado por **k=w**. Pense bem antes de usar a expressão dois, pois ela pode gerar erros de interpretação. Você *não* está comparando **k** e **w**. Você está atribuindo o valor de **w** a **k** e usando este valor para tomar a decisão.

5.6 Operadores Relacionais e Lógicos

Os operadores relacionais do C realizam ***comparações*** entre variáveis. São eles:

Operador	Ação
>	Maior do que
>=	Maior ou igual a
<	Menor do que
<=	Menor ou igual a
==	Igual a
!=	Diferente de

Os operadores relacionais retornam verdadeiro (1) ou falso (0). Para verificar o funcionamento dos operadores relacionais, execute o programa abaixo:

```
/* Este programa ilustra o funcionamento dos operadores relacionais. */
#include <stdio.h>

int main()
{
    int i, j;
    printf("\nEntre com dois numeros inteiros: ");
    scanf("%d%d", &i, &j);
    printf("\n%d == %d é %d\n", i, j, i==j);
    printf("\n%d != %d é %d\n", i, j, i!=j);
    printf("\n%d <= %d é %d\n", i, j, i<=j);
    printf("\n%d >= %d é %d\n", i, j, i>=j);
    printf("\n%d < %d é %d\n", i, j, i<j);
    printf("\n%d > %d é %d\n", i, j, i>j);
    return(0);
}
```

Você pode notar que o resultado dos operadores relacionais é sempre igual a 0 (falso) ou 1 (verdadeiro).

Para fazer ***operações com valores lógicos*** (verdadeiro e falso) temos ***os operadores lógicos***:

Operador	**Ação**
&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

Usando os operadores relacionais e lógicos podemos realizar uma grande gama de testes. A tabela-verdade destes operadores é dada a seguir:

p	**q**	**p AND q**	**p OR q**
falso	falso	falso	falso
falso	verdadeiro	falso	verdadeiro
verdadeiro	falso	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro	verdadeiro

O programa a seguir ilustra o funcionamento dos operadores lógicos. Compile-o e faça testes com vários valores para i e j:

```
#include <stdio.h>
int main()
{
    int i, j;
    printf("informe dois números(cada um sendo 0 ou 1): ");
    scanf("%d%d", &i, &j);
    printf("%d AND %d é %d\n", i, j, i && j);
    printf("%d OR %d é %d\n", i, j, i || j);
    printf("NOT %d é %d\n", i, !i);
}
```

Exemplo: No trecho de programa abaixo a operação j++ será executada, pois o resultado da expressão lógica é verdadeiro:

int i = 5, j =7;
if ((i > 3) && (j <= 7) && (i != j)) j++;
V AND V AND V = V

Mais um exemplo. O programa abaixo, imprime na tela somente os números pares entre 1 e 100, apesar da variação de i ocorrer de 1 em 1:

```
/* Imprime os números pares entre 1 e 100. */
#include <stdio.h>
int main()
{
    int i;
    for(i=1; i<=100; i++)
       if(!(i%2)) printf("%d ",I);
}
/* é invertido pelo ! */
```

5.7 Operadores Lógicos Bit a Bit

O C permite que se faça ***operações lógicas "bit-a- bit"*** em números. Ou seja, neste caso, o número é representado por sua forma binária e as operações são feitas em cada bit dele. Imagine um número inteiro de 16 bits, a variável i, armazenando o valor 2. A representação binária de i, será: 0000000000000010 (quinze zeros e um único 1 na segunda posição da direita para a esquerda). Poderemos fazer operações em cada um dos bits deste número. Por exemplo, se fizermos a negação do número (operação binária NOT, ou operador binário ~ em C), isto é, ~i, o número se transformará em 1111111111111101. As operações binárias ajudam programadores que queiram trabalhar com o computador em "baixo nível". As operações lógicas bit a bit só podem ser usadas nos tipos **char**, **int** e **long int**. Os operadores são:

Operador	Ação
&	AND
|	OR
^	XOR (OR exclusivo)
~	NOT
>>	Deslocamento de bits a direita
<<	Deslocamento de bits a esquerda

Os operadores **&**, **|**, **^** e **~** são as operações lógicas bit a bit. A forma geral dos operadores de deslocamento é:

valor>>número_de_deslocamentos

valor<<número_de_deslocamentos

O número_de_deslocamentos indica o quanto cada bit irá ser deslocado. Por exemplo, para a variável i anterior, armazenando o número 2:

```
i << 3;
```

fará com que i agora tenha a representação binária: 0000000000010000, isto é, o valor armazenado em i passa a ser igual a 16.

5.8 Expressões

Expressões são combinações de variáveis, constantes e operadores. Quando montamos expressões temos que levar em consideração a ordem com que os operadores são executados, conforme a tabela de precedências da linguagem C.

Exemplos de expressões:

```
Anos=Dias/365.25;
i = i+3;
c= a*b + d/e;
c= a*(b+d)/e;
```

5.8.1 Conversão de tipos em expressões

Quando o C avalia expressões onde temos variáveis de tipos diferentes o compilador verifica se as conversões são possíveis. Se não são, ele não compilará o programa, dando uma mensagem de erro. Se as conversões forem possíveis ele as faz, seguindo as regras abaixo:

1. Todos os **char**s e **short int**s são convertidos para **int**s. Todos os **float**s são convertidos para **double**s.
2. Para pares de operandos de tipos diferentes: se um deles é **long double** o outro é convertido para **long double**; se um deles é **double** o outro é convertido para **double**;

se um é **long** o outro é convertido para **long**; se um é **unsigned** o outro é convertido para **unsigned**.

5.8.2 Expressões que Podem ser Simplificadas

O C admite as seguintes equivalências, que podem ser usadas para simplificar expressões ou para facilitar o entendimento de um programa:

Expressão Original	Expressão Equivalente
x=x+k;	x+=k;
x=x-k;	x-=k;
x=x*k;	x*=k;
x=x/k;	x/=k;
x=x>>k;	x>>=k;
x=x<<k;	x<<=k;
x=x&k;	x&=k;

5.8.3 Tabela de Precedências do C

Esta é a tabela de precedência dos operadores em C.

Maior precedência	() [] ->
↓	! ~ ++ -- . -(unário) (cast) *(unário) &(unário) sizeof
	* / %
	+ -
	<< >>
	<<= >>=
	== !=
	&
	^
	|
	&&
	||
	?
Menor precedência	= += -= *= /=

Uma dica aos iniciantes: Você não precisa saber toda a tabela de precedências de cor. É útil que você conheça as principais relações, mas é aconselhável que ao escrever o seu código, você tente isolar as expressões com parênteses, para tornar o seu programa mais legível.

5.8.4 Modeladores (Casts)

Um modelador é aplicado a uma expressão. Ele *força* a mesma a ser de um tipo especificado. Sua forma geral é:

(tipo)expressão Um exemplo:

```
#include <stdio.h>
int main ()
```

```
{
      int num;
      float f;
      num=10;
      f=(float)num/7;
      printf ("%f",f);
      return(0);
}
```

Se não tivéssemos usado o modelador no exemplo acima o C faria uma divisão inteira entre 10 e 7. O resultado seria 1 (um) e este seria depois convertido para **float** mas continuaria a ser 1.0. Com o modelador temos o resultado correto.

6. MATRIZES E STRINGS

6.1 Vetores

Vetores nada mais são que matrizes unidimensionais. Vetores são uma estrutura de dados muito utilizada. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Para se declarar um vetor podemos utilizar a seguinte forma geral:

tipo_da_variável nome_da_variável [tamanho];

Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

float exemplo [20];

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

exemplo[0]
exemplo[1]
.
.
.
exemplo[19]

Mas ninguém o impede de escrever:

exemplo[30]
exemplo[103]

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que *você* deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter variáveis sobreescritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
#include <stdio.h>
int main ()
{
      int num[100];  /* Declara um vetor de inteiros de 100 posicoes */
      int count=0;
      int totalnums;
      do
      {
            printf ("\nEntre com um numero (-999 p/ terminar): ");
            scanf ("%d",&num[count]);
            count++;
      }
      while (num[count-1]!=-999);
      totalnums=count-1;
      printf ("\n\n\n\t Os números que você digitou foram:\n\n");
      for (count=0;count<totalnums;count++)
           printf (" %d",num[count]);
      return(0);
}
```

No exemplo acima, o inteiro *count* é inicializado em 0. O programa pede pela entrada de números até que o usuário entre com o Flag -999. Os números são armazenados no vetor

num. A cada número armazenado, o contador do vetor é incrementado para na próxima iteração escrever na próxima posição do vetor. Quando o usuário digita o flag, o programa abandona o primeiro loop e armazena o total de números gravados. Por fim, todos os números são impressos. É bom lembrar aqui que nenhuma restrição é feita quanto a quantidade de números digitados. Se o usuário digitar mais de 100 números, o programa tentará ler normalmente, mas o programa os escreverá em uma parte não alocada de memória, pois o espaço alocado foi para somente 100 inteiros. Isto pode resultar nos mais variados erros no instante da execução do programa.

6.2 Strings

Strings são vetores de **chars**. Nada mais e nada menos. As strings são o uso mais comum para os vetores. Devemos apenas ficar atentos para o fato de que as strings têm o seu último elemento como um '\0'. A declaração geral para uma string é:

char nome_da_string [tamanho];

Devemos lembrar que o tamanho da string deve incluir o '\0' final. A biblioteca padrão do C possui diversas funções que manipulam strings. Estas funções são úteis pois, não se pode, por exemplo, igualar duas strings:

```
string1=string2;       /* NÃO faça isto */
```

Fazer isto é um desastre. Quando você terminar de ler a seção que trata de ponteiros você entenderá porquê. As strings devem ser igualadas elemento a elemento.

Quando vamos fazer programas que tratam de string muitas vezes podemos fazer bom proveito do fato de que uma string termina com '\0' (isto é, o número inteiro 0). Veja, por exemplo, o programa abaixo que serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):

```
#include <stdio.h>
int main ()
{
      int count;
      char str1[100],str2[100];
      ...       /* Aqui o programa le str1 que sera copiada para str2 */
      for (count=0;str1[count];count++)
              str2[count]=str1[count];
     str2[count]='\0';
      ....             /* Aqui o programa continua */
}
```

A condição no loop **for** acima é baseada no fato de que a string que está sendo copiada termina em '\0'. Quando o elemento encontrado em **str1[count]** é o '\0', o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

6.2.1 gets

A função **gets()** lê uma string do teclado. Sua forma geral é:

gets (nome_da_string);

O programa abaixo demonstra o funcionamento da função **gets()**:

```
#include <stdio.h>

int main ()
```

```
{
      char string[100];
      printf ("Digite o seu nome: ");
      gets (string);
      printf ("\n\n Ola %s",string);
      return(0);
}
```

Repare que é válido passar para a função **printf()** o nome da string. Você verá mais adiante porque isto é válido. Como o primeiro argumento da função **printf()** é uma string também é válido fazer:

printf (string);

isto simplesmente imprimirá a string.

6.2.2 strcpy

Sua forma geral é:

strcpy (string_destino,string_origem);

A função **strcpy()** copia a string-origem para a string- destino. Seu funcionamento é semelhante ao da rotina apresentada na seção anterior. As funções apresentadas nestas seções estão no arquivo cabeçalho **string.h**. A seguir apresentamos um exemplo de uso da função **strcpy()**:

```
#include <stdio.h>
#include <string.h>
int main ()
{
      char str1[100],str2[100],str3[100];
      printf ("Entre com uma string: ");
      gets (str1);
      strcpy (str2,str1);   /* Copia str1 em str2 */
      strcpy (str3,"Voce digitou a string ");
      printf ("\n\n%s%s",str3,str2);
      return(0);
}
```

6.2.3 strcat

A função **strcat()** tem a seguinte forma geral:

strcat (string_destino,string_origem);

A string de origem permanecerá inalterada e será anexada ao fim da string de destino. Um exemplo:

```
#include <stdio.h>
#include <string.h>
int main ()
{
      char str1[100],str2[100];
      printf ("Entre com uma string: ");
      gets (str1);
      strcpy (str2,"Voce digitou a string ");
```

```
      strcat (str2,str1);
      /* str2 armazenara' Voce digitou a string + o conteudo de str1 */
      printf ("\n\n%s",str2);
      return(0);
}
```

6.2.4 strlen

Sua forma geral é:
strlen (string);
A função **strlen()** retorna o comprimento da string fornecida. O terminador nulo não é contado. Isto quer dizer que, de fato, o comprimento do vetor da string deve ser um a mais que o inteiro retornado por **strlen()**. Um exemplo do seu uso:

```
#include <stdio.h>
#include <string.h>
int main ()
{
      int size;
      char str[100];
      printf ("Entre com uma string: ");
      gets (str);
      size=strlen (str);
      printf ("\n\nA string que voce digitou tem tamanho %d",size);
      return(0);
}
```

6.2.5 strcmp

Sua forma geral é:
strcmp (string1,string2);
A função **strcmp()** compara a string 1 com a string 2. Se as duas forem idênticas a função retorna zero. Se elas forem diferentes a função retorna não-zero. Um exemplo da sua utilização:

```
#include <stdio.h>
#include <string.h>
int main ()
{
      char str1[100],str2[100];
      printf ("Entre com uma string: ");
      gets (str1);
      printf ("\n\nEntre com outra string: ");
      gets (str2);
      if (strcmp(str1,str2))
          printf ("\n\nAs duas strings são diferentes.");
      else
          printf ("\n\nAs duas strings são iguais.");
      return(0);
}
```

6.3 Matrizes

6.3.1 Matrizes bidimensionais

Já vimos como declarar matrizes unidimensionais (vetores). Vamos tratar agora de matrizes bidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

tipo_da_variável nome_da_variável [altura][largura];

É muito importante ressaltar que, nesta estrutura, o índice da esquerda indexa as linhas e o da direita indexa as colunas. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador. Abaixo damos um exemplo do uso de uma matriz:

```
#include <stdio.h>
int 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++;
          }
return(0);
}
```

No exemplo acima, a matriz **mtrx** é preenchida, sequencialmente por linhas, com os números de 1 a 200. Você deve entender o funcionamento do programa acima antes de prosseguir.

6.3.2 Matrizes de strings

Matrizes de strings são matrizes bidimensionais. Imagine uma string. Ela é um vetor. Se fizermos um vetor de strings estaremos fazendo uma lista de vetores. Esta estrutura é uma matriz bidimensional de **chars**. Podemos ver a forma geral de uma matriz de strings como sendo:

char nome_da_variável [num_de_strings][compr_das_strings];

Aí surge a pergunta: como acessar uma string individual? Fácil. É só usar apenas o primeiro índice. Então, para acessar uma determinada string faça:

nome_da_variável [índice]

Aqui está um exemplo de um programa que lê 5 strings e as exibe na tela:

```
#include <stdio.h>
int main ()
{
      char strings [5][100];
      int count;
      for (count=0;count<5;count++)
```

```
     {
          printf ("\n\nDigite uma string: ");
          gets (strings[count]);
     }
     printf ("\n\n\nAs strings que voce digitou foram:\n\n");
     for (count=0;count<5;count++)
          printf ("%s\n",strings[count]);
     return(0);
}
```

6.3.3 Matrizes multidimensionais

O uso de matrizes multidimensionais na linguagem C é simples. Sua forma geral é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

Uma matriz N-dimensional funciona basicamente como outros tipos de matrizes. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

6.3.4 Inicialização

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como inicialização é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN] = {lista_de_valores};

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
char str [10] = "Joao";
char str_vect [3][10] = { "Joao", "Maria", "Jose" };
```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde **matrx** está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o ; no final da inicialização.

6.3.5 Inicialização sem especificação de tamanho

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho *a priori*. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```
char mess [] = "Linguagem C: flexibilidade e poder.";
int matrx [][2] = { 1,2,2,4,3,6,4,8,5,10 };
```

No primeiro exemplo, a string mess terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

7. PONTEIROS

7.1 Como Funcionam os Ponteiros

Os **int**s guardam inteiros. Os **float**s guardam números de ponto flutuante. Os **char**s guardam caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Voce anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.

No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro **int** aponta para um inteiro, isto é, guarda o endereço de um inteiro.

7.2 Declarando e Utilizando 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;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstânicias pode levar a um travamento do micro, ou a algo pior. *O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!* Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count=10;
int *pt;
pt=&count;
```

Criamos um inteiro **count** com o valor 10 e um apontador para um inteiro **pt**. A expressão **&count** nos dá o endereço de count, o qual armazenamos em **pt**. Simples, não é? Repare que *não* alteramos o valor de **count**, que continua valendo 10.

Como nós colocamos um endereço em **pt**, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de **count** usando **pt**. Para tanto vamos usar o operador "inverso" do operador **&**. É o operador *****. No exemplo acima, uma vez que fizemos **pt=&count**

a expressão ***pt** é equivalente ao próprio **count**. Isto significa que, se quisermos mudar o valor de count para 12, basta fazer ***pt=12**.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador ***** ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador ***** (multiplicação) não é o mesmo operador que o ***** (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário pré-fixado.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
int 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%d\n",valor);
      printf ("Endereco para onde o ponteiro aponta: %p\n",p);
      printf ("Valor da variavel apontada: %d\n",*p);
      return(0);
}

#include <stdio.h>
int main ()
{
      int num,*p;
      num=55;
      p=&num;      /* Pega o endereco de num */
      printf ("\nValor inicial: %d\n",num);
      *p=100; /* Muda o valor de num de uma maneira indireta */
      printf ("\nValor final: %d\n",num);
      return(0);
}
```

Nos exemplos acima vemos um primeiro exemplo do funcionamento dos ponteiros. No primeiro exemplo, o código **%p** usado na função **printf()** indica à função que ela deve imprimir um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se temos dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1=p2**. Repare que estamos fazendo com que **p1** aponte para o mesmo lugar que **p2**. Se quisermos que a variável apontada por **p1** tenha o mesmo conteúdo da variável apontada por

p2 devemos fazer ***p1=*p2**. Basicamente, depois que se aprende a usar os dois operadores (**&** e ***) fica fácil entender operações com ponteiros.

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **char*** ele anda 1 byte na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que **p** é um ponteiro, as operações são escritas como:

```
p++;
p--;
```

Mais uma vez insisto. Estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15;  ou  p+=15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira. Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (**==** e **!=**). No caso de operações do tipo **>**, **<**, **>=** e **<=** estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*. Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
p1>p2
```

Há entretanto operações que você *não* pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair **float**s ou **double**s de ponteiros.

7.2.1 Ponteiros e Vetores

Veremos nestas seções que ponteiros e vetores têm uma ligação muito forte.

7.2.1.1 Vetores como ponteiros

Vamos dar agora uma idéia de como o C trata vetores.

Quando você declara uma matriz da seguinte forma:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo

O compilador então aloca este número de bytes em um espaço livre de memória. O *nome da variável* que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*. Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz.

Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

nome_da_variável[índice]

Isto pode ser facilmente explicado desde que você entenda que a notação acima é *absolutamente equivalente* a se fazer:

(nome_da_variável+índice)

Agora podemos entender como é que funciona um vetor! Vamos ver o que podemos tirar de informação deste fato. Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, ***nome_da_variável** e então devemos ter um índice igual a zero. Então sabemos que:

**nome_da_variável* é equivalente a *nome_da_variável[0]*

Outra coisa: apesar de, na maioria dos casos, não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. Isto explica também porque o C não verifica a validade dos índices. Ele *não* sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura sequencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
int main ()
{
      float matrx [50][50];
      int i,j;
      for (i=0;i<50;i++)
           for (j=0;j<50;j++)
               matrx[i][j]=0.0;
      return(0);
}
```

Podemos reescrevê-lo usando ponteiros:

```
int main ()
{
      float matrx [50][50];
      float *p;
      int count;
      p=matrx[0];
      for (count=0;count<2500;count++)
      {
            *p=0.0;
            p++;
      }
      return(0);
}
```

No primeiro programa, *cada* vez que se faz **matrx[i][j]** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.

Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Seja:

```
int vetor[10];
```

```
int *ponteiro, i;
ponteiro = &i;

/* as operacoes a seguir sao invalidas */

vetor = vetor + 2;      /* ERRADO: vetor nao e' variavel */
vetor++;                /* ERRADO: vetor nao e' variavel */
vetor = ponteiro;       /* ERRADO: vetor nao e' variavel */
```

Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que vetor não é um Lvalue. Lvalue, significa "Left value", um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, isto é, uma variável. Outros compiladores dirão que tem-se "incompatible types in assignment", tipos incompatíveis em uma atribuição.

```
/* as operacoes abaixo sao validas */

    ponteiro = vetor;      /* CERTO: ponteiro e' variavel */
    ponteiro = vetor+2;    /* CERTO: ponteiro e' variavel */
```

7.2.1.2 Ponteiros como vetores

Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como consequência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>
int main ()
{
     int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
     int *p;
     p=matrx;
     printf ("O terceiro elemento do vetor e: %d",p[2]);
     return(0);
}
```

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

7.2.2 Strings

Seguindo o raciocínio acima, nomes de strings, são do tipo **char***. Isto nos permite escrever a nossa função **StrCpy()**, que funcionará de forma semelhante à função **strcpy()** da biblioteca:

```
#include <stdio.h>

void StrCpy (char *destino,char *origem)
{
        while (*origem)
        {
              *destino=*origem;
              origem++;
              destino++;
```

```
        }
       *destino='\0';
}

int 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);
     return(0);
}
```

Há vários pontos a destacar no programa acima. Observe que podemos passar ponteiros como argumentos de funções. Na verdade é assim que funções como **gets()** e **strcpy()** funcionam. Passando o ponteiro você possibilita à função *alterar* o conteúdo das strings. Você já estava passando os ponteiros e não sabia. No comando **while (*origem)** estamos usando o fato de que a string termina com '\0' como critério de parada. Quando fazemos **origem++** e **destino++** o leitor poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que recomendei que se deveria fazer, no final de uma seção anterior. O que o leitor talvez não saiba ainda (e que será estudado em detalhe mais adiante) é que, no C, são passados para as funções *cópias* dos argumentos. Desta maneira, quando alteramos o ponteiro **origem** na função **StrCpy()** o ponteiro **str2** permanece inalterado na função **main()**.

7.2.3 Endereços de elementos de vetores

Nesta seção vamos apenas ressaltar que a notação

&nome_da_variável[índice]

é válida e retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a nome_da_variável + indice. É interessante notar que, como consequência, o ponteiro **nome_da_variável** tem o endereço **&nome_da_variável[0]**, que indica onde na memória está guardado o valor do primeiro elemento do vetor.

7.2.4 Vetores de ponteiros

Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

```
int *pmatrx [10];
```

No caso acima, **pmatrx** é um vetor que armazena 10 ponteiros para inteiros.

7.2.5 Ponteiros para Ponteiros

Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

*tipo_da_variável **nome_da_variável;*

Algumas considerações: ****nome_da_variável** é o conteúdo final da variável apontada; ***nome_da_variável** é o conteúdo do ponteiro intermediário.

No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros e assim por diante. Para fazer isto basta aumentar o número de asteriscos na declaracão. A lógica é a mesma.

Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no exemplo abaixo:

```
#include <stdio.h>

int main()
{
    float fpi = 3.1415, *pf, **ppf;
    pf = &fpi;                  /* pf armazena o endereco de fpi */
    ppf = &pf;                  /* ppf armazena o endereco de pf */
    printf("%f", **ppf);       /* Imprime o valor de fpi */
    printf("%f", *pf);          /* Tambem imprime o valor de fpi */
    return(0);
}
```

8. FUNÇÕES

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.

Uma função no C tem a seguinte forma geral:

tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
corpo_da_função
}

O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo **int**, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:

tipo nome1, tipo nome2, ... , tipo nomeN

Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no tipo-de-retorno).

O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

8.1 O Comando return

O comando **return** tem a seguinte forma geral:

return valor_de_retorno; ou *return;*

Digamos que uma função está sendo executada. Quando se chega a uma declaração **return** a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.

Uma função pode ter mais de uma declaração **return**. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração **return**. Abaixo estão dois exemplos de uso do **return**:

```
#include <stdio.h>
int Square (int a)
{
     return (a*a);
}
int main ()
{
     int num;
     printf ("Entre com um numero: ");
     scanf ("%d",&num);
     num=Square(num);
     printf ("\n\nO seu quadrado vale: %d\n",num);
     return 0;
}

#include <stdio.h>
int EPar (int a)
{
     if (a%2)           /* Verifica se a e divisivel por dois */
          return 0;        /* Retorna 0 se nao for divisivel */
```

```
      else
           return 1;         /* Retorna 1 se for divisivel */
}
int main ()
{
      int num;
      printf ("Entre com numero: ");
      scanf ("%d",&num);
      if (EPar(num))
            printf ("\n\nO numero e par.\n");
      else
            printf ("\n\nO numero e impar.\n");
      return 0;
}
```

É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemos fazer:

```
func(a,b)=x;    /* Errado! */
```

No segundo exemplo vemos o uso de mais de um **return** em uma função.

Fato importante: se uma função retorna um valor você *não precisa aproveitar* este valor. Se você não fizer nada com o valor de retorno de uma função ele será descartado. Por exemplo, a função **printf()** retorna um inteiro que nós nunca usamos para nada. Ele é descartado.

8.2 Protótipos de Funções

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função **main()**. Isto é, as funções estão fisicamente antes da função **main()**. Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto as funções foram colocadas antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos.

Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?

A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:

```
#include <stdio.h>
float Square (float a);

int main ()
{
```

```
      float num;
      printf ("Entre com um numero: ");
      scanf ("%f",&num);
      num=Square(num);
      printf ("\n\nO seu quadrado vale: %f\n",num);
      return 0;
}

float Square (float a)
{
      return (a*a);
}
```

Observe que a função **Square()** está colocada depois de **main()**, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.

Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

8.3 O Tipo void

Agora vamos ver o único tipo da linguagem C que não detalhamos ainda: o **void**. Em inglês, **void** quer dizer vazio e é isto mesmo que o **void** é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros! Podemos agora escrever o protótipo de uma função que não retorna nada:

void nome_da_função (declaração_de_parâmetros);

Numa função, como a acima, não temos valor de retorno na declaração **return**. Aliás, neste caso, o comando **return** não é necessário na função.

Podemos, também, fazer funções que não têm parâmetros:

tipo_de_retorno nome_da_função (void);

ou, ainda, que não tem parâmetros e não retornam nada:

void nome_da_função (void);

Um exemplo de funções que usam o tipo **void**:

```
#include <stdio.h>
void Mensagem (void);
int main ()
{
      Mensagem();
      printf ("\tDiga de novo:\n");
      Mensagem();
      return 0;
}
void Mensagem (void)
{
      printf ("Ola! Eu estou vivo.\n");
}
```

Se quisermos que a função retorne algo, devemos usar a declaração **return**. Se não quisermos, basta declarar a função como tendo tipo-de-retorno **void**. Devemos lembrar agora

que a função **main()** *é* uma função e como tal devemos tratá-la. O compilador acha que a função **main()** deve retornar um inteiro. Isto pode ser interessante se quisermos que o sistema operacional receba um valor de retorno da função **main()**. Se assim o quisermos, devemos nos lembrar da seguinte convenção: se o programa retornar zero, significa que ele terminou normalmente, e, se o programa retornar um valor diferente de zero, significa que o programa teve um término anormal. Se não estivermos interessados neste tipo de coisa, basta declarar a função main como retornando **void**.

8.4 Arquivos-Cabeçalhos

Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em **.h**. A extensão **.h** vem de **header** (cabeçalho em inglês). Já vimos exemplos como **stdio.h**, **conio.h**, **string.h**. Estes arquivos, na verdade, não possuem os códigos completos das funções. Eles só contêm *protótipos* de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem". Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.

Se você criar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-cabeçalhos e incluí-los também.

Suponha que a função 'int EPar(int a)', do segundo exemplo da página 54 seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado por exemplo de 'funcao.h' teremos a seguinte declaração:

int EPar(int a);

O código da função será escrito num arquivo a parte. Vamos chamá-lo de 'funcao.c'. Neste arquivo teremos a definição da função:

```
int EPar (int a)
{
      if (a%2)                /* Verifica se a e divisivel por dois */
         return 0;
      else
         return 1;
}
```

Por fim, no arquivo do programa principal teremos o programa principal. Vamos chamar este arquivo aqui de 'princip.c'.

```
#include <stdio.h>
#include "funcao.h"
void main ()
{
      int num;
      printf ("Entre com numero: ");
      scanf ("%d",&num);
      if (EPar(num))
            printf ("\n\nO numero e par.\n");
      else
            printf ("\n\nO numero e impar.\n");
}
```

8.5 Passagem de parâmetros por valor e passagem por referência

Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros. Veja o exemplo a seguir:

```
#include <stdio.h>
float sqr (float num);
void main ()
{
      float num,sq;
      printf ("Entre com um numero: ");
      scanf ("%f",&num);
      sq=sqr(num);
      printf ("\n\nO numero original e: %f\n",num);
      printf ("O seu quadrado vale: %f\n",sq);
}

float sqr (float num)
{
      num=num*num;
      return num;
}
```

No exemplo acima o parâmetro formal **num** da função **sqr()** sofre alterações dentro da função, mas a variável **num** da função **main()** permanece inalterada: é uma chamada por valor.

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recurso que permite ao programador fazer chamadas por referência. Há entretanto, no C, um recurso de programação que podemos usar para simular uma chamada por referência.

Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um **&** na frente das variáveis que estivermos passando para a função. Veja um exemplo:

```
#include <stdio.h>
void Swap (int *a,int *b);
```

```
void main (void)
{
      int num1,num2;
      num1=100;
      num2=200;
      Swap (&num1,&num2);
      printf ("\n\nEles agora valem %d  %d\n",num1,num2);
}
void Swap (int *a,int *b)
{
      int temp;
      temp=*a;
      *a=*b;
      *b=temp;
}
```

Não é muito difícil. O que está acontecendo é que passamos para a função Swap o endereço das variáveis num1 e num2. Estes endereços são copiados nos ponteiros a e b. Através do operador * estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Mas, quem é este conteúdo? Nada mais que os valores armazenados em num1 e num2, que, portanto, estão sendo modificados!

Espere um momento... será que nós já não vimos esta estória de chamar uma função com as variáveis precedidas de **&**? Já! É assim que nós chamamos a função **scanf()**. Mas porquê? Vamos pensar um pouco. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e portanto precisa alterar seus valores. Por isto passamos para a função o endereço da variável a ser modificada!

8.6 Vetores como Argumentos de Funções

Quando vamos passar um vetor como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Seja o vetor:

```
int matrx [50];
```

e que queiramos passá-la como argumento de uma função **func()**. Podemos declarar **func()** das três maneiras seguintes:

```
void func (int matrx[50]);
void func (int matrx[]);
void func (int *matrx);
```

Nos três casos, teremos dentro de **func()** um **int*** chamado **matrx**. Ao passarmos um vetor para uma função, na realidade estamos passando um ponteiro. Neste ponteiro é armazenado o endereço do primeiro elemento do vetor. Isto significa que não é feita uma cópia, elemento a elemento do vetor. Isto faz com que possamos alterar o valor dos elementos do vetor dentro da função.

Um exemplo disto já foi visto quando implementamos a função **StrCpy()**.

8.7 Os Argumentos argc e argv

A função **main()** pode ter parâmetros formais. Mas o programador não pode escolher quais serão eles. A declaração mais completa que se pode ter para a função **main()** é:

```
int main (int argc,char *argv[]);
```

Os parâmetros **argc** e **argv** dão ao programador acesso à linha de comando com a qual o programa foi chamado.

O **argc** (argument count) é um inteiro e possui o número de argumentos com os quais a função **main()** foi chamada na linha de comando. Ele é, no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.

O **argv** (argument values) é um ponteiro para uma matriz de strings. Cada string desta matriz é um dos parâmetros da linha de comando. O **argv[0]** sempre aponta para o nome do programa (que, como já foi dito, é considerado o primeiro argumento). É para saber quantos elementos temos em **argv** que temos **argc**.

Exemplo: Escreva um programa que faça uso dos parâamentros *argv* e *argc*. O programa deverá receber da linha de comando o dia, mês e ano correntes, e imprimir a data em formato apropriado. Veja o exemplo, supondo que o executável se chame data:
data 19 04 99
O programa deverá imprimir:
19 de abril de 1999

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{

      int mes;
      char *nomemes [] = {"Janeiro", "Fevereiro", "Março", "Abril", "Maio",
            "Junho", "Julho", "Agosto", "Setembro", "Outubro", "Novembro", "Dezembro"};

      if(argc == 4)
       /* Testa se o numero de parametros fornecidos esta' correto
          o primeiro parametro e' o nome do programa, o segundo o dia
          o terceiro o mes e o quarto os dois ultimos algarismos do ano */
      {
           mes = atoi(argv[2]);
            /* argv contem strings. A string referente ao mes deve ser
                 transformada em um numero inteiro. A funcao atoi esta
                 sendo usada para isto: recebe a string e transforma no
                 inteiro equivalente */
           if (mes<1 || mes>12)  /* Testa se o mes e' valido */
              printf("Erro!\nUso: data dia mes ano, todos inteiros");
           else
               printf("\n%s de %s de 19%s", argv[1], nomemes[mes-1], argv[3]);
}
else printf("Erro!\nUso: data dia mes ano, todos inteiros");
```

9. DIRETIVAS DE COMPILAÇÃO

O pré-processador C é um programa que examina o programa fonte escrito em C e executa certas modificações nele, baseado nas *Diretivas de Compilação*. As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, que é executado pelo compilador antes da execução do processo de compilação propriamente dito.

Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado. Todas as diretivas de compilação são iniciadas pelo caracter #. As diretivas podem ser colocadas em qualquer parte do programa. Já vimos, e usamos muito, a diretiva **#include**. Sabemos que ela não *gera* código mas diz ao compilador que ele deve incluir um arquivo externo na hora da compilação. As diretivas do C são identificadas por começarem por **#**. As diretivas que estudaremos são definidas pelo padrão ANSI:

#if	*#ifdef*	*#ifndef*
#else	*#elif*	*#endif*
#include	*#define*	*#undef*

9.1 A Diretiva include

A diretiva **#include** já foi usada durante o nosso curso diversas vezes. Ela diz ao compilador para incluir, na hora da compilação, um arquivo especificado. Sua forma geral é:

#include "nome_do_arquivo" ou

#include <nome_do_arquivo>

A diferença entre se usar **" "** e **< >** é somente a ordem de procura nos diretórios pelo arquivo especificado. Se você quiser informar o nome do arquivo com o caminho completo, ou se o arquivo estiver no diretório de trabalho, use **" "**. Se o arquivo estiver nos caminhos de procura pré-especificados do compilador, isto é, se ele for um arquivo do próprio sistema (como é o caso de arquivos como **stdio.h**, **string.h,** etc...) use **< >**.

Observe que não há ponto e vírgula após a diretiva de compilação. Esta é uma característica importante de todas as diretivas de compilação e não somente da diretiva #include

9.2 As Diretivas define e undef

A diretiva **#define** tem a seguinte forma geral:

#define nome_da_macro sequência_de_caracteres

Quando você usa esta diretiva, você está dizendo ao compilador para que, toda vez que ele encontrar o nome_da_macro no programa a ser compilado, ele deve substituí-lo pela sequência_de_caracteres fornecida. Isto é muito útil para deixar o programa mais geral. Veja um exemplo:

```
#include <stdio.h>
#define PI 3.1416
#define VERSAO "2.02"
int main ()
{
     printf ("Programa versao %s",VERSAO);
     printf ("O numero pi vale: %f",PI);
     return 0;
}
```

Se quisermos mudar o nosso valor de **PI**, ou da **VERSAO**, no programa acima, basta mexer no início do programa. Isto torna o programa mais flexível. Há quem diga que, em um programa, nunca se deve usar constantes como 10, 3.1416, etc., pois estes são números que ninguém sabe o que significam (muitas pessoas os chamam de "números mágicos"). Ao invés disto, deve-se usar apenas **#define**s. É uma convenção de programação (que deve ser seguida, pois torna o programa mais legível) na linguagem C que as macros declaradas em **#define**s devem ser todas em maiúsculas.

Um outro uso da diretiva #define é o de simplesmente definir uma macro. Neste caso usa-se a seguinte forma geral:

#define nome_da_macro

Neste caso o objetivo não é usar a macro no programa (pois ela seria substituída por nada), mas, sim, definir uma macro para ser usada como uma espécie de flag. Isto quer dizer que estamos definindo um valor como sendo "verdadeiro" para depois podermos testá-lo.

Também é possível definir macros com argumentos. Veja o exemplo a seguir:

```
#define max(A,B) ((A>B) ? (A):(B))
#define min(A,B) ((A<B) ? (A):(B))
...
x = max(i,j);
y = min(t,r);
```

Embora pareça uma chamada de função, o uso de max (ou min) simplesmente substitui, em tempo de compilação, o código especificado. Cada ocorrência de um parâmetro formal (A ou B, na definição) será substituído pelo argumento real correspondente. Assim, a linha de código:

```
x = max(i,j);
```

será substituída pela linha:

```
x = ((i)>(j) ? (i):(j));
```

A linha de código:

```
x = max(p+q,r+s);
```

será substituída pela linha:

```
x = ((p+q)>(r+s) ? (p+q):(r+s));
```

Isto pode ser muito útil. Verifique que as macros max e min não possuem especificação de tipo. Logo, elas trabalham corretamente para qualquer tipo de dado, enquanto os argumentos passados forem coerentes. Mas isto pode trazer também algumas armadilhas. Veja que a linha

```
x = max(p++,r++);
```

será substituída pelo código

```
x = ((p++)>(r++) ? (p++):(r++));
```

e em consequência, incrementará o maior valor duas vezes.

Outra armadilha em macros está relacionada com o uso de parênteses. Seja a macro:

```
#define SQR(X) X*X
```

Imagine que você utilize esta macro na expressão abaixo:

```
y = SQR(A+B);
```

Ao fazer isto, a substituição que será efetuada não estará correta. A expressão gerada será:

```
y = A+B*A+B;
```

que obviamente é diferente de (A+B)*(A+B) !

A solução para este problema é incluir parênteses na definição da macro:

```
#define SQR(X)(X)*(X)
```

Quando você utiliza a diretiva #define nunca deve haver espaços em branco no identificador. Por exemplo, a macro:

```
#define PRINT (i) printf(" %d \n", i)
```

não funcionará corretamente porque existe um espaço em branco entre PRINT e (i). Ao se tirar o espaço, a macro funcionará corretamente e poderá ser utilizada para imprimir o número inteiro i, saltando em seguida para a próxima linha.

A diretiva **#undef** tem a seguinte forma geral:

#undef nome_da_macro

Ela faz com que a macro que a segue seja apagada da tabela interna que guarda as macros.O compilador passa a partir deste ponto a não conhecer mais esta macro.

9.3 As Diretivas ifdef e endif

Nesta seção, e até mais a frente, veremos as diretivas de compilação condicional. Elas são muito parecidas com os comandos de execução condicional do C. As duas primeiras diretivas que veremos são as **#ifdef** e **#endif**. Suas formas gerais são:

#ifdef nome_da_macro
sequência_de_declarações
#endif

A sequência de declarações será compilada apenas se o nome da macro estiver definido. A diretiva de compilação **#endif** é util para definir o fim de uma sequência de declarações para todas as diretivas de compilação condicional. As linhas

```
#define PORT_0 0x378
...
/* Linhas de codigo qualquer... */
...
#ifdef PORT_0
  #define PORTA PORT_0
  #include "../sys/port.h"
#endif
```

demonstram como estas diretivas podem ser utilizadas. Caso *PORT_0* tenha sido previamente definido, a macro *PORTA* é definida e o header file *port.h* é incluído.

9.4 A Diretiva ifndef

A diretiva **#ifndef** funciona ao contrário da diretiva **#ifdef**. Sua forma geral é:

#ifndef nome_da_macro
    sequência_de_declarações
#endif

A sequência de declarações será compilada se o nome da macro não tiver sido definido

9.5 A Diretiva if

A diretiva **#if** tem a seguinte forma geral:

#if expressão_constante
    sequência_de_declarações
#endif

A sequência de declarações será compilada se a expressão-constante for verdadeira. É muito importande ressaltar que a expressão fornecida deve ser constante, ou seja, não deve ter nenhuma variável.

9.6 A Diretiva else

A diretiva **#else** tem a seguinte forma geral:

#if expressão_constante
sequência_de_declarações
#else
sequência_de_declarações
#endif

Ela funciona como seu correspondente, o comando **else**.

Imagine que você esteja trabalhando em um sistema, e deseje que todo o código possa ser compilado em duas diferentes plataformas (i.e. Unix e Dos). Para obter isto, você "encapsula" toda a parte de entrada e saída em arquivos separados, que serão carregados de acordo com o header file carregado. Isto pode ser facilmente implementado da seguinte forma:

```
#define SISTEMA DOS
...
/*linhas de codigo..*/
...
#if SISTEMA == DOS
  #define CABECALHO "dos_io.h"
#else
  #define CABECALHO "unix_io.h"
#endif
#include CABECALHO
```

9.7 A Diretiva elif

A diretiva **#elif** serve para implementar a estrutura **if-else-if**. Sua forma geral é:

#if expressão_constante_1
sequência_de_declarações_1
#elif expressão_constante_2
sequência_de_declarações_2
#elif expressão_constante_3
sequência_de_declarações_3
.
.
.
#elif expressão_constante_n
sequência_de_declarações_n
#endif

O funcionamento desta estrutura é idêntico ao funcionamento apresentado anteriormente.

10. ENTRADAS E SAÍDAS PADRONIZADAS

O sistema de entrada e saída da linguagem C está estruturado na forma de uma biblioteca de funções. Já vimos algumas destas funções, e agora elas serão reestudadas. Novas funções também serão apresentadas.

Não é objetivo deste curso explicar, em detalhes, todas as possíveis funções da biblioteca de entrada e saída do C. A sintaxe completa destas funções pode ser encontrada no manual do seu compilador. Alguns sistemas trazem uma descrição das funções na ajuda do compilador, que pode ser acessada "on line". Isto pode ser feito, por exemplo, no Rhide.

Um ponto importante é que agora, quando apresentarmos uma função, vamos, em primeiro lugar, apresentar o seu protótipo. Você já deve ser capaz de interpretar as informações que um protótipo nos passa. Se não, deve voltar a estudar a aula sobre funções.

Outro aspecto importante, quando se discute a entrada e saída na linguagem C é o conceito de *fluxo*. Seja qual for o dispositivo de entrada e saída (discos, terminais, teclados, acionadores de fitas) que se estiver trabalhando, o C vai enxergá-lo como um fluxo, que nada mais é que um dispositivo lógico de entrada ou saída. Todos os fluxos são similares em seu funcionamento e independentes do dispositivo ao qual estão associados. Assim, as mesmas funções que descrevem o acesso aos discos podem ser utilizadas para se acessar um terminal de vídeo. Todas as operações de entrada e saída são realizadas por meio de fluxos.

Na linguagem C, um arquivo é entendido como um conceito que pode ser aplicado a arquivos em disco, terminais, modens, etc ... Um fluxo é associado a um arquivo através da realização de uma operação de abertura. Uma vez aberto, informações podem ser trocadas entre o arquivo e o programa. Um arquivo é dissociado de um fluxo através de uma operação de fechamento de arquivo.

10.1 Lendo e Escrevendo Caracteres

Uma das funções mais básicas de um sistema é a entrada e saída de informações em dispositivos. Estes podem ser um monitor, uma impressora ou um arquivo em disco. Vamos ver os principais comandos que o C nos fornece para isto.

10.1.1 getche e getch

As funções getch() e getche() não são definidas pelo padrão ANSI. Porém, elas geralmente são incluídas em compiladores baseados no DOS, e se encontram no header file *conio.h*. Vale a pena repetir: são funções comuns apenas para compiladores baseados em DOS e, se você estiver no UNIX normalmente não terá estas funções disponíveis.

Protótipos:

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

getch() espera que o usuário digite uma tecla e retorna este caractere. Você pode estar estranhando o fato de **getch()** retornar um inteiro, mas não há problema pois este inteiro é tal que quando igualado a um **char** a conversão é feita corretamente. A função **getche()** funciona exatamente como **getch()**. A diferença é que **getche()** gera um "echo" na tela antes de retornar a tecla.

Se a tecla pressionada for um caractere especial estas funções retornam zero. Neste caso você deve usar as funções novamente para pegar o código da tecla extendida pressionada.

A função equivalente a **getche()** no mundo ANSI é o **getchar()**. O problema com getchar é que o caracter lido é colocado em uma área intermediária até que o usuário digite um <ENTER>, o que pode ser extremamente inconveniente em ambientes interativos.

10.1.2 putchar

Protótipo:

int putchar (int c);

putchar() coloca o caractere c na tela. Este caractere é colocado na posição atual do cursor. Mais uma vez os tipos são inteiros, mas você não precisa se preocupar com este fato. O header file é *stdio.h*.

10.2 Lendo e Escrevendo Strings

10.2.1 gets

Protótipo:

char *gets (char *s);

Pede ao usuário que entre uma string, que será armazenada na string **s**. O ponteiro que a função retorna é o próprio **s**.

gets não é uma função segura. Por quê? Simplesmente porque com **gets** pode ocorrer um estouro da quantidade de posições que foi especificada na string. Veja o exemplo abaixo:

```
#include <stdio.h>
int main()
{
    char buffer[10];
    printf("Entre com o seu nome");
    gets(buffer);
    printf("O nome é: %s", buffer);
    return 0;
}
```

Se o usuário digitar como entrada:

Renato Cardoso Mesquita

ou seja, digitar um total de 23 caracteres: 24 posições (incluindo o '\0') serão utilizadas para armazenar a string. Como a string buffer[] só tem 10 caracteres, os 14 caracteres adicionais serão colocados na área de memória subsequente à ocupada por ela, escrevendo uma região de memória que não está reservada à string. Este efeito é conhecido como "estouro de buffer" e pode causar problemas imprevisíveis. Uma forma de se evitar este problema é usar a função fgets, conforme veremos posteriormente.

10.2.2 puts

Protótipo:

int puts (char *s);

puts() coloca a string **s** na tela.

10.3 Entrada e Saída Formatada

As funções que resumem todas as funções de entrada e saída formatada no C são as funções **printf()** e **scanf()**. Um domínio destas funções é fundamental ao programador.

10.3.1 printf

Protótipo:

int printf (char *str,...);

As reticências no protótipo da função indicam que esta função tem um número de argumentos variável. Este número está diretamente relacionado com a string de controle **str**, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato. Como já vimos, os últimos determinam uma exibição de variáveis na saída. Os comandos de formato são precedidos de **%**. A cada comando de formato deve corresponder um argumento na função **printf()**. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.

Abaixo apresentamos a tabela de códigos de formato:

Código	**Formato**
%c	Um caracter (char)
%d	Um número inteiro decimal (int)
%i	O mesmo que %d
%e	Número em notação científica com o "e"minúsculo
%E	Número em notação científica com o "e"maiúsculo
%f	Ponto flutuante decimal
%g	Escolhe automaticamente o melhor entre %f e %e
%G	Escolhe automaticamente o melhor entre %f e
%o	Número octal
%s	String
%u	Decimal "unsigned" (sem sinal)
%x	Hexadecimal com letras minúsculas
%X	Hexadecimal com letras maiúsculas
%%	Imprime um %
%p	Ponteiro

Vamos ver alguns exemplos:

Código	**Imprime**
printf ("Um %%%c %s",'c',"char");	Um %c char
printf ("%X %f %e",107,49.67,49.67);	6B 49.67 4.967e1
printf ("%d %o",10,10);	10 12

É possível também indicar o tamanho do campo, justificação e o número de casas decimais. Para isto usa-se códigos colocados entre o **%** e a letra que indica o tipo de formato.

Um inteiro indica o tamanho mínimo, em caracteres, que deve ser reservado para a saída. Se colocarmos então **%5d** estamos indicando que o campo terá cinco caracteres de comprimento *no mínimo*. Se o inteiro precisar de mais de cinco caracteres para ser exibido então o campo terá o comprimento necessário para exibi-lo. Se o comprimento do inteiro for menor que cinco então o campo terá cinco de comprimento e será preenchido com espaços em branco. Se se quiser um preenchimento com zeros pode-se colocar um zero antes do número. Temos então que **%05d** reservará cinco casas para o número e se este for menor então se fará o preenchimento com zeros.

O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se um sinal - antes do número de casas. Então **%-5d** será o nosso inteiro com o número mínimo de cinco casas, só que justificado a esquerda.

Pode-se indicar o número de casas decimais de um número de ponto flutuante. Por exemplo, a notação **%10.4f** indica um ponto flutuante de comprimento total dez e com 4 casas decimais. Entretanto, esta mesma notação, quando aplicada a tipos como inteiros e strings indica o número mínimo e máximo de casas. Então **%5.8d** é um inteiro com comprimento mínimo de cinco e máximo de oito.

Vamos ver alguns exemplos:

Código	**Imprime**
printf ("%-5.2f",456.671);	| 456.67|
printf ("%5.2f",2.671);	| 2.67|
printf ("%-10s","Ola");	|Ola |

Nos exemplos o "pipe" (**|**) indica o início e o fim do campo mas não são escritos na tela.

10.3.2 scanf

Protótipo:

int scanf (char *str,...);

A string de controle str determina, assim como com a função **printf()**, quantos parâmetros a função vai necessitar. Devemos sempre nos lembrar que a função **scanf()** deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador **&**. Os especificadores de formato de entrada são muito parecidos com os de **printf()**. Os caracteres de conversão *d, i, u* e *x* podem ser precedidos por *h* para indicarem que um apontador para *short* ao invés de *int* aparece na lista de argumento, ou pela letra *l* (letra ele) para indicar que que um apontador para *long* aparece na lista de argumento. Semelhantemente, os caracteres de conversão *e, f* e *g* podem ser precedidos por *l* para indicarem que um apontador para *double* ao invés de *float* está na lista de argumento. Exemplos:

Código	**Formato**
%c	Um único caracter (char)
%d	Um número decimal (int)
%i	Um número inteiro
%hi	Um short int
%li	Um long int
%e	Um ponto flutuante
%f	Um ponto flutuante
%lf	Um double
%h	Inteiro curto
%o	Número octal
%s	String
%x	Número hexadecimal
%p	Ponteiro

10.3.3 sprintf e sscanf

sprintf e sscanf são semelhantes a printf e scanf. Porém, ao invés de escreverem na saída padrão ou lerem da entrada padrão, escrevem ou leem em uma string. Os protótipos são:

int sprintf (char *destino, char *controle, ...);

int sscanf (char *destino, char *controle, ...);

Estas funções são muito utilizadas para fazer a conversão entre dados na forma numérica e sua representação na forma de strings. No programa abaixo, por exemplo, a variável i é "impressa" em string1. Além da representação de i como uma string, string1 também conterá "Valor de i=" .

```
#include <stdio.h>
int main()
{
    int i;
    char string1[20];
    printf( " Entre um valor inteiro: ");
    scanf("%d", &i);
    sprintf(string1,"Valor de i = %d", i);
    puts(string1);
    return 0;
}
```

Já no programa abaixo, foi utilizada a função sscanf para converter a informação armazenada em string1 em seu valor numérico:

```
#include <stdio.h>
int main()
{
    int i, j, k;
    char string1[]= "10 20 30";
    sscanf(string1, "%d %d %d", &i, &j, &k);
    printf("Valores lidos: %d, %d, %d", i, j, k);
    return 0;
}
```

11. ARQUIVOS

O sistema de entrada e saída do ANSI C é composto por uma série de funções, cujos protótipos estão reunidos em **stdio.h** . Todas estas funções trabalham com o conceito de "ponteiro de arquivo". Este não é um tipo propriamente dito, mas uma definição usando o comando typedef. Esta definição também está no arquivo **stdio.h**. Podemos declarar um ponteiro de arquivo da seguinte maneira:

FILE *p;

p será então um ponteiro para um arquivo. É usando este tipo de ponteiro que vamos poder manipular arquivos no C.

11.1 fopen

Esta é a função de abertura de arquivos. Seu protótipo é:

FILE *fopen (char *nome_do_arquivo,char *modo);

O nome_do_arquivo determina qual arquivo deverá ser aberto. Este nome deve ser válido no sistema operacional que estiver sendo utilizado. O modo de abertura diz à função **fopen()** que tipo de uso você vai fazer do arquivo. A tabela abaixo mostra os valores de modo válidos:

Modo	Significado
"r"	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
"w"	Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
"a"	Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append"), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"rb"	Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o arquivo é binário.
"wb"	Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário.
"ab"	Acrescenta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário.
"r+"	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
"w+"	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.
"a+"	Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"r+b"	Abre um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que o arquivo é binário.
"w+b"	Cria um arquivo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário.
"a+b"	Acrescenta dados ou cria uma arquivo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário

Poderíamos então, para abrir um arquivo binário para escrita, escrever:

```
FILE *fp;                  /* Declaração da estrutura
fp=fopen ("exemplo.bin","wb");  /* o arquivo se chama exemplo.bin e está localizado no diretório corrente */
```

```
if (!fp)
      printf ("Erro na abertura do arquivo.");
```

A condição **!fp** testa se o arquivo foi aberto com sucesso porque no caso de um erro a função **fopen()** retorna um ponteiro nullo (**NULL**).

Uma vez aberto um arquivo, vamos poder ler ou escrever nele utilizando as funções que serão apresentadas nas próximas páginas.

Toda vez que estamos trabalhando com arquivos, há uma espécie de posição atual no arquivo. Esta é a posição de onde será lido ou escrito o próximo caractere. Normalmente, num acesso sequencial a um arquivo, não temos que mexer nesta posição pois quando lemos um caractere a posição no arquivo é automaticamente atualizada. Num acesso randômico teremos que mexer nesta posição (ver fseek()).

11.2 fclose

Quando acabamos de usar um arquivo que abrimos, devemos fechá-lo. Para tanto usa-se a função **fclose()**:

```
int fclose (FILE *fp);
```

O ponteiro **fp** passado à função **fclose()** determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

Fechar um arquivo faz com que qualquer caracter que tenha permanecido no "buffer" associado ao fluxo de saída seja gravado. Mas, o que é este "buffer"? Quando você envia caracteres para serem gravados em um arquivo, estes caracteres são armazenados temporariamente em uma área de memória (o "buffer") em vez de serem escritos em disco imediatamente. Quando o "buffer" estiver cheio, seu conteúdo é escrito no disco de uma vez. A razão para se fazer isto tem a ver com a eficiência nas leituras e gravações de arquivos. Se, para cada caracter que fossemos gravar, tivéssemos que posicionar a cabeça de gravação em um ponto específico do disco, apenas para gravar aquele caracter, as gravações seriam muito lentas. Assim estas gravações só serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.

A função exit() fecha todos os arquivos que um programa tiver aberto.

11.3 Lendo e Escrevendo Caracteres em Arquivos

11.3.1 putc

A função putc é a primeira função de escrita de arquivo que veremos. Seu protótipo é:

```
int putc (int ch,FILE *fp);
```

Escreve um caractere no arquivo.

O programa a seguir lê uma string do teclado e escreve-a, caractere por caractere em um arquivo em disco (o arquivo arquivo.txt, que será aberto no diretório corrente).

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *fp;
    char string[100];
    int i;
    fp = fopen("arquivo.txt","w");    /* Arquivo ASCII, para escrita */
    if(!fp)
    {
       printf( "Erro na abertura do arquivo");
       exit(0);
```

```
    }
    printf("Entre com a string a ser gravada no arquivo:");
    gets(string);
   for(i=0; string[i]; i++)
      putc(string[i], fp); /* Grava a string, caractere a caractere */
   fclose(fp);
   return 0;
}
```

Depois de executar este programa, verifique o conteúdo do arquivo arquivo.txt (você pode usar qualquer editor de textos). Você verá que a string que você digitou está armazenada nele.

11.3.2getc

Retorna um caractere lido do arquivo. Protótipo:

int getc (FILE *fp);

11.3.3feof

EOF ("End of file") indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto podemos usar a função **feof()**. Ela retorna não-zero se o arquivo chegou ao EOF, caso contrário retorna zero. Seu protótipo é:

int feof (FILE *fp);

Outra forma de se verificar se o final do arquivo foi atingido é comparar o caractere lido por getc com EOF. O programa a seguir abre um arquivo já existente e o lê, caracter por caracter, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela:

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

int main()
{
    FILE *fp;
    char c;
    fp = fopen("arquivo.txt","r");   /* Arquivo ASCII, para leitura */
    if(!fp)
    {
       printf( "Erro na abertura do arquivo");
       exit(0);
    }
    while((c = getc(fp) ) != EOF)
    /* Enquanto não chegar ao final do arquivo */
          printf("%c", c);                /* imprime o caracter lido */
    fclose(fp);
    return 0;
}
```

A seguir é apresentado um programa onde várias operações com arquivos são realizadas, usando as funções vistas nesta página. Primeiro o arquivo é aberto para a escrita, e imprime-se algo nele. Em seguida, o arquivo é fechado e novamente aberto para a leitura. Verifique o exemplo.

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

void main()
{
        FILE *p;
        char c, str[30], frase[80] = "Este e um arquivo chamado: ";
        int i;
        /* Le um nome para o arquivo a ser aberto: */
        printf("\n\n Entre com um nome para o arquivo:\n");
        gets(str);

        if (!(p = fopen(str,"w")))  /* Caso ocorra algum erro na abertura
do arquivo..*/
        {                           * o programa aborta automaticamente */
          printf("Erro! Impossivel abrir o arquivo!\n");
          exit(1);
        }
        /* Se nao houve erro, imprime no arquivo e o fecha ...*/
        strcat(frase, str);
        for (i=0; frase[i]; i++)
         putc(frase[i],p);
        fclose(p);

        /* Abre novamente para  leitura  */
        p = fopen(str,"r");
        c = getc(p);       /* Le o primeiro caracter */
        while (!feof(p)) /* Enquanto não se chegar no final do arquivo */
        {
                printf("%c",c); /*   Imprime o caracter na tela */
                c = getc(p);    /* Le um novo caracter no arquivo */
        }
        fclose(p);              /* Fecha o arquivo */
}
```

11.4 Outros Comandos de Acesso a Arquivos

11.4.1fgets

Para se ler uma string num arquivo podemos usar **fgets()** cujo protótipo é:

char *fgets (char *str, int tamanho,FILE *fp);

A função recebe 3 argumentos: a string a ser lida, o limite máximo de caracteres a serem lidos e o ponteiro para FILE, que está associado ao arquivo de onde a string será lida. A função lê a string até que um caracter de nova linha seja lido ou *tamanho-1* caracteres tenham sido lidos. Se o caracter de nova linha ('\n') for lido, ele fará parte da string, o que não acontecia com gets. A string resultante sempre terminará com '\0' (por isto somente *tamanho-1* caracteres, no máximo, serão lidos).

A função fgets é semelhante à função **gets()** porém, além dela poder fazer a leitura a partir de um arquivo de dados e incluir o caracter de nova linha na string, ela ainda especifica o tamanho máximo da string de entrada. Como vimos, a função gets não tinha este controle, o

que poderia acarretar erros de "estouro de buffer". Portanto, levando em conta que o ponteiro fp pode ser substituído por stdin, como vimos acima, uma alternativa ao uso de gets é usar a seguinte construção:

fgets (str, tamanho, stdin);

onde str e' a string que se está lendo e tamanho deve ser igual ao tamanho alocado para a string subtraído de 1, por causa do '\0'.

11.4.2 fputs

Protótipo:

char *fputs (char *str,FILE *fp);

Escreve uma string num arquivo.

11.4.3 ferror e perror

Protótipo de ferror:

int ferror (FILE *fp);

A função retorna zero, se nenhum erro ocorreu e um número diferente de zero se algum erro ocorreu durante o acesso ao arquivo.

ferror() se torna muito útil quando queremos verificar se cada acesso a um arquivo teve sucesso, de modo que consigamos garantir a integridade dos nossos dados. Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado. Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não conseguimos ler, etc.

Uma função que pode ser usada em conjunto com **ferror()** é a função **perror()** (print error), cujo argumento é uma string que normalmente indica em que parte do programa o problema ocorreu.

No exemplo a seguir, fazemos uso de **ferror, perror e fputs**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *pf;
    char string[100];
    if((pf = fopen("arquivo.txt","w")) ==NULL)
    {
        printf("\nNao consigo abrir o arquivo ! ");
        exit(1);
    }
    do
    {
        printf("\nDigite uma string. Para terminar, digite <enter>: ");
        gets(string);
        fputs(string, pf);
        putc('\n', pf);
        if(ferror(pf))
        {
            perror("Erro na gravacao");
            fclose(pf);
            exit(1);
        }
    }
```

```
    while (strlen(string) > 0);
    fclose(pf);
}
```

11.4.4 fread

Podemos escrever e ler blocos de dados. Para tanto, temos as funções **fread()** e **fwrite()**. O protótipo de **fread()** é:

unsigned fread (void *buffer, int numero_de_bytes, int count, FILE *fp);

O *buffer* é a região de memória na qual serão armazenados os dados lidos. O *número de bytes* é o tamanho da unidade a ser lida. *count* indica quantas unidades devem ser lidas. Isto significa que o número total de bytes lidos é:

*numero_de_bytes*count*

A função retorna o número de unidades efetivamente lidas. Este número pode ser menor que *count* quando o fim do arquivo for encontrado ou ocorrer algum erro.

Quando o arquivo for aberto para dados binários, fread pode ler qualquer tipo de dados.

11.4.5 fwrite

A função **fwrite()** funciona como a sua companheira **fread()**, porém escrevendo no arquivo. Seu protótipo é:

unsigned fwrite(void *buffer,int numero_de_bytes,int count,FILE *fp);

A função retorna o número de itens escritos. Este valor será igual a count a menos que ocorra algum erro.

O exemplo abaixo ilustra o uso de **fwrite** e **fread** para gravar e posteriormente ler uma variável float em um arquivo binário.

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

int main()
{
    FILE *pf;
    float pi = 3.1415;
    float pilido;

    if((pf = fopen("arquivo.bin", "wb")) == NULL)
    /* Abre arquivo binário para escrita */
    {
        printf("Erro na abertura do arquivo");
        exit(1);
    }
    if(fwrite(&pi, sizeof(float), 1,pf) != 1)
    /* Escreve a variável pi */
       printf("Erro na escrita do arquivo");
    fclose(pf);
                                /* Fecha o arquivo */
    if((pf = fopen("arquivo.bin", "rb")) == NULL)
    /* Abre o arquivo novamente para leitura */
    {
       printf("Erro na abertura do arquivo");
```

```
        exit(1);
    }

    if(fread(&pilido, sizeof(float), 1,pf) != 1)
    /* Le em pilido o valor da variável armazenada anteriormente */
       printf("Erro na leitura do arquivo");

    printf("\nO valor de PI, lido do arquivo e': %f", pilido);
    fclose(pf);
    return(0);
}
```

Note-se o uso do operador sizeof, que retorna o tamanho em bytes da variável ou do tipo de dados.

11.4.6 fseek

Para se fazer procuras e acessos randômicos em arquivos usa-se a função **fseek()**. Esta move a posição corrente de leitura ou escrita no arquivo de um valor especificado, a partir de um ponto especificado. Seu protótipo é:

int fseek (FILE *fp,long numbytes,int origem);

O parâmetro *origem* determina a partir de onde os *numbytes* de movimentação serão contados. Os valores possíveis são definidos por macros em **stdio.h** e são:

Nome	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Ponto corrente no arquivo
SEEK_END	2	Fim do arquivo

Tendo-se definido a partir de onde irá se contar, numbytes determina quantos bytes de deslocamento serão dados na posição atual.

11.4.7 rewind

A função **rewind()** de protótipo

void rewind (FILE *fp);

retorna a posição corrente do arquivo para o início.

11.4.8 remove

Protótipo:

int remove (char *nome_do_arquivo);

Apaga um arquivo especificado.

O exercício da página anterior poderia ser reescrito usando-se, por exemplo, **fgets**() e **fputs**(), ou **fwrite**() e **fread**(). A seguir apresentamos uma segunda versão que se usa das funções **fgets**() e **fputs**(), e que acrescenta algumas inovações.

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

```
int main()
{
      FILE *p;
      char  str[30],  frase[]  =  "Este  e  um  arquivo  chamado:  ",
resposta[80];
      int i;

      /* Le um nome para o arquivo a ser aberto: */
      printf("\n\n Entre com um nome para o arquivo:\n");
      fgets(str,29,stdin);       /* Usa fgets como se fosse gets */

      for(i=0; str[i]; i++) if(str[i]=='\n')
          str[i]=0;
      /* Elimina o \n da string lida */

      if (!(p = fopen(str,"w")))  /
      {
            printf("Erro! Impossivel abrir o arquivo!\n");
            exit(1);
      }

      fputs(frase, p);
      fputs(str,p);
      fclose(p);

      p = fopen(str,"r");
      fgets(resposta, 79, p);
      printf("\n\n%s\n", resposta);
      fclose(p);
      remove(str);
      return(0);
}
```

11.5 Fluxos Padrão

Os fluxos padrão em arquivos permitem ao programador ler e escrever em arquivos da maneira padrão com a qual o já líamos e escrevíamos na tela.

11.5.1 fprintf

A função **fprintf()** funciona como a função **printf()**. A diferença é que a saída de **fprintf()** é um arquivo e não a tela do computador. Protótipo:

```
int fprintf (FILE *fp,char *str,...);
```

Como já poderíamos esperar, a única diferença do protótipo de **fprintf()** para o de **printf()** é a especificação do arquivo destino através do ponteiro de arquivo.

11.5.2 fscanf

A função **fscanf()** funciona como a função **scanf()**. A diferença é que **fscanf()** lê de um arquivo e não do teclado do computador. Protótipo:

```
int fscanf (FILE *fp,char *str,...);
```

Como já poderíamos esperar, a única diferença do protótipo de **fscanf()** para o de **scanf()** é a especificação do arquivo destino através do ponteiro de arquivo.

Talvez a forma mais simples de escrever o programa da página 79 e 80 se usando **fprintf** () e **fscanf**(). Fica assim:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        FILE *p;
        char str[80],c;

        /* Le um nome para o arquivo a ser aberto: */
        printf("\n\n Entre com um nome para o arquivo:\n");
        gets(str);

        if (!(p = fopen(str,"w")))
        {
                printf("Erro! Impossivel abrir o arquivo!\n");
                exit(1);
        }
        fprintf(p,"Este e um arquivo chamado:\n%s\n", str);
        fclose(p);

        p = fopen(str,"r");
        while (!feof(p))
        {
                fscanf(p,"%c",&c);
                printf("%c",c);
        }
        fclose(p);
        return(0);
}
```

12. TIPOS DE DADOS AVANÇADOS

12.1 Modificadores de Acesso

Estes modificadores, como o próprio nome indica, mudam a maneira com a qual a variável é acessada e modificada.

12.1.1 const

O modificador **const** faz com que a variável não possa ser modificada no programa. Como o nome já sugere é útil para se declarar constantes. Poderíamos ter, por exemplo:

const float PI=3.141;

Podemos ver pelo exemplo que as variáveis com o modificador **const** podem ser inicializadas. Mas **PI** não poderia ser alterado em qualquer outra parte do programa. Se o programador tentar modificar **PI** o compilador gerará um erro de compilação.

O uso mais importante de **const** não é declarar variáveis constantes no programa. Seu uso mais comum é evitar que um parâmetro de uma função seja alterado pela função. Isto é muito útil no caso de um ponteiro, pois o conteúdo de um ponteiro pode ser alterado por uma função. Para tanto, basta declarar o parâmetro como **const**. Veja o exemplo:

```
#include <stdio.h>
int sqr (const int *num);
main (void)
{
      int a=10;
      int b;
      b=sqr (&a);
}

int sqr (const int *num)
{
      return ((*num)*(*num));
}
```

No exemplo, **num** está protegido contra alterações. Isto quer dizer que, se tentássemos fazer

*num=10;

dentro da função **sqr()** o compilador daria uma mensagem de erro.

12.1.2 volatile

O modificador **volatile** diz ao compilador que a variável em questão pode ser alterada sem que este seja avisado. Isto evita "bugs" seríssimos. Digamos que, por exemplo, tenhamos uma variável que o BIOS do computador altera de minuto em minuto (um relógio por exemplo). Seria muito bom que declarássemos esta variável como sendo **volatile**.

12.2 Especificadores de Classe de Armazenamento

Estes modificadores de tipo atuam sobre a maneira com a qual o compilador vai armazenar a variável.

12.2.1 auto

O especificador de classe de armazenamento **auto** define variáveis automáticas, isto é, variáveis locais. Raramente usado pois todas as variáveis locais do C são **auto** por definição.

12.2.2 extern

O **extern** define variáveis que serão usadas em um arquivo apesar de terem sido declaradas em outro. Ao contrário dos programas até aqui vistos, podemos ter programas de vários milhares de linhas. Estes podem ser divididos em vários arquivos (módulos) que serão compilados separadamente. Digamos que para um programa grande tenhamos duas variáveis globais: um inteiro **count** e um **float sum**. Estas variáveis são declaradas normalmente em um dos módulos do programa. Por exemplo:

```
int count;
float sum;
main (void)
{
      ...
      return 0;
}
```

Num outro módulo do programa temos uma rotina que deve usar as variáveis globais acima. Digamos que a rotina que queremos se chama **RetornaCount()** e retorna o valor atual de **count**. O problema é que este módulo será compilado em separado e não tomará conhecimento dos outros módulos. O que fazer? Será que funcionaria se fizermos assim:

```
int count;              /* errado */
float sum;
int RetornaCount (void)
{
      return count;
}
```

Não. O módulo compilaria sem problema, mas, na hora que fizermos a linkagem (união dos módulos já compilados para gerar o executável) vamos nos deparar com uma mensagem de erro dizendo que as variáveis globais count e sum foram declaradas mais de uma vez. A maneira correta de se escrever o módulo com a função **RetornaCount()** é:

```
extern int count;       /* certo */
extern float sum;
int RetornaCount (void)
{
      return count;
}
```

Assim, o compilador irá saber que **count** e **sum** estão sendo usados no bloco mas que foram declarados em outro.

12.2.3 static

O funcionamento das variáveis declaradas como **static** depende se estas são globais ou locais.

Variáveis globais **static** funcionam como variáveis globais dentro de um módulo, ou seja, são variáveis globais que não são (e nem podem ser) conhecidas em outros modulos. Isto é util se quisermos isolar pedaços de um programa para evitar mudanças acidentais em variáveis globais.

Variáveis locais **static** são variáveis cujo valor é mantido de uma chamada da função para a outra. Veja o exemplo:

```
int count (void)
{
      static int num=0;
      num++;
      return num;
}
```

A função **count()** retorna o número de vezes que ela já foi chamada. Veja que a variável local **int** é inicializada. Esta inicialização só vale para a *primeira* vez que a função é chamada pois **num** deve manter o seu valor de uma chamada para a outra. O que a função faz é incrementar num a cada chamada e retornar o seu valor. A melhor maneira de se entender esta variável local **static** é implementando. Veja por si mesmo, executando seu próprio programa que use este conceito.

12.2.4 register

O computador tem a memória principal e os registradores da CPU. As variáveis (assim como o programa como um todo) são armazenados na memória. O modificador **register** diz ao compilador que a variável em questão deve ser, se possível, usada em um registrador da CPU.

Vamos agora ressaltar vários pontos importantes. Em primeiro lugar, porque usar o **register**? Variáveis nos registradores da CPU vão ser acessadas em um tempo muito menor pois os registradores são muito mais rápidos que a memória. Em segundo lugar, em que tipo de variável usar o **register**? O **register** *não* pode ser usado em variáveis globais. Isto implicaria que um registrador da CPU ficaria o tempo todo ocupado por conta de uma variável. Os tipos de dados onde é mais aconselhado o uso do **register** são os tipos **char** e **int**, mas pode-se usá-lo em qualquer tipo de dado. Em terceiro lugar, o **register** é um pedido que o programador faz ao compilador. Este não precisa ser atendido necessariamente.

Um exemplo do uso do register é dado:

```
main (void)
{
      register int count;
      for (count=0;count<10;count++)
      {
         ...
      }
      return 0;
}
```

O loop **for** acima será executado mais rapidamente do que seria se não usássemos o **register**. Este é o uso mais recomendável para o **register**: uma variável que será usada muitas vezes em seguida.

12.3 Alocação Dinâmica

A alocação dinâmica permite ao programador alocar memória para variáveis quando o programa está sendo executado. Assim, poderemos definir, por exemplo, um vetor ou uma matriz cujo tamanho descobriremos em tempo de execução. O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**:

No entanto, existem diversas outras funções que são amplamente utilizadas, mas dependentes do ambiente e compilador. Neste curso serão abordadas somente estas funções padronizadas.

12.3.1 malloc

A função **malloc()** serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

A funçao toma o número de bytes que queremos alocar (**num**), aloca na memória e retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **malloc()** retorna um ponteiro nulo. Veja um exemplo de alocação dinâmica com malloc():

```
#include <stdio.h>
#include <stdlib.h>   /* Para usar malloc() */

main (void)

{

     int *p;
     int a;
     int i;

... /* Determina o valor de a em algum lugar */

     p=(int *)malloc(a*sizeof(int));
     /* Aloca a números inteiros p pode agora ser tratado como um vetor
com a posicoes */

     if (!p)
        {
           printf ("** Erro: Memoria Insuficiente **");
           exit;
        }

     for (i=0; i<a ; i++)
     /* p pode ser tratado como um vetor com a posicoes */
          p[i] = i*i;

...

     return 0;
}
```

No exemplo acima, é alocada memória suficiente para se armazenar **a** números inteiros. O operador **sizeof()** retorna o número de bytes de um inteiro. Ele é util para se saber o tamanho de tipos. O ponteiro **void*** que **malloc()** retorna é convertido para um **int*** pelo cast e é atribuído a **p**. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de **p[0]** a **p[(a-1)]**.

12.3.2 calloc

A função **calloc()** também serve para alocar memória, mas possui um protótipo um pouco diferente:

void *calloc (unsigned int num, unsigned int size);

A funçao aloca uma quantidade de memória igual a **num * size**, isto é, aloca memória suficiente para um vetor de **num** objetos de tamanho **size**. Retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função **calloc()** retorna um ponteiro nulo. Veja um exemplo de alocação dinâmica com calloc():

```
#include <stdio.h>
#include <stdlib.h>   /* Para usar calloc() */

main (void)

{

      int *p;
      int a;
      int i;

... /* Determina o valor de a em algum lugar */

      p=(int *)calloc(a,sizeof(int));

      if (!p)
         {
            printf ("** Erro: Memoria Insuficiente **");
            exit;
         }

      for (i=0; i<a ; i++)
           p[i] = i*i;

...

      return 0;
}
```

No exemplo acima, é alocada memória suficiente para se colocar **a** números inteiros. O operador **sizeof()** retorna o número de bytes de um inteiro. Ele é util para se saber o tamanho de tipos. O ponteiro **void *** que **calloc()** retorna é convertido para um **int *** pelo cast e é atribuído a **p**. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação tiver sido bem

sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de **p[0]** a **p[(a-1)]**.

12.3.3 realloc

A função **realloc()** serve para realocar memória e tem o seguinte protótipo:

void *realloc (void *ptr, unsigned int num);

A funçao modifica o tamanho da memória previamente alocada apontada por ***ptr** para aquele especificado por **num**. O valor de **num** pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque **realloc()** pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida. Se **ptr** for nulo, aloca **size** bytes e devolve um ponteiro; se **size** é zero, a memória apontada por **ptr** é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

```
#include <stdio.h>
#include <stdlib.h>   /* Para usar malloc()  e realloc*/
main (void)

{

      int *p;
      int a;
      int i;

      ... /* Determina o valor de a em algum lugar */
      a = 30;

      p=(int *)malloc(a*sizeof(int));
      if (!p)
         {
            printf ("** Erro: Memoria Insuficiente **");
            exit;
         }

      for (i=0; i<a ; i++)
           p[i] = i*i;

      /* O tamanho de p deve ser modificado, por algum motivo ... */
      a = 100;
      p = realloc (p, a*sizeof(int));
      for (i=0; i<a ; i++)
          p[i] = a*i*(i-6);
      ...

      return 0;
}
```

12.3.4 free

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função **free()** cujo protótipo é:

```
void free (void *p);
```

Basta então passar para **free()** o ponteiro que aponta para o início da memória alocada. Mas você pode se perguntar: como é que o programa vai saber quantos bytes devem ser liberados? Ele sabe pois quando você alocou a memória, ele guardou o número de bytes alocados numa "tabela de alocação" interna. Vamos reescrever o exemplo usado para a função **malloc()** usando o **free()** também agora:

```
#include <stdio.h>
#include <stdlib.h>   /* Para usar malloc e free */

main (void)
{
      int *p;
      int a;
      ...
      p=(int *)malloc(a*sizeof(int));
      if (!p)
      {
          printf ("** Erro: Memoria Insuficiente **");
         exit;
      }
      ...
      free(p);
      ...
      return 0;
}
```

12.4 Alocação Dinâmica de Vetores e Matrizes

12.4.1 Alocação Dinâmica de Vetores

A alocação dinâmica de vetores utiliza os conceitos aprendidos na aula sobre ponteiros e as funções de alocação dinâmica apresentados. Um exemplo de implementação para vetor real é fornecido a seguir:

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

float *Alocar_vetor_real (int n)
{
      float *v;         /* ponteiro para o vetor */
      if (n < 1)
      {
            printf ("** Erro: Parametro invalido **\n");
            return (NULL);
      }
```

```
  /* aloca o vetor */

     v = (float *) calloc (n, sizeof(float));
     if (v == NULL)
     {
         printf ("** Erro: Memoria Insuficiente **");
         return (NULL);
     }
     return (v);    /* retorna o ponteiro para o vetor */
}

float *Liberar_vetor_real (float *v)
{
     if (v == NULL) return (NULL);
        free(v);        /* libera o vetor */
  return (NULL);  /* retorna o ponteiro */
}

void main (void)
{
     float *p;
     int a;
     ...    /* outros comandos, inclusive a inicializacao de a */
     p = Alocar_vetor_real (a);
     ...    /* outros comandos, utilizando p[] normalmente */
     p = Liberar_vetor_real (p);
}
```

12.4.2Alocação Dinâmica de Matrizes

A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, ou seja é um ponteiro para ponteiro, o que é denominado *indireção múltipla*. A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. Um exemplo de implementação para matriz real bidimensional é fornecido a seguir. A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz. Em cada linha existe um vetor alocado dinamicamente, como descrito anteriormente (compondo o segundo índice da matriz).

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

float **Alocar_matriz_real (int m, int n)
{
     float **v;  /* ponteiro para a matriz */
     int   i;    /* variavel auxiliar      */
     if (m < 1 || n < 1)  /* verifica parametros recebidos */
     {
         printf ("** Erro: Parametro invalido **\n");
         return (NULL);
     }
```

```
  /* aloca as linhas da matriz */
      v = (float **) calloc (m, sizeof(float *));
      / Um vetor de m ponteiros para float */

      if (v == NULL)
      {
          printf ("** Erro: Memoria Insuficiente **");
          return (NULL);
      }

      /* aloca as colunas da matriz */
      for ( i = 0; i < m; i++ )
      {
            v[i] = (float*) calloc (n, sizeof(float));
            /* m vetores de n floats */
            if (v[i] == NULL) {
                  printf ("** Erro: Memoria Insuficiente **");
                  return (NULL);
          }
       }
       return (v); /* retorna o ponteiro para a matriz */
}

float **Liberar_matriz_real (int m, int n, float **v)
{
      int  i;  /* variavel auxiliar */
      if (v == NULL)
         return (NULL);

      if (m < 1 || n < 1)  /* verifica parametros recebidos */
      {
          printf ("** Erro: Parametro invalido **\n");
          return (v);
      }

      for (i=0; i<m; i++)
           free (v[i]); /* libera as linhas da matriz */

     free (v);      /* libera a matriz (vetor de ponteiros) */
     return (NULL); /* retorna um ponteiro nulo */
}

void main (void)
{
      float **mat;  /* matriz a ser alocada */
      int   l, c;   /* numero de linhas e colunas da matriz */
      int i, j;
      ...       /* outros comandos, inclusive inicializacao para l e c */

      mat = Alocar_matriz_real (l, c);
```

```
      for (i = 0; i < l; i++)
          for ( j = 0; j < c; j++)
              mat[i][j] = i+j;

      ...           /* outros comandos utilizando mat[][] normalmente */
      mat = Liberar_matriz_real (l, c, mat);
      ...
}
```

13. TIPOS DE DADOS DEFINIDOS PELO USUÁRIO

13.1 Estruturas

Uma estrutura agrupa várias variáveis numa só. Funciona como uma ficha pessoal que tenha nome, telefone e endereço. A ficha seria uma estrutura. A estrutura, então, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

13.1.1 Criação

Para se criar uma estrutura usa-se o comando **struct**. Sua forma geral é:

struct nome_do_tipo_da_estrutura
{
tipo_1 nome_1;
tipo_2 nome_2;
...
tipo_n nome_n;
} variáveis_estrutura;

O nome_do_tipo_da_estrutura é o nome para a estrutura. As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo nome_do_tipo_da_estrutura. Um primeiro exemplo:

```
struct est{
    int i;
    float f;
} a, b;
```

Neste caso, est é uma estrutura com dois campos, i e f. Foram também declaradas duas variáveis, a e b que são do tipo da estrutura, isto é, a possui os campos i e f, o mesmo acontecendo com b.

Vamos criar uma estrutura de endereço:

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

Vamos agora criar uma estrutura chamada ficha_pessoal com os dados pessoais de uma pessoa:

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

Vemos, pelos exemplos acima, que uma estrutura pode fazer parte de outra (a struct tipo_endereco é usada pela struct ficha_pessoal).

13.1.2 Uso

Vamos agora utilizar as estruturas declaradas na seção anterior para escrever um programa que preencha uma ficha.

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

main (void)
{
     struct ficha_pessoal ficha;
     strcpy (ficha.nome,"Luiz Osvaldo Silva");
     ficha.telefone=4921234;
     strcpy (ficha.endereco.rua,"Rua das Flores");
     ficha.endereco.numero=10;
     strcpy (ficha.endereco.bairro,"Cidade Velha");
     strcpy (ficha.endereco.cidade,"Belo Horizonte");
     strcpy (ficha.endereco.sigla_estado,"MG");
     ficha.endereco.CEP=31340230;
     return 0;
}
```

O programa declara uma variável ficha do tipo **ficha_pessoal** e preenche os seus dados. O exemplo mostra como podemos acessar um elemento de uma estrutura: basta usar o ponto (**.**). Assim, para acessar o campo telefone de ficha, escrevemos:

ficha.telefone = 4921234;

Como a struct ficha pessoal possui um campo, endereco, que também é uma struct, podemos fazer acesso aos campos desta struct interna da seguinte maneira:

ficha.endereco.numero = 10;
ficha.endereco.CEP=31340230;

Desta forma, estamos acessando, primeiramente, o campo endereco da struct ficha e, dentro deste campo, estamos acessando o campo numero e o campo CEP.

13.1.3 Atribuição

Podemos atribuir duas estruturas que sejam do *mesmo* tipo. O C irá, neste caso, copiar uma estrutura, campo por campo, na outra. Veja o programa abaixo:

```
struct est1
{
    int i;
    float f;
};

void main()
{
    struct est1 primeira, segunda;
    primeira.i = 10;
    primeira.f = 3.1415;
    segunda = primeira;
   printf(" Os valores armazenasdos na segunda struct sao :  %d  e  %f
", segunda.i , segunda.f);
}
```

São declaradas duas estruturas do tipo *est1,* uma chamada *primeira* e outra chamada *segunda*. Atribuem-se valores aos dois campos da struct primeira. Os valores de *primeira* são copiados em *segunda* apenas com a expressão de atribuição:

segunda = primeira;

Todos os campos de primeira serão copiados na segunda. Note que isto é diferente do que acontecia em vetores, onde, para fazer a cópia dos elementos de um vetor em outro, tínhamos que copiar elemento por elemento do vetor. Nas structs é muito mais fácil!

Porém, devemos tomar cuidado na atribuição de structs que contenham campos ponteiros. Veja abaixo:

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

struct tipo_end
{
     char *rua;     /* A struct possui um campo que é um ponteiro */
     int numero;
};

void main()
{
   struct tipo_end end1, end2;
   char buffer[50];
   printf("\nEntre o nome da rua:");
   gets(buffer);
   end1.rua = (char *) malloc((strlen(buffer)+1)*sizeof(char));
   strcpy(end1.rua, buffer);
   printf("\nEntre o numero:");
   scanf("%d", &end1.numero);
```

```
   end2 = end1;
   printf("Depois da atribuicao:\n Endereco em end1 %s %d  \n Endereco em
end2 %s %d", end1.rua,end1.numero,end2.rua, end2.numero);

   strcpy(end2.rua, "Rua Mesquita");
   end2.numero = 1100;
   printf(" \n\nApos modificar o endereco em end2:\n Endereco em end1 %s
%d \n Endereco em end2 %s %d", end1.rua, end1.numero, end2.rua,
end2.numero);
}
```

Neste programa há um erro grave, pois ao se fazer a atribuição end2 = end1, o campo rua de end2 estará apontando para a mesma posição de memória que o campo rua de end1. Assim, ao se modificar o conteúdo apontado por end2.rua estaremos também modificando o conteúdo apontado por end1.rua !!!

13.2 Passando para funções

No exemplo apresentado no ítem usando, vimos o seguinte comando:

strcpy (ficha.nome,"Luiz Osvaldo Silva");

Neste comando um elemento de uma estrutura é passado para uma função. Este tipo de operação pode ser feita sem maiores considerações.

Podemos também passar para uma função uma estrutura inteira. Veja a seguinte função:

```
void PreencheFicha (struct ficha_pessoal ficha)
{
...
}
```

Como vemos acima é fácil passar a estrutura como um todo para a função. Devemos observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor. A estrutura que está sendo passada, vai ser copiada, campo por campo, em uma variável local da função PreencheFicha. Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função. Mais uma vez podemos contornar este pormenor usando ponteiros e passando para a função um ponteiro para a estrutura.

13.2.1 Ponteiros

Podemos ter um ponteiro para uma estrutura. Vamos ver como poderia ser declarado um ponteiro para as estruturas de ficha que estamos usando nestas seções:

```
struct ficha_pessoal *p;
```

Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C. Para usá-lo, haveria duas possibilidades. A primeira é apontá-lo para uma variável struct já existente, da seguinte maneira:

```
struct ficha_pessoal ficha;
struct ficha_pessoal *p;
p  = &ficha;
```

A segunda é alocando memória para ficha_pessoal usando, por exemplo, malloc():

```
#include <stdlib.h>
main()
{
```

```
    struct ficha_pessoal *p;
    int a = 10; /* Faremos a alocacao dinamica de 10 fichas pessoais */
    p = (struct ficha_pessoal *) malloc (a * sizeof(struct
ficha_pessoal));
    p[0].telefone = 3443768;             /* Exemplo de acesso ao campo
telefone da primeira ficha apontada por p */
    free(p);
}
```

Há mais um detalhe a ser considerado. Se apontarmos o ponteiro p para uma estrutura qualquer (como fizemos em p = &ficha;) e quisermos acessar um elemento da estrutura poderíamos fazer:

(*p).nome

Os parênteses são necessários, porque o operador . tem precedência maior que o operador * . Porém, este formato não é muito usado. O que é comum de se fazer é acessar o elemento nome através do operador seta, que é formado por um sinal de "menos" (-) seguido por um sinal de "maior que" (>), isto é: -> . 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

Fácil, não?

13.3 Unions

Uma declaração **union** determina uma *única* localização de memória onde podem estar armazenadas várias variáveis diferentes. A declaração de uma união é semelhante à declaração de uma estrutura:

union nome_do_tipo_da_union
{
tipo_1 nome_1;
tipo_2 nome_2;
...
tipo_n nome_n;
} variáveis_union;

Como exemplo, vamos considerar a seguinte união:

```
union angulo
{
      float graus;
      float radianos;
};
```

Nela, temos duas variáveis (**graus** e **radianos**) que, apesar de terem nomes diferentes, ocupam o *mesmo* local da memória. Isto quer dizer que só gastamos o espaço equivalente a um único **float**. Uniões podem ser feitas também com variáveis de diferentes tipos. Neste caso, a memória alocada corresponde ao tamanho da maior variável no union. Veja o exemplo:

```
#include <stdio.h>
#define GRAUS 'G'
#define RAD 'R'
```

```
union angulo
{
        int graus;
        float radianos;
};

void main()
{
     union angulo ang;
     char op;
     printf("\nNumeros em graus ou radianos? (G/R):");
     scanf("%c",&op);

     if (op == GRAUS)
      {
         ang.graus = 180;
        printf("\nAngulo: %d\n",ang.graus);
      }
     else

    if (op == RAD)
    {
        ang.radianos = 3.1415;
        printf("\nAngulo: %f\n",ang.radianos);
    }
   else printf("\nEntrada invalida!!\n");
   }
```

Temos que tomar o maior cuidado pois poderíamos fazer:

```
#include <stdio.h>
union numero
         {
         char Ch;
         int I;
         float F;
         };
main (void)
{
      union numero N;
      N.I = 123;
      printf ("%f",N.F);    /* Vai imprimir algo que nao e'
necessariamente 123 ...*/
      return 0;
}
```

O programa acima é muito perigoso pois você está lendo uma região da memória, que foi "gravada" como um inteiro, como se fosse um ponto flutuante. Tome cuidado! O resultado pode não fazer sentido.

13.4 Enumerações

Numa enumeração podemos dizer ao compilador quais os valores que uma determinada variável pode assumir. Sua forma geral é:

enum nome_do_tipo_da_enumeração {lista_de_valores} lista_de_variáveis;

Vamos considerar o seguinte exemplo:

```
enum dias_da_semana {segunda, terca, quarta, quinta, sexta,
                                          sabado, domingo};
```

O programador diz ao compilador que qualquer variável do tipo **dias_da_semana** só pode ter os valores enumerados. Isto quer dizer que poderíamos fazer o seguinte programa:

```
#include <stdio.h>
enum dias_da_semana {segunda, terca, quarta, quinta, sexta,
                                                  sabado, domingo};
main (void)
{
      enum dias_da_semana d1,d2;
      d1=segunda;
      d2=sexta;
      if (d1==d2)
          printf ("O dia e o mesmo.");
      else
         printf ("São dias diferentes.");

      return 0;
}
```

Você deve estar se perguntando como é que a enumeração funciona. Simples. O compilador pega a lista que você fez de valores e associa, a cada um, um número inteiro. Então, ao primeiro da lista, é associado o número zero, o segundo ao número 1 e assim por diante. As variáveis declaradas são então variáveis **int**.

13.5 sizeof

O operador **sizeof** é usado para se saber o tamanho de variáveis ou de tipos. Ele retorna o tamanho do tipo ou variável em bytes. Devemos usá-lo para garantir portabilidade. Por exemplo, o tamanho de um inteiro pode depender do sistema para o qual se está compilando. O **sizeof** é um operador porque ele é substituído pelo tamanho do tipo ou variável *no momento da compilação*. Ele não é uma função. O **sizeof** admite duas formas:

sizeof nome_da_variável

sizeof (nome_do_tipo)

Se quisermos então saber o tamanho de um **float** fazemos **sizeof(float)**. Se declararmos a variável **f** como **float** e quisermos saber o seu tamanho faremos **sizeof f**. O operador **sizeof** também funciona com estruturas, uniões e enumerações.

Outra aplicação importante do operador **sizeof** é para se saber o tamanho de tipos definidos pelo usuário. Seria, por exemplo, uma tarefa um tanto complicada a de alocar a memória para um ponteiro para a estrutura *ficha_pessoal*, criada na primeira página desta aula, se não fosse o uso de **sizeof**. Veja o exemplo:

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

void main(void)
{
        struct ficha_pessoal *ex;
        ex = (struct ficha_pessoal *) malloc(sizeof(struct ficha_pessoal));
        ...
        free(ex);
}
```

13.6 typedef

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

typedef antigo_nome novo_nome;

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

typedef int inteiro;

Agora podemos declarar o tipo **inteiro**.

O comando **typedef** também pode ser utilizado para dar nome a tipos complexos, como as estruturas. As estruturas criadas no exemplo da página anterior poderiam ser definidas como tipos através do comando **typedef**. O exemplo ficaria:

```
#include <stdio.h>
typedef struct tipo_endereco
{
        char rua [50];
        int numero;
        char bairro [20];
        char cidade [30];
        char sigla_estado [3];
        long int CEP;
} TEndereco;

typedef struct ficha_pessoal
{
        char nome [50];
        long int telefone;
        TEndereco endereco;
}TFicha;
```

```
void main(void)
{
        TFicha *ex;
        ...
}
```

Veja que não é mais necessário usar a palavra chave struct para declarar variáveis do tipo ficha pessoal. Basta agora usar o novo tipo definido TFicha.