TREINAMENTO EM LINGUAGEM C

Victorine Viviane Mizrahi

Outros Livros na Área

Jamsa	*C Library – Bibliotecas*
Mayer	*Linguagem C ANSI – Um Curso Introdutório para Programadores*
Mayer	*Integrando Clipper com Linguagem C*
Mizrahi	*Treinamento em Linguagem C – Módulo 2*
Schildt	*C Avançado – Guia do Usuário*
Schildt	*C – Guia de Referência Básica*
Schildt	*Inteligência Artificial Utilizando Linguagem C*
Schildt	*Linguagem C – Guia do Usuário*
Schildt	*Linguagem C – Guia Prático e Interativo*
Schildt	*Turbo C – Guia de Referência Básica*
Schildt	*Turbo C – Guia do Usuário*
Schildt	*Turbo C Avançado – Guia do Usuário*
Silveira	*Linguagem C – Comandos Básicos – Guia do Operador*

0-07-460855-X

MAKRON Books

VICTORINE VIVIANE MIZRAHI

LINGUAGEM C CURSO COMPLETO

MAKRON Books

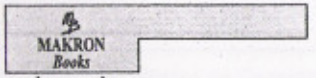
TREINAMENTO EM LINGUAGEM C

CURSO COMPLETO

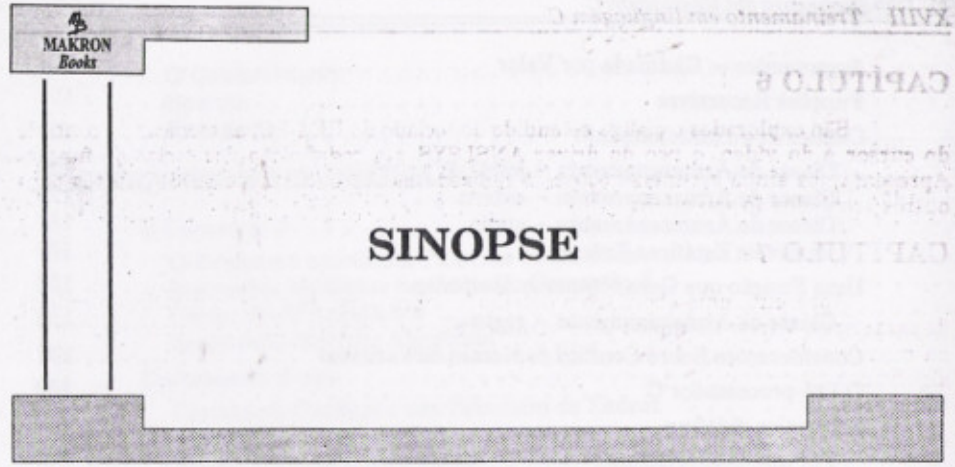
MÓDULO 1

VICTORINE VIVIANE MIZRAHI

MAKRON Books



SUMÁRIO



SINOPSE

CAPÍTULO 1

Apresenta uma pequena descrição sobre compiladores e interpretadores e uma introdução à linguagem C, incluindo: a forma geral de um programa em C, a função printf() para impressão de dados e os tipos de variáveis em C.

CAPÍTULO 2

Trata dos operadores aritméticos e relacionais, da função scanf() para leitura de dados e como documentar programas com comentários.

CAPÍTULO 3

Aborda as estruturas que possibilitam a repetição de uma seqüência de instruções até que uma certa condição seja satisfeita.

CAPÍTULO 4

Neste capítulo são mostradas as estruturas de decisão que permitem praticar testes para decidir entre ações alternativas e os operadores lógicos.

CAPÍTULO 5

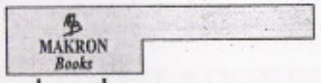
Neste capítulo você vai aprender a criar suas próprias funções e manipular seus argumentos. São mostrados os tipos de funções C, funções recursivas e funções para a geração de números aleatórios. São apresentadas as diretivas do Pré-Processador C, como definir e usar macros e uma pequena discussão sobre as vantagens e desvantagens do uso de macros.

CAPÍTULO 6

São explorados o código estendido do teclado do IBM-PC, as técnicas de controle do cursor e do vídeo, o uso do driver ANSI.SYS e a redefinição das teclas de função. Apresentamos ainda algumas noções de redirecionamento dos dispositivos padrões para outros acoplados ao seu micro.

CAPÍTULO 7

Apresenta uma abordagem detalhada sobre o uso de vetores, matrizes e cadeias de caracteres. Vários exemplos são discutidos e mostram como uma matriz é armazenada na memória do computador e como o seu uso pode auxiliar na implementação de vários algoritmos. São apresentadas as funções de biblioteca C para a manipulação de cadeias de caracteres.



PREFÁCIO

Este livro foi planejado para um curso completo de programação em linguagem C. O estudante deve ter conhecimentos básicos de sistema operacional nos equipamentos da família IBM-PC, XT, AT e PS/2.

O objetivo principal é o de apresentar a linguagem C de maneira simples e clara e mostrar como C pode ser usada para a criação de programas sérios. Os exemplos apresentados não estão relacionados a nenhum assunto específico e podem ser facilmente adaptados a qualquer área de aplicação.

O livro apresenta uma organização didática dos temas abordados e pode ser adotado em cursos de computação de escolas técnicas ou universidades. Ao aluno interessado é dada a possibilidade de criar novos programas e a de desenvolver novas idéias a partir das apresentadas.

É certamente possível aprender C como primeira linguagem de programação. Entretanto sua sintaxe não é tão fácil como a de uma linguagem como BASIC e programadores iniciantes podem sentir alguma dificuldade.

Os exemplos deste livro foram processados utilizando-se o compilador TURBO C versão 2.0 da Borland para computadores IBM-PC e sistema operacional MS-DOS versão 3.3, mas podem ser processados em qualquer compilador compatível com o MS-DOS ou UNIX.

MAKRON Books

CAPÍTULO 1

INTRODUÇÃO

- *Compiladores*
- *Interpretadores*
- *Estrutura Básica de um Programa C*
- *As Funções main( ) e printf( )*
- *Constantes*
- *Tipos de Variáveis*
- *Nomes de Variáveis*
- *Palavras-Chaves*
- *Caracteres Gráficos*

INTRODUÇÃO

A linguagem C foi primeiramente criada por Dennis M. Ritchie e Ken Thompson no laboratório Bell em 1972, baseada na linguagem B de Thompson que era uma evolução da antiga linguagem BCPL. B foi nomeada com a primeira letra de BCPL e C com a segunda. A próxima linguagem progressiva da idéia de C provavelmente se chamará P.

A definição de C está contida no livro *The C Programming Language*, escrito por Brian W. Kernighan e Dennis M. Ritchie.

C é uma linguagem vitoriosa como ferramenta na programação de qualquer tipo de sistema (sistemas operacionais, planilhas eletrônicas, processadores de texto, gerenciadores de banco de dados, processadores gráficos, sistemas de transmissão de dados, para solução de problemas de engenharia ou física etc.). Como exemplo, o sistema operacional UNIX é desenvolvido em C.

A linguagem de programação C tornou-se rapidamente uma das mais importantes e populares, principalmente por ser muito poderosa, portátil, pela padronização dos compiladores existentes e flexível.

C foi desenhada para que o usuário possa planejar programas estruturados e modulares. O resultado é um programa mais legível e documentado. Os programas em C tendem a ser bastante compactos e de execução rápida.

C é amiga do programador, suficientemente estruturada para encorajar bons hábitos de programação. Programas em C podem ser desenvolvidos em partes separadas por pessoas distintas e depois unidos num produto final, o que significa que bibliotecas de funções podem ser criadas ou usadas sem realmente conhecer o código de cada uma delas.

Existem muitas outras virtudes de C que você conhecerá ao longo do seu aprendizado.

Paulo Régis C. de Araújo
Coordenador de Informática

COMPILADORES E INTERPRETADORES

A maneira de se comunicar com um computador chama-se programa e a única linguagem que o computador entende chama-se linguagem de máquina. Portanto todos os programas que se comunicam com a máquina devem estar em linguagem de máquina.

A forma como os programas são traduzidos para a linguagem de máquina classifica-os em duas categorias:

- INTERPRETADOS
- COMPILADOS

Os programas que fazem estas traduções são chamados interpretadores e compiladores.

INTERPRETADORES

Um interpretador lê a primeira instrução do programa, faz uma consistência de sua sintaxe e se não houver erro converte-a para linguagem de máquina para finalmente executá-la. Segue, então, para a próxima instrução, repetindo o processo até que a última instrução seja executada ou a consistência aponte algum erro.

O interpretador precisa estar presente todas as vezes que vamos executar o nosso programa e o trabalho de checagem da sintaxe e tradução deverá ser repetido. Se uma parte do programa necessitar ser executada muitas vezes, o processo é feito o mesmo número de vezes.

COMPILADORES

Um compilador lê a primeira instrução do programa, faz uma consistência de sua sintaxe e se não houver erro converte-a para linguagem de máquina e, em vez de executá-la, segue para a próxima instrução repetindo o processo até que a última instrução seja atingida ou a consistência aponte algum erro.

Se não houver erros, o compilador gera um programa em disco com o sufixo **.OBJ** com as instruções já traduzidas. Este programa não pode

ser executado até que sejam agregadas a ele rotinas em linguagem de máquina que lhe permitirão a sua execução. Este trabalho é feito por um programa chamado "linkeditor" que, além de juntar as rotinas necessárias ao programa **.OBJ**, cria um produto final em disco com o sufixo **.EXE** que pode ser executado diretamente do sistema operacional.

Com isto a velocidade de execução do programa chega a ser 15 a 20 vezes mais rápida do que quando o programa é interpretado.

Além da velocidade, outras vantagens podem ser mencionadas:

- é desnecessária a presença do interpretador ou do compilador para executar o programa já compilado e linkeditado;
- programas **.EXE** não podem ser alterados, o que protege o código-fonte.

Apesar de um compilador ter muitas vantagens sobre um interpretador, algumas considerações devem ser feitas.

Os interpretadores permitem que se produzam programas com maior facilidade na fase de aprendizado, pois podem ser criados e executados imediatamente, enquanto um compilador requer muito mais tempo para que um programa possa ser executado.

Um compilador não criará um programa em linguagem de máquina antes que este esteja absolutamente livre de erros, e durante o processo de aprendizado de uma linguagem a quantidade de erros gerados é relativamente grande, o que faz com que um interpretador seja muito mais conveniente.

EXISTEM, NO MERCADO, COMPILADORES E INTERPRETADORES DA LINGUAGEM C.

COMO CRIAR UM PROGRAMA EXECUTÁVEL

- Datilografar seu programa com o auxílio de um processador de textos no modo não documento. Gravar o programa em disco dando a ele um nome com o sufixo **.C**. O programa gerado é chamado de **fonte**.

- Compilar o fonte seguindo as instruções do seu compilador, o que criará um programa com o sufixo **.OBJ** em disco. O programa gerado é chamado de **objeto**.
- Linkeditar o objeto seguindo as instruções do seu linkeditor o que criará um programa com o sufixo **.EXE** em disco. O programa gerado é chamado de **executável**.

A ESTRUTURA BÁSICA DE UM PROGRAMA EM C

Um programa C consiste em uma ou várias "funções". Os nomes programa e função se confundem em C.

FORMA GERAL DAS FUNÇÕES C

Vamos começar pelo menor programa possível em C.

```
main()
{
}
```

Este programa compõe-se de uma única função chamada **main**.

```
main()  ———————  primeira função a ser executada
{       ———————  inicia o corpo da função
}       ———————  termina a função
```

Os parênteses após o nome indicam que esta é uma função. O nome de uma função C pode ser qualquer um com exceção de "main", reservado para a função que inicia a execução do programa.

Toda função C deve ser iniciada por uma chave de abertura, {, e encerrada por uma chave de fechamento, }.

O nome da função, os parênteses e as chaves são os únicos elementos obrigatórios de uma função.

Você pode colocar espaços, caracteres de tabulação e pular linhas à vontade em seu programa, pois o compilador ignora estes caracteres. Em C não há um estilo obrigatório.

Os exemplos a seguir mostram como o estilo de programação pode causar problemas de legibilidade aos programas C.

`main () {}`	`main ()` `{ }`

`main`
`(`
`)`
`{`
`}`
	`main`
	`() {}`

Todos estes exemplos são de programas que não fazem nada, ou seja, programas vazios.

A FUNÇÃO main()

A função **main()** deve existir em algum lugar de seu programa e marca o ponto de início da execução do programa. Se um programa for constituído de uma única função esta será **main()**.

INSTRUÇÕES DE PROGRAMA

Vamos adicionar uma instrução em nosso programa.

```
main()
{
   printf ("primeiro programa");
}
```


Todas as instruções devem estar dentro das chaves que iniciam e terminam a função e são executadas na ordem em que as escrevemos.

As instruções C são sempre encerradas por um ponto-e-vírgula (;). O ponto-e-vírgula é parte da instrução e não um simples separador.

Esta instrução é uma chamada à função **printf()**, os parênteses nos certificam disso e o ponto-e-vírgula indica que esta é uma instrução.

Como em C não existe um estilo obrigatório, vamos reescrever o programa anterior de forma que a sua execução será exatamente como a do anterior:

```
main(){printf("primeiro programa");}
```

A FUNÇÃO printf()

A função **printf()** é uma das funções de E/S (entrada e saída) que podem ser usadas em C. Ela não faz parte da definição de C mas todos os sistemas têm uma versão de **printf()** implementada.

Os parênteses indicam que estamos, realmente, procedendo com uma função. No interior dos parênteses estão as informações passadas pelo programa **main()** a função **printf()**, isto é "primeiro programa". Esta informação é chamada de argumento de **printf()**.

Quando o programa encontra esta linha, passa o controle para a função **printf()** que imprime na tela do seu computador

primeiro programa

e, quando encerra a execução desta, o controle é transferido novamente para o programa.

Sintaxe:

printf("expr. de controle", lista de argumentos)

Outro exemplo:

```
main()
{
   printf("Este e' o numero dois: %d",2);
}
```

Este programa imprimirá na tela do seu computador:

Este e' o numero dois: 2

A função **printf()** pode ter um ou vários argumentos. No primeiro exemplo nós colocamos um único argumento: "primeiro programa". Agora, entretanto, colocamos dois: "Este e' o numero: **%d**" que está à esquerda e o valor 2 à direita. Estes dois argumentos são separados por uma vírgula.

A expressão de controle pode conter caracteres que serão exibidos na tela e códigos de formatação que indicam o formato em que os argumentos devem ser impressos. No nosso exemplo o código de formatação **%d** solicita a **printf()** imprimir o segundo argumento em formato decimal.

printf() é uma função da biblioteca padrão de C e pode receber um número variável de argumentos. Isto é, a cadeia de caracteres de controle e mais tantos argumentos quantas especificações de formato a cadeia de controle contiver.

Cada argumento deve ser separado por uma vírgula.

IMPRIMINDO CADEIA DE CARACTERES

Além do código de formatação decimal **(%d)**, **printf()** aceita vários outros. O próximo exemplo mostra o uso do código **%s** para imprimir uma cadeia de caracteres.

```
main()
{
   printf("%s esta a %d milhoes de milhas\ndo sol","Venus",67);
}
```


A saída será:

Venus esta a 67 milhoes de milhas
do sol

Aqui, além do código de formatação, a expressão de controle de **printf()** contém um conjunto de caracteres estranho: **\n**.

O **\n** é um código especial que informa a **printf()** que o restante da impressão deve ser feito em nova linha. A combinação de caracteres **\n** representa, na verdade, um único caractere em C, chamado de nova-linha. Em outras palavras, este caractere desempenha a mesma função que a executada quando pressionamos a tecla **[enter]** do teclado. Por que, então, não usar a tecla **[enter]** ? Porque, quando pressionamos a tecla **[enter]**, o editor de textos que estamos usando para editar nosso programa deixa a linha atual onde estamos trabalhando e passa para outra linha, deixando a linha anterior inacabada, e a função **printf()** não o tomará para a impressão.

Os caracteres que não podem ser obtidos diretamente do teclado para dentro do programa (como a mudança de linha) são escritos em C, como a combinação do sinal \ (barra invertida) com outros caracteres. Por exemplo, **\n** representa a mudança de linha.

Vamos agora escrever um programa com mais de uma instrução:

```
main()
{
   printf("A letra %c ",'j');
   printf("pronuncia-se %s.","jota");
}
```

A saída será:

A letra j pronuncia-se jota.

'j' é um caractere e "jota" é uma cadeia de caracteres.

Note que 'j' é delimitado por aspas simples enquanto que "jota" é delimitado por aspas duplas. Isto indica ao compilador como diferenciar um caractere de uma cadeia de caracteres. Observe também que a saída é feita em duas linhas de programa, o que não constitui duas linhas impressas de texto. A função **printf()** não imprime automaticamente um caractere de nova linha; se você desejar, deve inserir um explicitamente.

A tabela seguinte mostra os códigos de C para caracteres que não podem ser inseridos diretamente do teclado. A função **printf()** aceita todos eles.

CÓDIGOS ESPECIAIS	SIGNIFICADO
\n	NOVA LINHA
\r	RETORNO DO CURSOR
\t	TAB
\b	RETROCESSO
\"	ASPAS
\\	BARRA
\f	SALTA PÁGINA DE FORMULÁRIO
\0	NULO

A próxima tabela mostra os códigos para impressão formatada de **printf()**.

CÓDIGO printf()	FORMATO
%c	CARACTERE SIMPLES
%d	DECIMAL
%e	NOTAÇÃO CIENTÍFICA
%f	PONTO FLUTUANTE
%g	%e OU %f (O MAIS CURTO)
%o	OCTAL
%s	CADEIA DE CARACTERES
%u	DECIMAL SEM SINAL
%x	HEXADECIMAL
%ld	DECIMAL LONGO
%lf	PONTO FLUTUANTE LONGO (DOUBLE)

CONSTANTES E VARIÁVEIS

Uma constante tem valor fixo e inalterável.

Nos exemplos anteriores, mostramos o uso de constantes numéricas, cadeias de caracteres e caracteres em **printf()**.

Em C uma constante caractere é escrita entre aspas simples, uma constante cadeia de caracteres entre aspas duplas e constantes numéricas como o número propriamente dito.

Exemplos de constantes:

```
'c'
"primeiro programa"
8
```

O programa

```
main()
{
   printf("Este e' o numero dois: %d",2);
}
```

imprime a constante 2, no formato especificado, %d.

Certamente esta não é a maneira mais simples de obter o mesmo resultado:

```
main()
{
   printf("Este e' o numero dois: 2");
}
```

As **variáveis** são o aspecto fundamental de qualquer linguagem de computador. Uma variável em C é um espaço de memória reservado para armazenar um certo tipo de dado e tendo um nome para referenciar o seu conteúdo.

O espaço de memória de uma variável pode ser compartilhado por diferentes valores segundo certas circunstâncias. Em outras palavras, uma variável é um espaço de memória que pode conter, a cada tempo, valores diferentes.

Para explicar o uso de variáveis vamos reescrever o programa anterior usando uma variável ao invés de uma constante:

```
main()
{
   int num;
   num=2;
   printf("Este e' o numero dois: %d",num);
}
```

A execução deste programa é exatamente a mesma que a da versão anterior porém ele cria a variável **num**, atribui a ela o valor 2 e imprime o valor contido nela.

A primeira instrução,

```
int num;
```

é um exemplo de declaração de variável, isto é, apresenta um tipo, **int**, e um nome, **num**.

A segunda instrução,

```
num=2;
```

atribui um valor à variável e este valor será acessado através de seu nome. Usamos o operador de atribuição (=) para este fim.

A terceira instrução chama a função **printf()** mandando o nome da variável como argumento, substituindo a constante 2 usada anteriormente.

DECLARAÇÕES DE VARIÁVEIS

Uma declaração de variável é uma instrução para reservar uma quantidade de memória apropriada para armazenar o tipo especificado, neste caso **int**, e indicar que o seu conteúdo será referenciado pelo nome dado, neste caso **num**.

```
int num;
```

UMA DECLARAÇÃO DE VARIÁVEL CONSISTE NO NOME DE UM TIPO, SEGUIDO DO NOME DA VARIÁVEL.

Em C todas as variáveis devem ser declaradas.

Se você tiver mais de uma variável do mesmo tipo, poderá declará-las de uma única vez, separando seus nomes por vírgulas.

```
int aviao, foguete, helicoptero;
```


POR QUE DECLARAR VARIÁVEIS

- Reunir variáveis em um mesmo lugar, dando a elas nomes significativos, facilita ao leitor entender o que o programa faz.
- Uma seção de declarações de variáveis encoraja o planejamento do programa antes de começar a escrevê-lo. Isto é, planejar as informações que devem ser dadas ao programa e quais as que o programa deverá nos fornecer.
- Declarar variáveis ajuda a prevenir erros. Por exemplo, se escrevermos 0 (zero) em vez de O:

```
int BOBO;
B0B0 = 5;
```

  o compilador acusará o erro.

- Se os motivos anteriores não convenceram você, este deverá convencê-lo:

C NÃO TRABALHA SE VOCÊ NÃO DECLARAR SUAS VARIÁVEIS.

TIPOS DE VARIÁVEIS

O tipo de uma variável informa a quantidade de memória, em bytes, que esta irá ocupar e a forma como o seu conteúdo será armazenado.

Em C existem 5 tipos de variáveis básicas. Nos computadores da linha IBM-PC a tabela seguinte é válida:

TIPO	BIT	BYTES	ESCALA
char	8	1	-128 a 127
int	16	2	-32768 a 32767
float	32	4	3.4E-38 a 3.4E+38
double	64	8	1.7E-308 a 1.7E+308
void	0	0	sem valor

Com exceção de **void**, os tipos de dados básicos podem estar acompanhados por modificadores na declaração de variáveis. Os modificadores de tipos oferecidos por C são:

long ou long int (4 bytes)
unsigned char (de 0 a 255)
unsigned int (de 0 a 65535)
unsigned long
short (2 bytes no IBM-PC)

O tipo **short** tem tamanho diferente do tipo **int** em outros computadores (no computador IBM-370, por exemplo, o tipo **short** tem a metade do tamanho de um inteiro).

O tipo **int** tem sempre o tamanho da palavra da máquina, isto é, em computadores de 16 bits ele terá 16 bits de tamanho.

Vamos examinar um programa que usa variáveis caractere, ponto flutuante e inteiras. Chamaremos este programa de *evento.c*.

```
main()
{
   int evento;
   char corrida;
   float tempo;
   evento=5;
   corrida='C';
   tempo=27.25;
   printf("O tempo vitorioso na eliminatoria %c",corrida);
   printf("\nda competicao %d foi %f.",evento,tempo);
}
```

A saída será:

O tempo vitorioso na eliminatoria C
da competicao 5 foi 27.25.

Este programa usa os 3 tipos de variáveis mais comuns: **int**, **char** e **float**.

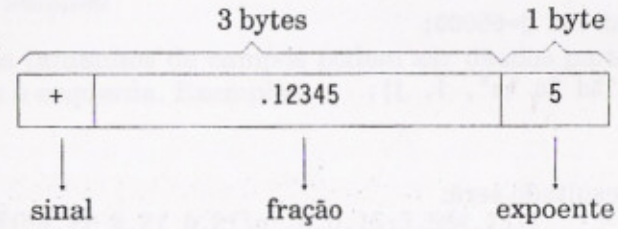
VARIÁVEIS PONTO FLUTUANTE

Números em ponto flutuante correspondem mais ou menos aos que os matemáticos chamam de "números reais".

Existem várias maneiras de escrever números em ponto flutuante. A notação "3.16e7" é um meio de indicar que 3.16 será multiplicado por 10 elevado a potência 7, isto é, 31600000. Esta indicação chama-se **notação científica** e é usada para armazenar números em ponto flutuante na memória do computador.

Assim, números em ponto flutuante são guardados na memória em duas partes. Estas duas partes são chamadas de **mantissa** e **expoente**. A mantissa é o valor do número e o expoente é a potência que irá aumentá-lo.

Por exemplo, 12345 é representado por .12345e5 onde o número que segue o (e) é o expoente, e .12345 é o valor do número.



Variáveis tipo **float** são guardadas em 4 bytes; um para o expoente e 3 para o valor do número e o sinal.

INICIALIZANDO VARIÁVEIS

É possível combinar uma declaração de variável com o operador de atribuição para que a variável tenha um valor ao mesmo tempo de sua declaração; é o que chamaremos de inicialização de variável. Como exemplo reescreveremos o programa *evento.c* inicializando as variáveis na sua declaração.

```
main()
{
   int evento=5;
   char corrida='C';
   float tempo=27.25;
   printf("O tempo vitorioso na eliminatoria %c",corrida);
   printf("\nda competicao %d foi %f.",evento,tempo);
}
```

A execução do programa será exatamente a mesma do da versão anterior.

INTEIROS COM E SEM SINAL

O modificador de tipo **unsigned** indica que o tipo associado deve ter seu bit de ordem superior interpretado de maneira diferente. Observe o programa a seguir:

```
main()
{
   unsigned int j=65000;
   int i=j;
   printf("%d %u \n", i, j);
}
```

O resultado será:

```
-536 65000
```

A razão disto está na maneira como o computador interpreta o bit de ordem superior do inteiro, ou seja, o bit 15.

Na forma binária o bit 15 de um inteiro positivo é sempre 0 e o de um inteiro negativo é sempre 1. Se usarmos o modificador **unsigned** em nossos programas, o computador irá ignorar o bit de sinal tratando-o como um bit a mais para números positivos.

Os números negativos são conhecidos como **complemento de dois** dos números positivos, pois a conversão de um número positivo para o seu negativo é feita por um processo de duas etapas. No momento você não precisa entender bem como é feita esta conversão. No segundo volume deste livro voltaremos a este assunto.

No nosso exemplo usamos o número 65000 que é maior que o maior inteiro interpretado com sinal, o que deixa evidente que o seu bit 15 é 1.

NOMES DE VARIÁVEIS

A escolha de nomes significativos para suas variáveis pode ajudá-lo a entender o que o programa faz e prevenir erros.

Você pode usar quantos caracteres quiser para um nome de variável com o primeiro sendo obrigatoriamente uma letra ou o caractere de sublinhar e os demais podendo ser letras, números ou caracteres de sublinhar.

Uma variável não pode ter o mesmo nome de uma palavra-chave de C.

Em C letras minúsculas e maiúsculas são diferentes.

PESO
Peso
peso
peSo

Em C os 32 primeiros caracteres são significativos, mas o sistema operacional MS-DOS limita os nomes em 8 caracteres significativos, isto é:

Altura_de_Maria
Altura_de_Pedro

são considerados o mesmo nome em programas processados sob o sistema operacional MS-DOS.

PALAVRAS-CHAVES EM C

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

EXPLORANDO A FUNÇÃO printf()

Como já vimos, a função **printf()** usa o caractere % seguido de uma letra para identificar o formato de impressão. O problema surge quando queremos imprimir o caractere %. Se usarmos simplesmente % na expressão de controle de **printf()**, alguns compiladores acharão que não especificamos o formato corretamente e acusarão um erro. O caminho é usar dois símbolos %.

```
main()
{
   int reajuste = 10;
   printf("O reajuste foi de %d%%.\n",reajuste);
}
```

A saída será:

O reajuste foi de 10%.

TAMANHO DE CAMPOS NA IMPRESSÃO

Em **printf()** é possível estabelecer o tamanho mínimo para a impressão de um campo.

```
main()
{
   printf("Os alunos sao %2d.\n",350);
   printf("Os alunos sao %4d.\n",350);
   printf("Os alunos sao %5d.\n",350);
}
```

A saída será:

Os alunos sao 350.
Os alunos sao 350.
Os alunos sao   350.

Pode-se usar tamanho de campos com números em ponto flutuante para obter precisão e arredondamento.


```
main()
{
   printf("%4.2f\n",3456.78);
   printf("%3.2f\n",3456.78);
   printf("%3.1f\n",3456.78);
   printf("%10.3f\n",3456.78);
}
```

A saída será:

3456.78
3456.78
3456.8
** 3456.780**

Os tamanhos de campos podem ser usados para alinhamento à direita ou à esquerda. Exemplos:

```
main()
{
   printf("%.2f %.2f %.2f\n",8.0,15.3,584.13);
   printf("%.2f %.2f %.2f\n",834.0,1500.55,4890.21);
}
```

8.00 15.30 584.13
834.00 1500.55 4890.21

```
main()
{
   printf("%10.2f %10.2f %10.2f\n",8.0,15.3,584.13);
   printf("%10.2f %10.2f %10.2f\n",834.0,1500.55,4890.21);
}
```

**   8.00   15.30   584.13**
**   834.00   1500.55   4890.21**

O sinal de menos (–) precedendo a especificação do tamanho do campo justifica os campos à esquerda, como mostra o próximo programa:

```
main()
{
   printf("%-10.2f %-10.2f %-10.2f\n",8.0,15.3,584.13);
```

```
    printf("%-10.2f %-10.2f %-10.2f\n",834.0,1500.55,4890.21);
}
```

```
8.00        15.30       584.13
834.00      1500.55     4890.21
```

Este formato é muito útil em certas circunstâncias, especialmente quando imprimimos cadeia de caracteres.

COMPLEMENTANDO COM ZEROS À ESQUERDA

Além de especificar o tamanho do campo, podemos complementar o campo todo ou parte dele com zeros à esquerda. Observe o exemplo a seguir.

```
main()
{
   printf("\n%04d",21);
   printf("\n%06d",21);
   printf("\n%6.4d",21);
   printf("\n%6.0d",21);
}
```

A saída será:

```
0021
000021
  0021
    21
```

IMPRIMINDO CARACTERES

Em C um caractere pode ser representado de diversas maneiras: o próprio caractere entre aspas simples ou sua representação decimal, hexadecimal ou octal segundo a tabela ASCII.

Por exemplo, a instrução:

```
printf("%d %c %x %o \n",'A','A','A','A');
```


imprime

```
65 A 41 101
```

e a instrução

```
printf("%c %c %c %c \n",'A',65,0x41,0101);
```

imprime

```
A A A A
```

Um octal em C sempre é iniciado por 0 e um hexadecimal por 0x para que o compilador saiba diferenciá-los de números decimais.

A tabela ASCII tem 256 códigos decimais numerados de 0 a 255. Se imprimirmos em formato caractere um número maior que 255, o computador imprimirá o equivalente ao resto da divisão do número por 256, ou seja, se o número for 3393 ele será impresso como 'A' pois o resto da divisão de 3393 por 256 é 65.

```
printf("%d %c \n",3393,3393);
```

```
3393 A
```

IMPRIMINDO CARACTERES GRÁFICOS

Como você provavelmente sabe, todo caractere (letra, dígito, caractere de pontuação etc...) é representado, no computador, por um número. O código ASCII dispõe de números de 0 a 127 (decimal) cobrindo letras, dígitos de 0 a 9, caracteres de pontuação e caracteres de controle como salto de linha, tabulação etc...

Os computadores IBM usam 128 caracteres adicionais com códigos de 128 a 255 que consistem em símbolos de línguas estrangeiras e caracteres gráficos.

Já mostramos como imprimir caracteres ASCII usando a função **printf()**. Os caracteres gráficos e outros não standard requerem uma

outra maneira de escrita para serem impressos. A forma de se representar um caractere de código acima de 127 decimal é:

\xdd

onde **dd** representa o código do caractere em notação hexadecimal. Observe que **\xdd** é um caractere e pode ser usado na expressão de controle de **printf()** como qualquer outro caractere.

Usaremos este formato para impressão de qualquer caractere gráfico.

O programa a seguir imprime um carro e uma caminhonete usando caracteres gráficos:

```
main()
{
   printf("\n\n\n");
   printf("\n \xDC\xDC\xDB\xDB\xDB\xDB\xDC\xDC");
   printf("\n \xDF0\xDF\xDF\xDF\xDF0\xDF");
   printf("\n\n\n");
   printf("\n \xDC\xDC\xDB \xDB\xDB\xDB\xDB\xDB\xDB");
   printf("\n \xDF0\xDF\xDF\xDF\xDF\xDF00\xDF");
   printf("\n\n\n");
}
```

O próximo exemplo será denominado *box.c* e imprimirá uma moldura na tela.

```
main()
{
   printf("\xC9\xCD\xBB\n");
   printf("\xBA \xBA\n");
   printf("\xC8\xCD\xBC\n");
}
```


A saída será:

```
C9 CD BB
BA    BA
C8 CD BC
```

REVISÃO

1. Todo programa C deve ter uma função chamada **main()**, ela é a primeira função a ser executada.
2. Toda instrução C é terminada por um ponto-e-vírgula.
3. A função **printf()** é usada para enviar informações à tela. Os seus argumentos consistem em uma expressão de controle contendo caracteres e códigos de formatação iniciados pelo caractere %, e tantos argumentos quantos forem os códigos de formatação colocados na expressão de controle.
4. Os caracteres que não podem ser obtidos diretamente do teclado são escritos com a combinação do caractere \ seguido por outro caractere, segundo a tabela de códigos especiais.
5. Um programa C deve declarar todas as suas variáveis antes de usá-las. Os 5 tipos básicos de variáveis C são: *char*, *int*, *float*, *double* e *void*.
6. Em C, letras maiúsculas e minúsculas são tratadas diferentemente.
7. Um caractere gráfico é representado em C pela forma **\xdd** onde **dd** é o código ASCII hexadecimal do caractere.
8. O programa termina sua execução quando é encontrada a chave de fechamento da função **main()**.

EXERCÍCIOS

1. Um dos alunos preparou o seguinte programa e o apresentou para ser avaliado. Ajude-o.

```
main()
{
   printf(Existem %d semanas no ano.,56);
}
```

2. O programa seguinte tem vários erros em tempo de compilação. Execute-o e observe as mensagens apresentadas por seu compilador.

```
Main()
{
   int a=1; b=2, c=3;
   printf("Os numeros sao: %d %d %d\n,a,b,c,d)
}
```

3. Qual será a saída do programa abaixo:

```
main()
{
   printf("%s\n%s\n%s","um","dois","tres");
}
```

4. Qual será a impressão obtida por cada uma destas instruções? Assuma que fazem parte de um programa completo.

   a) printf("Bom Dia ! Shirley.");
   printf("Voce ja tomou cafe ?\n");
   b) printf("A solucao nao existe!\nNao insista");
   c) printf("Duas linhas de saida\nou uma ?");

5. Identifique o tipo das seguintes constantes:

a) '\r'	*b)* 2130	*c)* -123
d) 33.28	*e)* 0x42	*f)* 0101
g) 2.0e30	*h)* '\xDC'	*i)* '\"'
j) '\\'	*k)* 'F'	*l)* 0
m) '\0'		

6. O que é uma variável em C ?

7. Quais os 5 tipos básicos de variáveis em C ?

8. Quais dos seguintes nomes são válidos para variáveis em C?

a) 3ab	*b)* _sim	*c)* n_a_o
d) 00FIM	*e)* int	*f)* A123
g) x**x	*h)* ___A	*i)* y-2
j) OOFIM	*k)* \meu	*l)* *y2

9. Quais das seguintes instruções são corretas ?

   a) int a;
   b) float b;
   c) double float c;
   d) unsigned char d;
   e) long float e;

10. O tipo **float** ocupa o mesmo espaço que _____ variáveis do tipo char.

11. *Verdadeiro ou Falso*: tipos de variáveis **long int** podem conceber números não maiores que o dobro da maior variável do tipo **int**.

12. Escreva um programa que contenha uma única instrução e imprima na tela:

```
Esta e' a linha um.
Esta e' a linha dois.
```

13. Escreva um programa que imprima na tela:

```
um
      dois
            tres
```

14. Escreva um programa que declare 3 variáveis inteiras e atribua os valores 1,2 e 3 a elas; 3 variáveis caracteres e atribua a elas as letras a, b, e c; finalmente imprima na tela:

As variaveis inteiras contem os numeros 1, 2, e 3.
As variaveis caracteres contem os valores a, b, e c.

15. Reescreva o programa *box.c* para que desenhe uma moldura similar, mas que tenha 4 caracteres de largura e 4 caracteres de altura. Use o caractere ‖, de código BA hexa, para complementar a moldura.

MAKRON Books

CAPÍTULO 2

OPERADORES

- *A Função scanf( )*
- *O Operador de Endereço (&)*
- *getche( ) e getch( )*
- *getchar( ) e putchar( )*
- *Os Operadores Aritméticos*
  = + - * / %
- *Os Operadores de Incremento e Decremento*
  + + --
- *Os Operadores Aritméticos de Atribuição*
  + = - = * = / = % =
- *Os Operadores Relacionais*
  > < > = < = = = ! =
- *Comentários*
  /* e */

A FUNÇÃO scanf()

A função **scanf()** é outra das funções de E/S implementadas em todos os compiladores C. Ela é o complemento de **printf()** e nos permite ler dados formatados da entrada padrão (teclado).

Sua sintaxe é similar à de **printf()**, isto é, uma expressão de controle seguida por uma lista de argumentos separados por vírgulas.

A principal diferença está na lista de argumentos. Os argumentos de **scanf()** devem ser endereços de variáveis.

Sintaxe:

scanf("expressão de controle", lista de argumentos)

A expressão de controle pode conter códigos de formatação, precedidos por um sinal % ou ainda o caractere * colocado após o % que avisa à função que deve ser lido um valor do tipo indicado pela especificação, mas não deve ser atribuído a nenhuma variável (não deve ter parâmetros na lista de argumentos para estas especificações).

A lista de argumentos deve consistir nos endereços das variáveis. C oferece um operador para tipos básicos chamado **operador de endereço** e referenciado pelo símbolo **&** que resulta o endereço do operando.

O OPERADOR DE ENDEREÇO (&)

A memória de seu computador é dividida em bytes, e estes bytes são numerados de 0 até o limite de memória de sua máquina (524.287 se você tem 512K de memória). Estes números são chamados de "endereços" de bytes. Um endereço é o nome que o computador usa para identificar a variável.

Toda variável ocupa uma certa localização na memória, e seu endereço é o do primeiro byte ocupado por ela. Um inteiro ocupa 2 bytes. Se você declarou a variável **n** como inteira e atribuiu a ela o valor 2, quando **n** for referenciada devolverá 2. Entretanto, se você referenciar **n** precedido de **& (&n)** devolverá o endereço do primeiro byte onde **n** está guardada.

O programa seguinte imprime o valor e o endereço de **n**:

```
main()
{
   int num;
   num=2;
   printf("Valor=%d, endereco=%u",num,&num);
}
```

Um endereço de memória é visto como um número inteiro sem sinal, por isso usamos **%u**.

A saída deste programa varia conforme a máquina e a memória do equipamento, um exemplo é:

Valor=2, endereco=1370

CÓDIGO DE FORMATAÇÃO DA FUNÇÃO scanf()

CÓDIGO	FUNÇÃO
%c	Leia um único caractere
%d	Leia um inteiro decimal
%e	Leia um número em notação científica
%f	Leia um número em ponto flutuante
%o	Leia um inteiro octal
%s	Leia uma série de caracteres
%x	Leia um número hexadecimal
%u	Leia um decimal sem sinal
%l	Leia um inteiro longo
%lf	Leia um double

Vamos escrever um programa para exemplificar a função **scanf()**, que chamaremos de *idade.c*.

```
main()
{
   float anos,dias;
   printf("Digite sua idade em anos: ");
   scanf("%f",&anos);
   dias = anos*365;
   printf("Sua idade em dias e' %.0f.\n",dias);
}
```

Eis uma execução do programa:

```
C>idade
Digite sua idade em anos: 4
Sua idade em dias e' 1460.
```

Visto que usamos variáveis float podemos entrar com frações decimais:

```
C>idade
Digite sua idade em anos: 12.5
Sua idade em dias e' 4562.
```

Neste programa utilizamos variável **float** ao invés de **int** para entrar com frações decimais em anos e para obter números maiores para dias.

O próximo programa, além de mostrar um outro uso de **scanf()**, mostra também a saída formatada **printf()**.

```
main()
{
   char a;
   printf("Digite um caractere e veja-o em decimal,");
   printf(" octal e hexadecimal.\n");
   scanf("%c",&a);
   printf("\n%c=%d dec.,%o oct. e %x hex.\n",a,a,a,a);
}
```

Eis uma execução do programa:

```
C>codchar
Digite um caractere e veja-o em decimal, octal e hexadecimal.
m
m=109 dec.,155 oct. e 6D hex.
```

AS FUNÇÕES getche() e getch()

Em algumas situações, a função **scanf()** não se adapta perfeitamente pois você precisa pressionar **[enter]** depois da sua entrada para que **scanf()** termine a leitura.

A biblioteca de C oferece funções que lêem um caractere no instante em que é datilografado, sem esperar **[enter]**.

A função **getche()** lê o caractere do teclado e permite que seja impresso na tela. Esta função não aceita argumentos e devolve o caractere lido para a função que a chamou.

O programa seguinte chama a função **getche()** e atribui o caractere que ela devolve à variável *ch* para depois imprimi-lo com **printf()**.

```
main()
{
   char ch;
   printf("Digite algum caractere: ");
   ch=getche();
   printf("\n A tecla que voce pressionou e' %c.",ch);
}
```

Eis a execução:

```
C>Ecoa
Digite algum caractere: a
A tecla que voce pressionou e' a.
```

A função **getch()** lê o caractere do teclado e não permite que seja impresso na tela. Como **getche()**, esta função não aceita argumentos e devolve o caractere lido para a função que a chamou.

O programa seguinte chama a função **getch()** e atribui o caractere que ela devolve à variável *ch* para depois imprimir o caractere e o seu sucessor na tabela ASCII com **printf()**.

```
main()
{
   char ch;
   printf("Digite algum caractere: ");
   ch=getch();
   printf("\n A tecla que voce pressionou e' %c",ch);
   printf(" e a sua sucessora ASCII e' %c.",ch+1);
}
```

Eis a execução:

```
C>necoa
Digite algum caractere:
A tecla que voce pressionou e' a e a sua sucessora ASCII e' b.
```

A biblioteca padrão provê outras funções para a leitura e escrita de um caractere por vez.

A FUNÇÃO getchar()

A função **getchar()** está definida no arquivo *stdio.h*, que acompanha seu compilador. Obtém o próximo caractere da entrada cada vez que é chamada, só terminando a leitura quando é pressionada a tecla **[enter]**, e retorna o caractere como seu valor. A função **getchar()** não aceita argumentos. Isto é, após

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

a variável **c** contém o próximo caractere da entrada. Caso encontre a indicação do fim de arquivo **getchar()** retorna -1.

A FUNÇÃO putchar()

A função **putchar()** é o complemento de **getchar()** e também está definida no arquivo *stdio.h*. A função **putchar()** aceita um argumento cujo valor será impresso. Os comandos a seguir mostram como ler um caractere da entrada, atribuir seu valor à variável **c** e imprimir o conteúdo da variável **c** na saída padrão.

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

Um argumento de uma função pode ser outra função. Por exemplo, as linhas anteriores podem ser escritas como:

```
putchar(getchar());
```


OPERADORES ARITMÉTICOS

C é uma linguagem rica em operadores, em torno de 40. Alguns são mais usados que outros, como é o caso dos operadores aritméticos que executam operações aritméticas.

C oferece 6 operadores aritméticos binários (operam sobre dois operandos) e um operador aritmético unário (opera sobre um operando). São eles:

Binários

=	Atribuição
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (devolve o resto da divisão inteira)

Unário

–	Menos unário

OPERADOR DE ATRIBUIÇÃO: =

Em C, o sinal de igual não tem a interpretação dada em matemática. Representa a atribuição da expressão à direita ao nome da variável à esquerda. Por exemplo:

```
num=2000;
```

atribui o valor 2000 à variável *num*. A ação é executada da direita para a esquerda deste operador.

Observe que:

```
2000=num
```

é uma igualdade válida em matemática mas que não tem sentido em C pois não podemos atribuir um valor a uma constante.

C aceita várias atribuições numa mesma instrução:

```
laranjas=cenouras=abacates=80;
```

OPERADORES : + - / *

Estes operadores representam as operações aritméticas básicas de soma, subtração, divisão e multiplicação.

A seguir está um programa que usa vários operadores aritméticos e converte temperatura Fahrenheit em seus correspondentes graus Celsius.

```
main()
{
   int ftemp,ctemp;
   printf("Digite temperatura em graus Fahrenheit: ");
   scanf("%d",&ftemp);
   ctemp=(ftemp-32) * 5/9;
   printf("Temperatura em graus Celsius e' %d",ctemp);
}
```

Eis um exemplo:

```
C>ftemp
Digite temperatura em graus Fahrenheit: 32
Temperatura em graus Celsius e' 0.
C>ftemp
Digite temperatura em graus Fahrenheit: 70
Temperatura em graus Celsius e' 21.
```

Vamos analisar a instrução:

```
ctemp=(ftemp-32) * 5/9;
```

Note que colocamos parênteses em **ftemp-32**. Se você lembra um pouco de álgebra, a razão estará clara. Nós queremos que 32 seja subtraído de ftemp antes de multiplicarmos por 5 e dividirmos por 9. A multiplicação e a divisão são feitas antes da soma ou subtração.

O programa a seguir usa um algoritmo interessante para adivinhar a soma de 5 números. O usuário digita um número qualquer e o computador informa o resultado da soma dos 5 números dos quais o primeiro o usuário já forneceu. O usuário digita o segundo número e o computador mostra o terceiro. O usuário digita o quarto número e o computador mostra o quinto.

Eis a listagem.

```
main()
{
   int x,r;
   printf("Digite um numero de ate 4 algarismos\n");
   scanf("%d",&x);
   r = 19998 + x;
   printf("O resultado da nossa conta sera: %d\n",r);
   printf("Digite o segundo numero (4 algarismos)\n");
   scanf("%d",&x);
   printf("O meu numero e': %d\n",9999-x);
   printf("Digite o quarto numero (4 algarismos)\n");
   scanf("%d",&x);
   printf("O meu numero e': %d\n",9999-x);
}
```

Eis um exemplo de sua execução:

```
Digite um numero de ate 4 algarismos
198
O resultado da nossa conta sera: 20196
Digite o segundo numero (4 algarismos)
1234
O meu numero e': 8765
Digite o quarto numero (4 algarismos)
2233
O meu numero e': 7766
```

OPERADOR MENOS UNÁRIO: -

O operador menos unário é usado somente para indicar a troca do sinal algébrico do valor. Pode também ser pensado como o operador que multiplica seu operando por -1. Por exemplo:

```
num = -8;
num1 = -num;
```

Depois destas duas instruções, o conteúdo de **num1** será 8.

OPERADOR MÓDULO: %

O operador módulo aceita somente operandos inteiros. Resulta o resto da divisão do inteiro à sua esquerda pelo inteiro à sua direita.

Por exemplo, 17%5 tem o valor 2 pois quando dividimos 17 por 5 teremos resto 2.

É também possível incluir expressões envolvendo operadores aritméticos (e outros operadores) diretamente em **printf()**.

Na verdade, uma expressão completa pode ser usada em quase todos os lugares onde uma variável pode ser usada. Como exemplo, vamos modificar o programa anterior usando uma expressão em **printf()** ao invés da variável.

```
main()
{
   int ftemp;
   printf("Digite temperatura em graus Fahrenheit: ");
   scanf("%d",&ftemp);
   printf("Temper. em graus Celsius e' %d",(ftemp-32)* 5/9);
}
```

OPERADORES DE INCREMENTO (++) E DECREMENTO (—)

Se comparar um programa C com um programa similar escrito em outra linguagem, você perceberá, imediatamente, que o programa C é menor. Uma das razões para que isto seja possível é que C tem vários operadores que podem comprimir comandos de programas.

Considere os operadores abaixo que não são comuns em outras linguagens:

++	incrementa de 1 seu operando
--	decrementa de 1 seu operando

O operador de incremento (++) incrementa de um seu operando. Este operador trabalha de dois modos. O primeiro modo é chamado pré-fixado e o operador aparece antes do nome da variável. O segundo é o modo pós-fixado em que o operador aparece seguindo o nome da variável.

Em ambos os casos, a variável é incrementada. Porém quando **++n** é usado numa instrução, **n** é incrementada antes de seu valor ser usado, e quando **n++** estiver numa instrução, **n** é incrementada depois de seu valor ser usado.

Exemplos:

Se as seguintes linhas de programa forem executadas:

```
n = 5;
x = n++;
printf("x=%d n=%d",x,n);
```

a saída será:

```
x=5 n=6
```

Na primeira linha atribuímos o valor 5 a **n**, na segunda linha o valor de **n** é atribuído a **x** e depois **n** é incrementada de 1, tornando seu valor 6.

Agora vamos ver um outro exemplo:

```
n = 5;
x = ++n;
printf("x=%d n=%d",x,n);
```

a saída será:

```
x=6 n=6
```

Na primeira linha atribuímos o valor 5 a **n**, na segunda linha **n** é incrementada de um e depois seu valor é atribuído a **x**.

Vamos analisar as duas expressões seguintes:

`k = 3 * n++;` ——→ **PRIMEIRO** n é multiplicado por 3
DEPOIS o resultado é atribuído a k
FINALMENTE n é incrementada de 1

`k = 3 * ++n;` ——→ **PRIMEIRO** n é incrementada de 1
DEPOIS n é multiplicado por 3
FINALMENTE o resultado é atribuído a k

Quando um destes operadores aparece sozinho numa instrução, como em:

```
num++;
```

não faz diferença o uso do modo pré-fixado ou pós-fixado.

Outros exemplos:

```
main()
{
   int num=0;

   printf("%d e' um belo numero\n",num);
   printf("%d e' um belo numero\n",num++);
   printf("%d e' um belo numero\n",num);
}
```

a saída será:

0 e' um belo numero
0 e' um belo numero
1 e' um belo numero

De fato, **num++** é exatamente igual a

```
num = num + 1;
```

entretanto num++ é uma declaração bem mais compacta.


```
main()
{
   int num=0;

   printf("%d e' um belo numero\n",num);
   printf("%d e' um belo numero\n",++num);
   printf("%d e' um belo numero\n",num);
}
```

a saída será:

0 e' um belo numero
1 e' um belo numero
1 e' um belo numero

Toda a análise anterior vale para o operador de decremento (--).

Exemplos:

```
main()
{
   int num=0;

   printf("%d e' um belo numero\n",num);
   printf("%d e' um belo numero\n",num--);
   printf("%d e' um belo numero\n",num);
}
```

a saída será:

0 e' um belo numero
0 e' um belo numero
-1 e' um belo numero

```
main()
{
   int num=0;

   printf("%d e' um belo numero\n",num);
   printf("%d e' um belo numero\n",--num);
   printf("%d e' um belo numero\n",num);
}
```

a saída será:

```
0 e' um belo numero
-1 e' um belo numero
-1 e' um belo numero
```

PRECEDÊNCIA

Operadores de incremento e decremento têm precedência maior que a dos aritméticos. Isto é,

```
a*b++
```

é equivalente a

```
(a)*(b++)
```

Operadores de incremento e decremento só podem ser usados com variáveis e não com expressões ou constantes. Portanto,

```
ERRADO:     (a*b)++;     5++;
```

Não confunda a precedência destes dois operadores com a ordem de avaliação. Analise estas instruções:

```
a = 2;
b = 5;
n = (a + b++) * 3;
```

Substituindo os valores, teremos:

```
n = (2 + 5) * 3 = 7 * 3 = 21
```

Somente depois da expressão ser avaliada, **b** é incrementada para 6. A precedência informa que ++ está afetando a variável **b** e qual valor de **b** será usado para avaliar a expressão, e a definição do operador de incremento determina quando o valor de **b** será mudado.

printf() ENGANANDO VOCÊ

Você pode ser iludido ao tentar imprimir uma mesma variável várias vezes usando o operador de incremento ou decremento. Por exemplo:

```
n=5;
printf("%d %d %d\n",n,n+1,n++);
```

Isto parece razoável e você pode pensar que a impressão será:

```
5 6 5
```

De fato, em vários sistemas estas linhas funcionarão de acordo, mas não em todos.

O problema é quando **printf()** toma os valores a serem impressos. Ela pode avaliar o último argumento primeiro, e isto provocaria a seguinte impressão:

```
6 7 5
```

o que pareceria loucura.

A mesma situação é encontrada em expressões de atribuição com a mesma variável aparecendo mais de uma vez. A maneira de avaliar é imprevisível.

Testes elaborados com o compilador TURBO C versão 2.0 da Borland apresentaram os seguintes resultados:

```
main()
{
   int i=3;
   int n;
   n=i*(i+1)+(++i);
   printf("\n n = %d",n);
}
```

```
n = 24
```

```
main()
{
   int i=3;
   int n;
   n=i*(i+1)+(i++);
   printf("\n n = %d",n);
}
```

n = 15

```
main()
{
   int i=3;
   int n;
   printf("\n n = %d",n=i*(i+1)+(++i));
}
```

n = 16

```
main()
{
   int i=3;
   int n;
   printf("\n n = %d",n=i*(i+1)+(i++));
}
```

n = 15

```
main()
{
   int i=3;
   printf("\n %d %d %d ",i=i+1,i=i+1,i=i+1);
}
```

6 5 4

OPERADORES ARITMÉTICOS DE ATRIBUIÇÃO
+=, -=, *=, /=, %=

Cada um destes operadores é usado com um nome de variável à sua esquerda e uma expressão à sua direita. A operação consiste em atribuir um novo valor à variável que dependerá do operador e da expressão à direita.

Se **x** é uma variável, **exp** uma expressão e **op** um operador aritmético (+, -, *, / ou %), então

x op= exp;

equivale a

x = (x) op (exp);

Exemplos:

i += 2;	equivale a	i = i + 2;
x *= y+1;	equivale a	x = x * (y+1);
t /= 2.5;	equivale a	t = t/2.5;
p %= 5;	equivale a	p = p%5;
d -= 3;	equivale a	d = d-3;

As expressões com estes operadores são mais compactas e normalmente produzem um código de máquina mais eficiente.

Exemplo:

```
main()
{
   int total=0;
   int cont=10;

   printf("Total=%d\n",total);
   total+=cont;
   printf("Total=%d\n",total);
   total+=cont;
   printf("Total=%d\n",total);
}
```

A saída será:

```
Total=0
Total=10
Total=20
```

O símbolo = pode ser combinado com outros operadores como os lógicos ou os operadores bit a bit dos quais falaremos no segundo volume.

OPERADORES RELACIONAIS

Operadores relacionais são usados para fazer comparações. São eles:

>	maior
>=	maior ou igual
<	menor
<=	menor ou igual
==	igualdade
!=	diferente

No próximo capítulo falaremos sobre comandos de decisão e laços. Estas construções requerem que o programa pergunte sobre relações entre variáveis. Operadores relacionais são o vocabulário que o programa usa para fazer estas perguntas.

Em C não existe um tipo de variável chamada "booleana", isto é, que assuma um valor verdadeiro ou falso. O valor zero (0) é considerado falso e qualquer valor diferente de 0 é considerado verdadeiro e é representado pelo inteiro um (1). Portanto, em C, qualquer valor é ou verdadeiro ou falso.

O programa a seguir mostra expressões booleanas como argumento da função **printf()**:

```
main()
{
   int verdad,falso;

   verdad = (15 < 20);
   falso = (15==20);
   printf("Verdadeiro= %d, falso= %d\n",verdad,falso);
}
```


A saída será:

```
Verdadeiro= 1, falso= 0
```

Note que o operador relacional "igual a" é representado por dois sinais de igual. Um erro comum é o de usar um único sinal de igual como operador relacional. O compilador não o avisará que este é um erro, por quê? Na verdade, como toda expressão C tem um valor verdadeiro ou falso, este não é um erro de programa e sim um erro de lógica do programador.

```
main ()
{
   int veloc;

   veloc = 75;
   printf("\nA velocidade e' igual a 55?%d",veloc==55);
   veloc = 55;
   printf("\nA velocidade e' igual a 55?%d",veloc==55);
}
```

Saída:

```
A velocidade e' igual a 55? 0
A velocidade e' igual a 55? 1
```

PRECEDÊNCIA

Os operadores aritméticos têm maior precedência que a dos relacionais. Isto significa que serão avaliados antes.

```
main ()
{
   printf ("A resposta e' %d",4+1<4);
}
```

a saída será:

```
A resposta e' 0
```

```
main ()
{
   printf ("A resposta e' %d",1<2+4);
}
```

A saída será:

```
A resposta e' 1
```

COMENTÁRIOS

Comentários podem ser colocados em qualquer lugar de seu programa. Mostraremos o programa que calcula a idade com os devidos comentários:

```
/* idade.c */
/* programa para calcular a sua idade em dias */
main()
{
   float anos,dias;                          /*declara variaveis*/

   printf("Digite sua idade em anos: ");
   scanf("%f",&anos);                          /*le idade em anos*/
   dias = anos*365;                         /*calcula idade em dias*/
                                          /* impressao do resultado */
   printf("Sua idade em dias e' %f.\n",dias);
}
```

Comentários começam com dois caracteres, chamados símbolos de comentários, barra-asterisco (/*) e terminam por asterisco-barra (*/). Como C ignora espaços, os comentários podem ser escritos em várias linhas:


```
/*
Aqui esta
um exemplo de
comentario em
multiplas linhas
*/
```

Asteriscos dentro de comentários podem ser colocados livremente:

```
/*****************
* Aqui esta
* um exemplo de
* comentario em
* multiplas linhas
*****************/
```

Não são permitidos os símbolos de /* ou */ no interior de um comentário.

```
/* estou escrevendo /* um comentario ilegal */
```

REVISÃO

1. A função **scanf()** é usada para ler informações do teclado. Os seus argumentos consistem em uma expressão de controle, contendo códigos de formatação iniciados pelo caractere %, e tantos argumentos quantos forem os códigos de formatação na expressão de controle.
2. Os argumentos de **scanf()** devem ser endereços, isto é, o nome da variável precedido do operador de endereço **&**.
3. O operador unário de endereço resulta o endereço da variável operando.
4. As funções **getche()**, **getch()** e **getchar()** retornam o caractere lido do teclado.
5. A função **putchar()** imprime um caractere na tela.
6. Os operadores C são numerosos e podem trazer alguma confusão. Por exemplo, o símbolo % executa coisas diferentes em contextos diferentes.

7. O operador de atribuição tem uma interpretação diferente da matemática. Representa a atribuição da expressão à sua direita à variável à sua esquerda.
8. Em C não existe o operador aritmético + unário.
9. Os operadores de incremento ++ e decremento -- podem ser usados pré-fixados ou pós-fixados e incrementam ou decrementam a variável operando de 1. A instrução **i++;** tem o valor de i, enquanto que a expressão **++i;** tem o valor de i+1.
10. Os operadores de incremento e decremento têm precedência maior que a dos aritméticos.
11. Os operadores aritméticos de atribuição alteram a variável à esquerda pela expressão à direita, usando o operador indicado.
12. Os operadores relacionais comparam a expressão à esquerda com a expressão à direita. Se a expressão toda for verdadeira, ela assumirá o valor 1, caso contrário, o valor 0.
13. Zero em C é avaliado como falso e qualquer outra coisa é avaliada como verdadeiro.
14. Os operadores aritméticos têm precedência maior que a dos relacionais.
15. Comentários são escritos entre /* e */ e auxiliam o leitor a usar e entender o programa.

EXERCÍCIOS

1. Qual é o erro deste programa?

```
main()
{
   int i;
   scanf("%3d",&i);
   printf("%3d",i);
}
```


2. Este programa tem um erro de lógica. Qual é?

```
main()
{
   int a,b,c;
   printf("Digite 3 numeros:\n");
   scanf("%d %d %d",a,b,c);
   printf("\n%d %d %d",a,b,c);
}
```

3. A função **scanf()** retorna o número de leituras feitas com sucesso. Considere o seguinte programa:

```
main()
{
   int i;j,k;
   printf("%d\n",scanf("%d %d %d",&i,&j,&k));
}
```

Execute-o digitando os seguintes valores:

a)	1	2	3
b)	1	2	a
c)	a	3	4
d)	3	4	2.1
e)	^Z	5	1
f)	1	2	^Z
g)	3.2	1	2

Verifique os possíveis inteiros retornados por **scanf()**. Observe que o caractere **^Z** é obtido pressionando-se, ao mesmo tempo, as teclas **Ctrl** e **Z**.

4. O programa seguinte tem um erro em tempo de execução. Verifique.

```
main()
{
   int a,b=0;

   a=5/b;
}
```

5. O programa seguinte tem vários erros em tempo de execução. Verifique.

```
Main()
{
   int a=1; b=2, c=3;
   printf("Os numeros sao: %d %d %d\n,a,b,c,d);
}
```

6. Quais dos seguintes operadores são aritméticos?

   a) +
   b) &
   c) %
   d) <

7. Assuma que todas as variáveis são do tipo **int**. Encontre o valor de cada uma delas e escreva um programa que as imprima para verificar os resultados:

   a) x=(2+1)*6;
   b) y=(5+1)/2*3;
   c) i=j=(2+3)/4;
   d) a=3+2*(b=7/2);
   e) c=5+10%4/2;

8. Reescreva a seguinte instrução usando operador de incremento:

   número = número + 1;

9. Como será interpretada a expressão **x+++y**?

   a) x++ + y
   b) x + ++y

   Escreva um pequeno programa e verifique a interpretação dada em seu compilador.

10. Reescreva a seguinte instrução usando operador aritmético de atribuição:

    laranja=laranja + x;

11. Considere o código:

    int x=1,y=2,z=3;

    x+=y+=z+=7;

    Quais serão os valores das variáveis x,y e z? Escreva um programa para checar a sua resposta.

12. Operadores relacionais são usados para:

    a) combinar valores;
    b) comparar valores;
    c) distinguir diferentes tipos de variáveis;
    d) trocar variáveis por valores lógicos.

13. Verdadeiro ou Falso:

    a) 1 > 2
    b) 'a' < 'b'
    c) 3==2
    d) '1'=='1'
    e) 3 >= 2
    f) 'j' != 'j'

14. Qual será o valor de k ?

    j = 3;
    k = j == 3;

15. A precedência dos operadores determina qual é o operador:

    a) mais importante;
    b) usado primeiro;
    c) mais adequado;
    d) que opera em números maiores.

16. O comentário seguinte é correto ?

    /* Este é um comentario
    /* que se estende em
    /* várias linhas
    */

17. Modifique o programa *idade.c* para que imprima a idade em minutos ao invés de em dias.

18. Escreva um programa que solicite 3 números em ponto flutuante e imprima a média aritmética.

MAKRON Books

CAPÍTULO 3

LAÇOS

- *O Laço for*
- *O Laço while*
- *O Laço do-while*
- *Os Comandos break E continue*
- *O Comando goto*

Em C existem 3 estruturas principais de laços: o laço **for**, o laço **while** e o laço **do-while**.

O LAÇO for

O laço **for** engloba 3 expressões numa única, e é útil principalmente quando queremos repetir algo um número fixo de vezes.

O exemplo seguinte imprime os números de 0 a 9 utilizando um laço **for** na sua forma mais simples:

```
/* lacofor.c */
/* imprime numeros de 0 a 9 */
main()
   {
   int conta;
   for(conta=0;conta<10;conta++)
   printf("conta=%d\n",conta);
}
```

Como saída teremos:

```
C>lacofor
conta=0
conta=1
conta=2
conta=3
conta=4
conta=5
conta=6
conta=7
conta=8
conta=9
```

O programa executa 10 vezes a instrução **printf()**. A função **printf()** imprime a frase "conta=" seguida do valor contido na variável *conta*.

FORMA GERAL DO LAÇO for:

```
for(inicialização;teste;incremento)
    instrução;
```

A ESTRUTURA DO LAÇO for

Os parênteses seguindo a palavra-chave **for** contêm três expressões separadas por pontos-e-vírgulas.

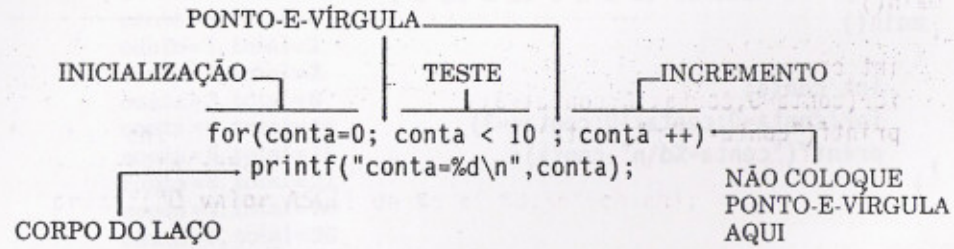
A expressão como um todo é chamada de "expressão do laço" e é dividida em: "expressão de inicialização", "expressão de teste" e "expressão de incremento".

As três expressões podem ser compostas por quaisquer instruções válidas em C.

Em sua forma mais simples, a *inicialização* é uma instrução de atribuição (conta=0) e é sempre executada uma única vez antes do laço ser iniciado.

O *teste* é uma instrução de condição que controla o laço (conta<10). Esta expressão é avaliada como verdadeira ou falsa toda vez que o laço for iniciado ou reiniciado. Se verdadeira (diferente de zero), o corpo do laço é executado. Quando a expressão tornar-se falsa (igual a zero), o laço é terminado e o controle passa para a instrução seguinte ao laço.

A expressão de *incremento* define a maneira como a variável de controle do laço será alterada cada vez que o laço é repetido (**conta++**). Esta instrução é executada, toda vez, imediatamente após a execução do corpo do laço.



NUM LAÇO for NÃO COLOQUE PONTO-E-VÍRGULA ENTRE OS PARÊNTESES QUE ENVOLVEM A EXPRESSÃO DO LAÇO E O CORPO DO LAÇO.

Vamos modificar o programa *lacofor.c* para que imprima os números de 9 a 0.

```
/* lacofor1.c */
/* imprime numeros de 9 a 0 */
main()
{
   int conta;
   for(conta=9;conta>=0;conta--)
   printf("conta=%d\n",conta);
}
```

Como saída teremos:

```
C>lacofor1
conta=9
conta=8
conta=7
conta=6
conta=5
conta=4
conta=3
conta=2
conta=1
conta=0
```

Agora de 0 a 9 de 3 em 3:

```
/* lacofor2.c */
/* imprime numeros de 9 a 0 de 3 em 3*/
main()
{
   int conta;
   for(conta=0;conta<10;conta+=3)
   printf("conta=%d\n",conta);
}
```


Como saída teremos:

```
C>lacofor2
conta=0
conta=3
conta=6
conta=9
```

FLEXIBILIDADE DO LAÇO for

Nos exemplos anteriores usamos a forma mais simples para as três expressões do laço **for**. Isto é, a primeira expressão para inicializar a variável *conta*, a segunda para expressar um limite para *conta* e a terceira para incrementar ou decrementar *conta*. Entretanto elas não são restritas apenas a estas formas. C permite muitas outras possibilidades que mostraremos nos próximos exemplos.

1. Qualquer uma das expressões de um laço **for** pode conter várias instruções separadas por vírgulas. A vírgula é na verdade um operador C que significa "faça isto e isto". Um par de expressões separadas por vírgula é avaliado da esquerda para a direita.

```
/* lacofor3.c*/
/* imprime os numeros de 0 a 98 em incremento de 2 */
main()
{
   int x,y;
   for( x = 0 , y = 0; x+y < 100 ; x =x+1,y = y+1)
   printf("%d ",x+y);
}
```

2. Podemos usar caracteres em vez de inteiros:

```
main()
{
   char ch;
   for(ch='a';ch<='z';ch++)
   printf("O valor ASCII de %c e' %d.\n",ch,ch);
}
```

Este programa imprimirá as letras minúsculas do alfabeto e seus respectivos códigos decimais da tabela ASCII.

3. Podemos usar chamadas a funções em qualquer uma das expressões do laço:

```
main()
{
   char ch;
   for(ch=getch();ch!='X';ch=getch())
   printf("%c",ch+1);
}
```

Este programa solicita a entrada de um texto. Lê caractere a caractere e imprime o caractere seguinte a partir do código ASCII.

4. Qualquer uma das três partes de um laço **for** pode ser omitida, embora os pontos-e-vírgulas devam permanecer. Se a expressão de inicialização ou a de incremento forem omitidas, elas serão simplesmente desconsideradas. Se a condição de teste não está presente é considerada permanentemente verdadeira.

```
main()
{
   char ch;
   for(;(ch=getch())!='X';)
      printf("%c",ch+1);
}

main()
{
   for (;;) printf("Laco infinito\n");
}
```

5. O corpo do laço pode ser vazio, entretanto o ponto-e-vírgula permanece.

```
main()
{
   char c;
   for(;(c=getch())!='X';printf("%c",c+1))
      ;
}

main()
{
   char c;
   for(c=0;c<=100;c++)
      ;
}
```


INSTRUÇÕES MÚLTIPLAS NO CORPO DE UM LAÇO for

Os exemplos anteriores usam somente uma instrução no corpo do laço **for**. Duas ou mais instruções podem ser colocadas se estiverem entre chaves.

```
/*lacofor4.c*/
/*imprime os numeros de 0 a 9 e totais */
main()
{
   int conta, total;
   for(conta=0, total=0;conta<10;conta++) {
      total+=conta;
      printf("conta=%d, total=%d\n",conta,total);
   }
}
```

A saída do programa será:

```
C>lacofor4
conta=0, total=0
conta=1, total=1
conta=2, total=3
conta=3, total=6
conta=4, total=10
conta=5, total=15
conta=6, total=21
conta=7, total=28
conta=8, total=36
conta=9, total=45
```

VOCÊ DEVE USAR CHAVES SE O CORPO DO LAÇO CONTIVER MAIS DE UMA INSTRUÇÃO.

Duas coisas devem ser lembradas sobre estas múltiplas instruções no corpo do laço **for**: a primeira é a chave de abertura, as instruções e a chave de fechamento, chamadas de bloco, que são tratados como uma simples instrução C. A segunda é que cada instrução do bloco é por sua vez uma instrução C e deve ser terminada por ponto-e-vírgula.

LAÇOS for ANINHADOS

Quando um laço está dentro de outro laço, dizemos que o laço interior está aninhado. Para mostrar esta estrutura preparamos um programa que imprime tabuada.

```
/* tabuada.c */
/* gera tabuada de 1 a 9 */
main()
{
   int i,j,k;
   printf("\n");
   for(k=0;k<=1;k++) {
      printf("\n");
      for(i=1;i<5;i++)
         printf("TABUADA DO %3d    ",i+4*k+1);
      printf("\n");
      for(i=1;i<=9;i++) {
         for(j=2+4*k;j<=5+4*k;j++)
            printf("%3d x%3d = %3d    ",j,i,j*i);
         printf("\n");
      }
   }
}
```

A saída deste programa cabe numa tela e você verá a seguinte impressão:

```
C>tabuada
```



```
TABUADA DO 2    TABUADA DO 3    TABUADA DO 4    TABUADA DO 5

2 x 1 =  2      3 x 1 =  3      4 x 1 =  4      5 x 1 =  5
2 x 2 =  4      3 x 2 =  6      4 x 2 =  8      5 x 2 = 10
2 x 3 =  6      3 x 3 =  9      4 x 3 = 12      5 x 3 = 15
2 x 4 =  8      3 x 4 = 12      4 x 4 = 16      5 x 4 = 20
2 x 5 = 10      3 x 5 = 15      4 x 5 = 20      5 x 5 = 25
2 x 6 = 12      3 x 6 = 18      4 x 6 = 24      5 x 6 = 30
2 x 7 = 14      3 x 7 = 21      4 x 7 = 28      5 x 7 = 35
2 x 8 = 16      3 x 8 = 24      4 x 8 = 32      5 x 8 = 40
2 x 9 = 18      3 x 9 = 27      4 x 9 = 36      5 x 9 = 45

TABUADA DO 6    TABUADA DO 7    TABUADA DO 8    TABUADA DO 9

6 x 1 =  6      7 x 1 =  7      8 x 1 =  8      9 x 1 =  9
6 x 2 = 12      7 x 2 = 14      8 x 2 = 16      9 x 2 = 18
6 x 3 = 18      7 x 3 = 21      8 x 3 = 24      9 x 3 = 27
6 x 4 = 24      7 x 4 = 28      8 x 4 = 32      9 x 4 = 36
6 x 5 = 30      7 x 5 = 35      8 x 5 = 40      9 x 5 = 45
6 x 6 = 36      7 x 6 = 42      8 x 6 = 48      9 x 6 = 54
6 x 7 = 42      7 x 7 = 49      8 x 7 = 56      9 x 7 = 63
6 x 8 = 48      7 x 8 = 56      8 x 8 = 64      9 x 8 = 72
6 x 9 = 54      7 x 9 = 63      8 x 9 = 72      9 x 9 = 81
```

O laço **for** mais externo (o da variável **k**) é executado duas vezes. A primeira para imprimir o primeiro bloco de tabuadas (de 2 a 5) e a segunda para imprimir o segundo bloco (de 6 a 9). O segundo laço **for** imprime os títulos. Os dois laços mais internos imprimem a tabuada propriamente dita.

Outro exemplo de laços aninhados é o programa *invtela.c* seguinte.

```
/* invtela.c */
/* preenche uma parte da tela com caracteres graficos*/
main()
{
   int linha,coluna;
   printf("\n");
   for(linha=1;linha<=24;linha++) {
      for(coluna=1;coluna<40;coluna++)
         printf("\xDB");
      printf("\n");
   }
}
```

Saída do programa:

Note que '\xDB' é um caractere e um caractere pode fazer parte da expressão de controle de **printf()**.

Agora que já aprendemos a trabalhar com o laço **for** vamos escrever um programa mais elaborado que imprimirá um cartão de natal. O programa usa todos os conceitos vistos até este ponto e vale a pena você tentar entendê-lo.

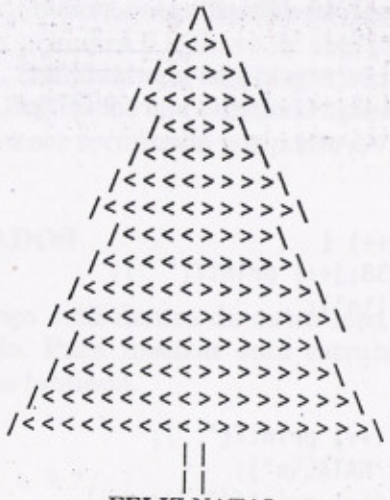
```
/* natal1.c */
/* imprime cartao de natal na tela */
main()
{
   char se,sd;
   int i,j,k;

   printf("\nSinal interno direito: "); sd=getche();
```



```
   printf("\nSinal interno esquerdo: "); se=getche();
   printf("\n\n");
   for(i=0;i<4;i++)
      for(k=1;k<5;k++) {
         for(j=1;j<=40-(2*i+k);j++) printf(" ");
         printf("/");
         for(j=1;j<(2*i+k);j++) printf("%c",se);
         for(j=1;j<(2*i+k);j++) printf("%c",sd);
         printf("\\\n");
      }

   for(i=0;i<2;i++) {
      for(j=0;j<38;j++) printf(" ");
      printf("| |\n");
   }

   printf("\n");
   for(i=0;i<35;i++) printf(" ");
   printf("FELIZ NATAL\n");
   for(i=0;i<31;i++) printf(" ");
   printf("E UM PROSPERO 1990!\n");
   getch();
}
```

Eis a execução:

```
C>natal1
```



```
FELIZ NATAL
E UM PROSPERO 1990!
```

O LAÇO while

A segunda estrutura de laço válida em C é o laço **while**. À primeira vista esta estrutura parece simples comparada a do laço **for**; ela utiliza os mesmos elementos, mas eles são distribuídos de maneira diferente no programa.

FORMA GERAL DO LAÇO while

```
while(expressão de teste)
instrução;
```

Se a expressão de teste for verdadeira (diferente de zero), o corpo do laço **while** é executado uma vez e a expressão de teste é avaliada novamente. Este ciclo de teste e execução é repetido até que a expressão

de teste se torne falsa (igual a zero), então o laço termina e o controle do programa passa para a linha seguinte ao laço.

O corpo de um **while** pode ter uma única instrução terminada por ponto-e-vírgula, várias instruções entre chaves ou ainda nenhuma instrução mantendo o ponto-e-vírgula. Em geral um laço **while** pode substituir um laço **for** da seguinte forma dentro de um programa:

```
inicializacao;
while(teste) {
      .
      .
   incremento;
      .
      .
}
```

Como exemplo, tomemos o programa *lacofor1.c* para reescrevê-lo usando um laço **while** ao invés do laço **for** original.

```
/* lacowh1.c*/
/*imprime os numeros de 0 a 9 e totais */

main()
{
   int conta=0; /* inicializa a var. conta */
   int total=0; /* inicializa a var. total */

   while(conta<10) { /* executa laco ate' conta ser 10 */
      total+=conta; /* mesmo que total=total+conta */
      printf("conta=%d, total=%d\n",conta++,total);
   }
}
```

A execução do programa é exatamente a mesma do *lacofor1.c*.

Em situações em que o número de interações é conhecido de antemão, como no programa *lacowh1.c*, o laço **while** é menos apropriado. Neste caso o laço **for** é a escolha mais natural.

Quando o laço **while** é mais apropriado? O laço **while** é apropriado em situações em que o laço pode ser terminado inesperadamente por condições desenvolvidas dentro do laço. Como exemplo, considere o seguinte programa:

```
/* contchar.c */
/* conta caracteres de uma frase */

main()
{
   int cont=0;

   printf("Digite uma frase:\n");
   while(getche()!='\r')
      cont++;
   printf("\nO numero de caracteres e' %d",cont);
}
```

Eis uma execução simples:

```
C>contchar
Digite uma frase:
Enquanto se vive e' necessario aprender a viver.
O numero de caracteres e' 47
```

Este programa solicita a você que digite uma frase. Cada caractere digitado é acumulado na variável *cont* até que você tecle **<RETURN>**, e então é impresso o total de caracteres da frase.

O LAÇO while É MAIS APROPRIADO QUE O LAÇO for QUANDO A CONDIÇÃO DE TÉRMINO DO LAÇO OCORRER INESPERADAMENTE.

LAÇOS while ANINHADOS

Laços aninhados permitem gerar programas interessantes. Por exemplo, considere o seguinte programa que testa sua capacidade de adivinhação.

```
/* adiv.c */
/* testa a sua capacidade de adivinhar uma letra */
```



```
main()
{
   char ch,c;
   char ch1='s';
   int tentativas;

   while(ch1=='s') {
      ch= rand()%26 +'a';
      tentativas=1;
      printf("\nDigite uma letra de 'a' a 'z':\n");
      while((c=getch())!=ch) {
         printf("%c e' incorreto. Tente novamente.\n\n",c);
         tentativas++;
      }
      printf("\n%c e' correto",c);
      printf("\nVoce acertou em %d tentativas",tentativas);
      printf("\nQuer jogar novamente ? (s/n): ");
      ch1=getche();
   }
}
```

Eis uma execução simples:

```
C>adiv
Digite uma letra de 'a' a 'z':
w e' incorreto. Tente novamente.
k e' incorreto. Tente novamente.
a e' incorreto. Tente novamente.
i e' correto
Voce acertou em 4 tentativas
Quer jogar novamente? (s/n):n
```

O aspecto mais importante no programa *adiv.c* é a expressão

```
while ( ( c = getch() ) != ch)
```

Já sabemos que a função **getch()** retorna um valor, agora este valor é atribuído à variável caractere **c** e, finalmente, comparado com o conteúdo da variável **ch**.

Note que colocamos parênteses extras envolvendo a expressão de atribuição **(c=getch())**. Estes parênteses são realmente necessários, pois a precedência de **!=** é maior que a de **=**; isto significa que na falta de

parênteses, o teste relacional != seria feito antes da atribuição ou a expressão seria equivalente a

```
while(c=(getch()!=ch))
```

e **c** teria um valor verdadeiro ou falso (0 ou 1).

Na instrução

```
ch = rand() % 26 + 'a';
```

foi usada a função **rand()** de biblioteca C que devolve um inteiro aleatório entre 0 e 32767. Esta função será desenvolvida mais adiante neste livro.

A expressão **rand() % 26** resulta o resto da divisão de **rand()** por 26, isto é, um número entre 0 e 25. A este número é somado o caractere 'a' para gerar letras entre 'a' e 'z'.

O fatorial de um número inteiro **N** é o produto de todos os inteiros entre 1 e N. Por exemplo, 5 fatorial é 1x2x3x4x5 ou 120. O fatorial de 0 é 1 por definição.

O programa seguinte calcula o fatorial de um número.

```
/* fator.c */
/* encontra o fatorial de um numero */

main()
{
   int num;
   long resposta;

   while(1) {
      printf("\n Digite o numero: ");
      scanf("%d",&num);
      resposta=1;
      while(num > 1)
         resposta *= num--;

      printf("O fatorial e': %ld\n", resposta);
   }
}
```


Eis uma execução:

```
C>fator
Digite o numero: 5
O fatorial e': 120
```

O laço **while** mais externo é um laço infinito. Você deverá pressionar as teclas **[Ctrl][Break]** para sair.

O LAÇO do-while

O último laço em C é o laço **do-while** que cria um ciclo repetido até que a expressão de teste seja falsa (zero). Este laço é bastante similar ao laço **while**.

A diferença entre os dois laços é que no laço **do-while** o teste de condição é avaliado depois do laço ser executado. Assim, o laço **do-while** é sempre executado pelo menos uma vez.

Forma geral:

```
do {
      instrução;
} while(expressão de teste); ——— PONTO-E-VÍRGULA AQUI
```

Embora as chaves não sejam necessárias quando apenas uma instrução está presente no corpo do laço **do-while**, elas são geralmente usadas para aumentar a legibilidade.

Vamos escrever o programa *adivinha.c* baseado no programa *adiv.c* escrito anteriormente, agora com um laço **do-while**.

```
/* adivinha.c */
/* testa a sua capacidade de adivinhar uma letra */
main()
{
   char ch,c;
   int tentativas;

   do {
```

```
      ch= rand()%26 +'a';
      tentativas=1;
      printf("\nDigite uma letra de 'a' a 'z':\n");
      while((c=getch())!=ch) {
         printf("%c e' incorreto. Tente novamente.\n\n",c);
         tentativas++;
      }
      printf("\n%c e' correto",c);
      printf("\nVoce acertou em %d tentativas",tentativas);
      printf("\nQuer jogar novamente ? (s/n): ");
   } while(getche()=='s');
}
```

QUANDO USAR do-while

Existe uma estimativa que aponta os laços **do-while** necessários somente em 5% dos laços. Várias razões influem na consideração de laços que avaliam a expressão de teste antes de serem executados como superiores.

Uma delas é a legibilidade, isto é, ler a expressão de teste antes de percorrer o laço ajuda o leitor a interpretar facilmente o sentido do bloco de instruções. Outra razão é a possibilidade da execução do laço mesmo que o teste seja falso de início.

OS COMANDOS break E continue

O comando **break** pode ser usado no corpo de qualquer estrutura de laço C. Causa a saída imediata do laço e o controle passa para o próximo estágio do programa.

Se o comando **break** estiver em estruturas de laços aninhados, afetará somente o laço que o contém e os laços internos a este.

Como exemplo, vamos tomar o programa *fator.c* que usa um laço infinito, e inserir uma instrução **break**.


```
/* fator1.c */
/* encontra o fatorial de um unico numero */

main()
{
   int num;
   long resposta;

   while(1) {
      printf("\n Digite o numero: ");
      scanf("%d",&num);
      resposta=1;
      while(num > 1)
         resposta *= num--;
      printf("O fatorial e': %ld\n", resposta);
      break;
   }
}
```

O comando **continue** força a próxima interação do laço e pula o código que estiver abaixo. Nos laços **while** e **do-while** um comando **continue** faz com que o controle do programa vá diretamente para o teste condicional e depois continue o processo do laço. No caso do laço **for**, o computador primeiro executa o incremento do laço e, depois, o teste condicional, e finalmente faz com que o laço continue.

O comando **continue** deve ser evitado, pois pode causar dificuldades de leitura e confusão ao se manter o programa.

O COMANDO goto

O comando **goto** está disponível em C para fornecer alguma compatibilidade com as linguagens como BASIC e FORTRAN. Os criadores da linguagem C, Kernighan e Ritchie, desaprovam o seu uso pois não existe nenhuma situação em que ele seja absolutamente necessário em C. Assim, este livro não fará o seu uso em nenhum programa.

A linguagem C tem as três estruturas de laços vistas anteriormente que permitem o controle de estruturas do programa, e os comandos

break e **continue** que aumentam estas possibilidades; assim o uso de **goto** torna-se desnecessário.

A instrução **goto** tem duas partes: a palavra **goto** e um nome. O nome segue as convenções de nomes de variáveis C. Por exemplo:

```
goto parte1;
```

Para que esta instrução opere, deve haver um rótulo em outra parte do programa. Um rótulo é um nome seguido por dois pontos.

```
parte1:
    printf("Análise dos Resultados\n");
```

A princípio você nunca precisará usar **goto** em seus programas. Mas se você tiver um programa em BASIC ou em FORTRAN que deve ser convertido para a linguagem C rapidamente, o comando **goto** irá ajudá-lo.

A instrução **goto** causa o desvio do controle do programa para a instrução seguinte ao rótulo com o nome indicado. Os dois pontos são usados para separar o nome da instrução.

REVISÃO

1. A possibilidade de repetir ações é uma das razões pelas quais usamos o computador. Os laços **for**, **while** e **do-while** são as estruturas oferecidas por C para cumprirem esta tarefa.
2. O operador vírgula permite que sejam inicializadas ou incrementadas mais de uma expressão no laço **for**.
3. Uma construção agrupada por chaves é tratada como uma única instrução.
4. O corpo de um laço ou **while** pode nunca ser executado. Entretanto o corpo de um laço **do-while** é sempre executado pelo menos uma vez.
5. Laços podem ser colocados no corpo de outros laços à vontade em C.
6. Os operadores relacionais têm precedência maior que a do operador de atribuição. Portanto, expressões relacionais contendo atribuição devem ter parênteses.


```
( c=getche() ) != '\n'
```

7. A função **rand()** retorna um inteiro aleatório.
8. Num laço **while** a expressão de teste é avaliada antes do corpo ser executado.
9. Num laço **do-while** a expressão de teste é avaliada depois do corpo do laço ser executado.
10. Um laço **do-while** é sempre encerrado por ponto-e-vírgula.
11. O comando **break** causa a saída imediata do laço que o contém.
12. O comando **continue** força a próxima interação do laço que o contém. Nos laços **while** e **do-while** o controle passa para a expressão de teste e no laço **for** o controle passa para a expressão de incremento.
13. O comando **goto** causa o desvio do controle para a instrução seguinte ao rótulo indicado.

EXERCÍCIOS

1. Qual a saída produzida pela execução do laço seguinte?

```
int a;
for( a=36 ; a>0 ; a/=2)
   printf("%d",a);
```

2. Qual a saída produzida pela execução do programa seguinte?

```
main()
{
   int i;

   for(printf("Inicializacao\n"), i=0;
      printf("Teste i=%d",i), i<5;
      printf("Incremento\n"), i++)
         ;
   printf("FORA DO LACO ! \n");
}
```

3. Escreva um programa usando um laço **for** que imprima os caracteres da tabela ASCII de códigos 32 a 255 decimal. O programa deve imprimir cada caractere, seu código decimal e seu código hexadecimal.

4. Escreva um programa usando um laço **for** que imprima uma linha na tela com o caractere gráfico de código **DB hexa**.

5. A expressão de inicialização de um laço **for** é executada uma única vez antes do laço ser iniciado.
Verdadeiro ou *Falso*: Os dois blocos seguintes produzem o mesmo resultado.

a)
```
for(i=0 ; i<10 ; i++)
      for(j=0 ; j<10 ; j++)
            printf("Lacos aninhados\n");
```

b)
```
for(i=0 , j=0 ; i<10 ; i++)
      for(; j<10 ; j++)
            printf("Lacos aninhados\n");
```

6. Uma expressão sem parênteses contendo operador relacional, operador de atribuição e operador aritmético é avaliada na seguinte ordem:

   a) atribuição, relacional, aritmético;
   b) aritmético, relacional, atribuição;
   c) relacional, aritmético, atribuição;
   d) atribuição, aritmético, relacional.

7. Faça um programa, utilizando um laço **while**, que solicite caracteres ao usuário e imprima seus códigos decimais. O programa deve terminar quando o usuário pressionar a tecla **Esc**.

8. Faça um programa que solicite um número inteiro de até 4 dígitos ao usuário e inverta a ordem de seus algarismos. Por exemplo, uma execução do programa é:

   Digite com um numero de ate 4 digitos: 5382
   Seu numero invertido e': 2835

9. Escreva um programa que imprima o quadrado de todos os inteiros de 1 a 20.

10. Escreva um programa que solicite dois caracteres ao usuário e imprima o número de caracteres que estão entre eles. Assuma que o usuário digitará os 2 caracteres em ordem alfabética. Exemplo:

    Digite 2 caracteres: c f
    O numero de caracteres entre eles e': 2

11. O número de combinações de **n** objetos diferentes, onde **r** objetos são escolhidos de cada vez, é dado pela seguinte fórmula:

$$_{n}C_{r} = \frac{\text{fatorial (n)}}{\text{fatorial (r)} \times \text{fatorial (n}-\text{r)}}$$

    Escreva um programa que calcule o número de combinações de **n** objetos tomados **r** de cada vez. Os valores **n** e **r** devem ser solicitados ao usuário.

MAKRON Books

CAPÍTULO 4

COMANDOS DE DECISÕES

- *O Comando if*
- *O Comando if-else*
- *Operadores Lógicos*
  || && !
- *O Comando else-if*
- *O Comando switch*
- *O Operador Condicional Ternário*
  ? :

Toda linguagem para computador precisa oferecer um mínimo de três formas básicas de controle:

1. Executar uma série de instruções.
2. Repetir uma seqüência de instruções até que uma certa condição seja encontrada.
3. Praticar testes para decidir entre ações alternativas.

A primeira e a segunda formas você já conhece. A última forma será explorada neste capítulo.

C oferece 4 principais estruturas de decisão : **if, if-else, switch** e o **operador condicional.**

O COMANDO if

O comando **if** instrui o computador a tomar uma decisão simples.

Forma geral:

```
if( expressão de teste )
    instrução;
```

Por exemplo:

```
/* testif.c */
/* mostra o uso do comando if */
main()
{
   char ch;
   ch = getche();
   if(ch =='p')
   printf("\nVoce pressionou a tecla p.");
}
```

Se você digitar 'p' o programa imprimirá "Voce pressionou a tecla p.". Se você apertar qualquer outra tecla, o programa não fará absolutamente nada.

O modo de operação de um comando **if** é bastante similar ao do laço **while**. Em ambos os casos o bloco de instruções não é executado se a expressão de teste for falsa. Entretanto, o laço **while** pode executar o bloco de instruções várias vezes, enquanto que um comando **if** o executa uma única vez, se a expressão de teste for verdadeira.

O PROGRAMA QUE CONTA PALAVRAS DA ENTRADA

No capítulo anterior, escrevemos o programa *contchar.c* que conta o número de caracteres numa frase. O programa a seguir é ligeiramente mais complexo e contará não somente caracteres mas também o número de palavras da frase.

```
/* contpal.c */
/* conta caracteres e palavras de uma frase */

main()
{
   int caracteres = 0;
   int palavras = 0;
   char ch;

   printf("Digite uma frase:\n");
   while((ch=getche()) != '\r') { /* le caractere e */
      caracteres ++; /* termina o laco com <RETURN> */
      if(ch == ' ') /* espaco ? */
         palavras ++; /* conta palavra */
   }
   printf("\nForam contados %d caracteres ", caracteres);
   printf("\ne %d palavras nesta frase", palavras + 1);
}
```

Eis uma execução:

```
C>contpal
Digite uma frase:
O vicio e' um erro de calculo na busca da felicidade.
Foram contados 52 caracteres
e 11 palavras nesta frase
```


A parte principal deste programa é o laço **while** que lê os caracteres do teclado até que seja pressionado **<RETURN>**.

O corpo do laço é composto por duas instruções: a primeira incrementa o contador de caracteres a cada leitura e a segunda verifica se o caractere é um espaço branco e, se for, o programa entende que uma palavra foi datilografada e incrementa o contador de palavras. Como o último caractere digitado não é um espaço em branco, indicando o término de uma palavra, o contador de palavras irá conter um número a menos do número de palavras digitadas na saída do laço. Assim **printf()** imprime **palavras+1**.

INSTRUÇÕES MÚLTIPLAS NO CORPO DO COMANDO if

Como nas outras estruturas já vistas, caso várias instruções sejam necessárias no corpo do comando **if** elas devem estar entre chaves. Como exemplo, reescreveremos o programa *testif.c* acrescentando outras instruções ao corpo do comando **if**.

```
/* testif2.c */
/* mostra o uso de multiplas instrucoes no corpo de um if*/

main()
{
   if(getche() == 'p') {
      printf("\nVoce pressionou a tecla p.");
      printf("\nPressione qualquer tecla para terminar.");
      getche();
   }
}
```

Como o corpo do comando **if** é composto por três instruções, elas devem estar entre chaves.

Em *testif2.c* incluimos a função **getche()** na expressão de teste do **if**. Esta instrução faz uma chamada a **getche()** e compara o caractere retornado com a letra 'p' e, se forem iguais, a expressão toda assume um valor verdadeiro, e o bloco de instruções é executado.

COMANDOS if ANINHADOS

Um comando **if** pode estar dentro de outro comando **if**. Dizemos então que o **if** interno está aninhado. Eis um exemplo:

```
/* ninhosif.c */
/* mostra if aninhados */

main()
{
   char ch;
   printf("Digite uma letra de 'a' a 'z':");
   ch=getche();
   if(ch >= 'a')
      if(ch <= 'z')
         printf("\nVoce digitou certo !!.");
}
```

Este programa somente imprimirá a frase "Voce digitou certo !!." se você digitar uma letra minúscula.

IMPLEMENTANDO UM ALGORITMO

Deparei, um dia, com um estranho método de encontrar o quadrado de um número positivo. O algoritmo é o seguinte:

O quadrado de um número positivo **n** *é*
igual à soma dos **n** *primeiros números ímpares.*

Por exemplo, o quadrado de 3 é

$9 = 1 + 3 + 5$

e o de 6 é

$36 = 1 + 3 + 5 + 7 + 9 + 11$

Este algoritmo pode ser traduzido na seguinte fórmula matemática que facilmente é demonstrada por indução finita.

$$n^2 = \sum_{i=0}^{n-1} (2i + 1)$$

O programa a seguir mostra como implementar este algoritmo.

```
/* quadr.c */
/* implementacao de um algoritmo */

main()
{
   int n, i, soma;

   printf("Digite o numero a ser elevado ao quadrado\n");
   scanf("%d",&n);
   printf("O quadrado de %d",n);
   if(n<0) n=-n;
   for(i=1,soma=0;n>0;soma+=i,n--,i+=2)
     ;
   printf("e' %d",soma);
}
```

Eis uma execução:

```
C>quadr
Digite o numero a ser elevado ao quadrado
8
O quadrado de 8 e' 64
```

O COMANDO if-else

Nos exemplos anteriores o comando **if** executará uma única instrução ou um grupo de instruções, se a expressão de teste for verdadeira. Não fará nada se a expressão de teste for falsa.

O comando **else**, quando associado ao **if**, executará uma instrução ou um grupo de instruções entre chaves, se a expressão de teste do comando **if** for falsa.

Forma geral:

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

Reescreveremos *testif.c* para que imprima uma mensagem caso a expressão de teste do **if** seja falsa:

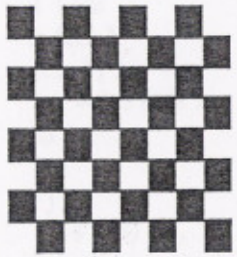
```
/* testelse.c */
/* mostra o comando if-else */

main()
{
   char ch;
   ch = getche();

   if(ch == 'p')
      printf("\nVoce pressionou a tecla p.");
   else
      printf("\nVoce nao pressionou a tecla p.");
}
```

CARACTERES GRÁFICOS E UM TABULEIRO DE XADREZ

Os caracteres gráficos do IBM-PC podem ser usados para fazer desenhos realmente atraentes. O programa seguinte imprime um tabuleiro de xadrez na tela. A estrutura de controle da impressão é feita por dois laços **for**, sendo um para controlar as linhas e o outro para controlar as colunas. O corpo do laço interno é composto por um comando **if-else** que reconhece quando imprimir um quadrado cheio e quando imprimir um quadrado branco.

Eis a listagem:

```
/* xadrez.c */
/* desenha um tabuleiro de xadrez na tela */
main()
{
   int x,y;

   for(y=1;y<9;y++) { /* passo de descida */
      for(x=1;x<9;x++) /* passo de largura */
         if((x+y)%2 == 0) /* e' numero par ? */
            printf("\xDB\xDB");/*imprime quadr.cheio*/
         else
            printf("  "); /*imprime quadr.branco*/
      printf("\n");  /* linha nova*/
   }
}
```

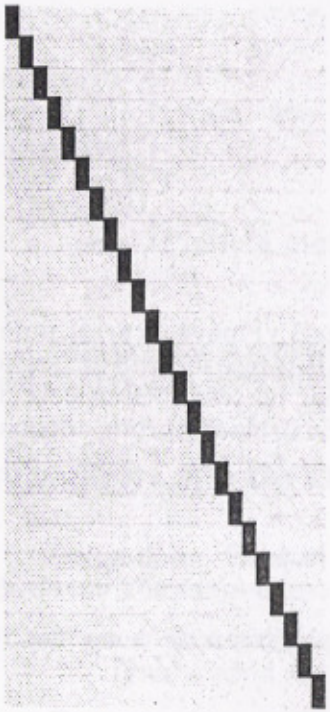
O laço **for** mais externo (da variável **y**) move o cursor uma linha para baixo a cada interação até que **y** seja 9. O laço interno (da variável **x**) move o cursor na horizontal uma coluna por vez (cada coluna é da largura de 2 caracteres) até que **x** seja 9. O comando **if-else** imprime ora quadrado cheio, ora quadrado branco.

DESENHANDO LINHAS

Outros exemplos de caracteres gráficos e de comandos **if-else** estão nos dois programas a seguir.

```
/* diag1.c */
/* imprime uma linha diagonal na tela */

main()
{
int x,y;

   for(y=1;y<24;y++) { /* passo de descida */
```

```
      for(x=1;x<24;x++) /* passo de largura */
         if(x == y)  /* estamos na diagonal? */
            printf("\xDB"); /*sim,desenha ret.escuro*/
         else
            printf("\xB0");/*nao,desenha ret.claro */
      printf("\n"); /* nova linha*/
   }
}
```

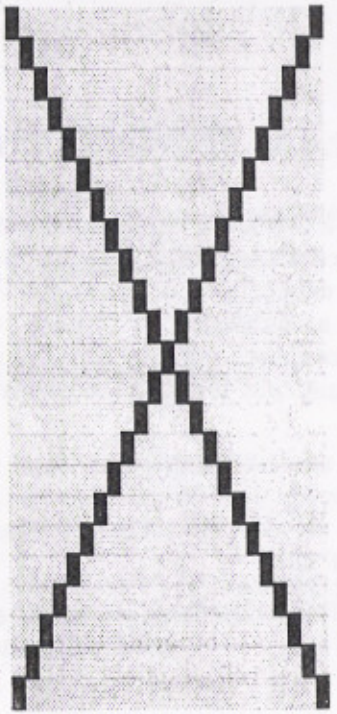


A estrutura de controle deste programa é a mesma da do programa *xadrez.c*.

COMANDOS if-else ANINHADOS

É perfeitamente possível aninhar construções **if-else**; veja o exemplo seguinte:


```
/* diag2.c */
/* imprime 2 linhas diagonais na tela */

main()
{
int x,y;

   for(y=1;y<24;y++){ /* passo de descida */
      for(x=1;x<24;x++) /* passo da largura */
         if(x==y) /* diagonal 1? */
            printf("\xDB"); /* cor escura */
         else
            if(x==24-y) /* diagonal 2? */
               printf("\xDB"); /* cor escura */
            else
               printf("\xB0"); /* cor clara */
      printf("\n"); /* nova linha */
   }
}
```



Este programa é similar ao anterior, exceto no fato de imprimir duas linhas diagonais na tela ao invés de uma. Estas duas linhas criam um X centralizado no retângulo claro.

Quando você tem um certo número de **if**(s) e **else**(s), como o computador decide qual **if** é de qual **else**? Por exemplo, considere o seguinte fragmento de programa:

```
if(n > 0)
   if(a > b)
      z = a;
   else
      z = b;
```

Quando será executada a instrução **z=b;** ? Quando **n** for menor ou igual a zero ou quando **b** for menor ou igual a **a** ? Em outras palavras o **else** está associado ao primeiro ou segundo **if** ?

O **else** é sempre associado ao mais recente **if** sem **else**. Então, se **n** for igual a -5 nada será executado, e se **n** for igual a 2 e **a** for menor do que **b**, a instrução **z=b;** será executada. Caso não seja isto o que você quer, use chaves:

```
if(n > 0) {
   if(a > b)
      z = a;
   } else
      z = b;
```

OPERADORES LÓGICOS

C oferece 3 operadores chamados lógicos:

&&	lógico E
||	lógico OU
!	lógico de negação

Destes operadores **!** é unário e **&&** e **||** são binários. Os operadores lógicos são geralmente aplicados a expressões relacionais.

Se **exp1** e **exp2** são duas expressões simples, então:

exp1 && exp2	é verdadeira se as duas **exp1** e **exp2** forem verdadeiras.
exp1 || exp2	é verdadeira se uma das duas **exp1** ou **exp2** for verdadeira ou as duas **exp1** e **exp2** forem verdadeiras.
!exp1	é verdadeira se **exp1** for falsa e vice-versa.

Alguns exemplos:

```
1 || 2
x && Y
<b || a==c
a>=2.8 && x<=6.5*y
!5
!x
!'j'
!(a+2.8)
```

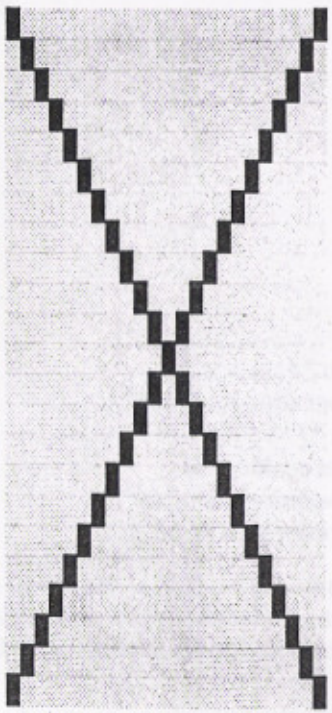
O que você não pode fazer:

```
x&&     /* erro - um operando */
x|y     /* erro - este é um operador bit-a-bit*/
x| |y   /* erro - espaço extra */
&x      /* erro - este é o endereço de x */
x&y     /* erro - este é um operador bit-a-bit*/
x!=y    /* erro - este é o operador não igual */
```

Podemos simplificar o programa *diag2.c* com o emprego de operador lógico. Operadores lógicos são um poderoso meio de condensar e clarear construções **if-else**. Mostraremos o uso do operador lógico **OU** (**||**) reescrevendo o programa *diag2.c*.

```
/* diag3.c */
/* imprime 2 linhas diagonais na tela */

main()
{
   int x,y;

   for(y=1;y<24;y++){ /* passo de descida */
```

```
      for(x=1;x<24;x++) /* passo da largura */
        if(x==y || x==24-y) /* diagonal? */
          printf("\xDB"); /* cor escura */
        else
          printf("\xB0"); /* cor clara */
      printf("\n"); /* nova linha */
   }
}
```



O programa vai gerar a mesma saída que o *diag2.c* mas com uma escrita mais elegante. Eis um exemplo que faz uso do operador **E** (&&):

```
/* contdig.c */
/* conta caracteres e digitos em uma frase */
main()
{
   int numchar=0;
   int numdigit=0;
```



```
   char c;

   printf("Digite uma frase:\n");
   while((c=getche()) != '\r') {
      numchar++;
      if(c >='0' && c <='9')
        numdigit++;
   }
   printf("\n0 numero de caracteres e' %d",numchar);
   printf("\n0 numero de digitos e' %d",numdigit);
}
```

Eis uma execução:

```
C>contdig
Digite uma frase:
12 vozes se faziam escutar.
O numero de caracteres e' 27
O numero de digitos e' 2
```

Este programa é uma modificação dos programas *contchar.c* e *contapal.c*. Conta caracteres de uma frase e dígitos de 0 a 9 que fizerem parte da frase.

O operador unário de negação **!** resulta um valor diferente de zero ou verdadeiro se o operando for 0, e um valor zero ou falso se o operando for diferente de 0.

Veja mais alguns exemplos:

```
!(x < 5)
```

é verdadeiro quando **x** for maior ou igual a 5.

```
if ( ! empalavra )
```

ao invés de

```
if ( empalavra == 0 )
```

PRECEDÊNCIA

O operador ! é o de maior precedência, a mesma que a do operador menos unário. A tabela seguinte mostra as precedências dos operadores:

Operadores	Tipos
! - ++ --	unários; não lógico e menos aritmético
* / %	aritméticos
+ -	aritméticos
< > <= >=	relacionais
== !=	relacionais
&&	lógico E
||	lógico OU
= += -= *= /= %=	aritméticos de atribuição

O COMANDO else-if

Vamos, agora, ver um exemplo um pouco mais complexo. Escreveremos o programa *calc.c* com os comandos **if** e **else** aninhados para, depois, reescrevê-lo com **else-if**. Este programa simula uma calculadora simples com 4 operações.

```
/* calc.c */
/* calculadora com 4 operacoes */

main()
{
   float num1,num2;
   char op;

   while(1) { /* sempre verdadeiro */
      printf("Digite um numero, operador, numero\n");
      scanf("%f %c %f",&num1,&op,&num2);
      if(op=='+')
         printf(" = %f",num1+num2);
      else
         if(op=='-')
            printf(" = %f",num1-num2);
```



```
         else
            if(op=='*')
               printf(" = %f",num1*num2);
            else
               if(op=='/')
                  printf(" = %f",num1/num2);
      printf("\n\n");
   }
}
```

Eis uma execução:

```
C>Digite um numero, operador, numero
4+2=6
Digite um numero, operador, numero
5-2=3
Digite um numero, operador, numero
3*5=15
<ctrl><break>
```

Agora com **else-if**:

```
/* calc2.c */
/* calculadora com 4 operacoes */
main()
{
   float num1,num2;
   char op;

   while(1) { /* sempre verdadeiro */
      printf("Digite um numero, operador, numero\n");
      scanf("%f %c %f",&num1,&op,&num2);
      if(op=='+')
         printf(" = %f",num1+num2);
      else if(op=='-')
         printf(" = %f",num1-num2);
      else if(op=='*')
         printf(" = %f",num1*num2);
      else if(op=='/')
         printf(" = %f",num1/num2);
      printf("\n\n");
   }
}
```

A construção **else-if** é uma maneira reformatada de ninhos **if-else.**

O COMANDO break ASSOCIADO AO if

Mostraremos a escapada **break** de um comando **if** dentro de um laço **while** com um jogo de adivinhações. Neste jogo, o usuário pensa um número entre 1 e 99 e o programa adivinhará qual é.

```
/* numadiv.c */
/* programa adivinha numero que o usuario pensou */

main()
{
   float adiv,incr;
   char ch;

   printf("Pense um numero entre 1 e 99, e\n");
   printf("eu adivinharei qual e'. Digite:\n");
   printf("\n '=' para igual,\n");
   printf(" '>' para maior que\n");
   printf(" '<' para menor que.\n");

   incr=adiv=50;
   while( incr > 1) {
      printf("\n >, <, = a %.0f?\n",adiv);
      incr /= 2;
      if((ch=getche())=='=')
         break;
      else if(ch=='>')
         adiv += incr;
      else
         adiv -= incr;
   }
    printf("\nO numero e' %.0f.\n",adiv);
    printf("\nCOMO SOU ESPERTO !!!!! ");
}
```


Eis uma execução:

```
C>Pense um numero de 1 a 99, e eu adivinharei qual e'. Digite:
          '=' para igual,
          '>' para maior que
          '<' para menor que
>, <, = a 50?
>
>, <, = a 75?
<
>, <, = a 63?
<
>, <, = a 56?
>
>, <, = a 61?
>
O numero e' 62.
COMO SOU ESPERTO!!!!!
```

O comando **break** é freqüentemente usado quando ocorre uma condição repentina onde é necessário deixar o laço antes da expressão do laço se tornar falsa.

O COMANDO switch

Construções **if-else** facilitam a escrita de programas que devem escolher uma entre duas alternativas. Algumas vezes, entretanto, o programa necessita escolher uma entre várias alternativas.

Embora construções **else-if** possam executar testes de vários modos, elas não são de maneira nenhuma elegante. O código pode ficar difícil de ser seguido e confundir até mesmo seu autor num momento futuro. Para estes casos C oferece a construção **switch.**

O comando **switch** é similar ao **else-if** mas tem maior flexibilidade e formato limpo e claro.

Forma geral:

```
switch( expressao constante ) {
      case constante1 :
            instrucoes;    /* opcional */
      case constante2 :
```

```
                instrucoes;    /* opcional */
        .....
        default :              /* opcional */
                instrucoes;    /* opcional */
    }
```

O nosso primeiro exemplo calcula o dia da semana a partir de uma data. O ano deve ser maior ou igual a 1600, pois nesta data houve uma redefinição do calendário.

```
/* diaseman.c */
/* calcula o dia da semana a partir de uma data */
main()
{
   int D,M,A,i;
   long int F=0;
   do{
      do {
         printf("\nDigite a data na forma dd/mm/aaaa: ");
         scanf("%d/%d/%d",&D,&M,&A);
      } while(A<1600 || M<1 || M>12 || D<1 || D>31);
      F=A+D+3*(M-1)-1;

      if(M<3) {
         A--;i=0;
      }else
         i=.4*M+2.3;

      F+=A/4-i; i=A/100+1; i*=.75; F-=i; F%=7;

      switch(F){
         case 0:printf("\nDomingo");break;
         case 1:printf("\nSegunda-feira");break;
         case 2:printf("\nTerca-feira"); break;
         case 3:printf("\nQuarta-feira");break;
         case 4:printf("\nQuinta-feira");break;
         case 5:printf("\nSexta-feira");break;
         case 6:printf("\nSabado");break;
      }
   }while(getche()!=27);
}
```


Observe a execução do programa.

```
Digite a data na forma dd/mm/aaaa: 07/11/1974
Quinta-feira
Digite a data na forma dd/mm/aaaa: 12/01/1976
Segunda-feira
Digite a data na forma dd/mm/aaaa: 30/07/1978
Domingo
[Esc]
```

O comando **switch** avalia a expressão entre parênteses, após a palavra **switch**, e compara seu valor com os rótulos dos casos.

A expressão entre parênteses deve ser de valor inteiro ou caractere.

Cada caso deve ser rotulado por uma constante do tipo inteiro ou caractere ou por uma expressão constante. Você não poderá usar uma variável e nem uma expressão lógica para rótulo. Esta constante deve ser terminada por dois pontos (:) e **não** por ponto-e-vírgula.

Pode haver uma ou mais instruções seguindo cada **case**. Estas instruções não necessitam estar entre chaves.

O corpo de um **switch** deve estar envolto por chaves.

Se um caso for igual ao valor da expressão, a execução começa nele.

Se nenhum caso for satisfeito e existir um caso *default:* a execução começará nele, senão o programa processará as instruções seguintes ao bloco **switch**. Um *default:* é opcional.

Os rótulos dos casos devem ser todos diferentes.

O comando **break** causa uma saída imediata do **switch**. Se não existir um comando **break** seguindo as instruções de um caso, o programa segue executando todas as instruções dos casos abaixo.

Reescreveremos o programa *calc2.c* fazendo uso do comando **switch**.

```
/* calc3.c */
/* calculadora com 4 operacoes */

main()
```

```
{
   float num1,num2;
   char op;

   while(1) { /* sempre verdadeiro */
      printf("Digite um numero, operador, numero\n");
      scanf("%f %c %f",&num1,&op,&num2);

      switch(op) {

         case '+':
            printf(" = %f",num1+num2);
            break;
         case '-':
            printf(" = %f",num1-num2);
            break;
         case '*':
            printf(" = %f",num1*num2);
            break;
         case '/':
            printf(" = %f",num1/num2);
            break;
         default:
            printf("Operador desconhecido");
      }
      printf("\n\n");
   }
}
```

O programa *calc4.c* mostrará casos sem **break.**

```
/* calc4.c */
/* calculadora com 4 operacoes */

main()
{
   float num1,num2;
   char op;

   while(1) { /* sempre verdadeiro */
      printf("Digite um numero, operador, numero\n");
      scanf("%f %c %f",&num1,&op,&num2);

      switch(op) {

         case '+':
            printf(" = %f",num1+num2);
            break;
         case '-':
            printf(" = %f",num1-num2);
            break;
         case '*':
         case 'x':
            printf(" = %f",num1*num2);
            break;
         case '/':
         case '\\':
            printf(" = %f",num1/num2);
            break;
         default:
            printf("Operador desconhecido");
      }
      printf("\n\n");
   }
}
```


Nesta versão o usuário pode digitar o sinal '*' ou 'x' para multiplicação e '/' ou '\' para divisão.

O OPERADOR CONDICIONAL TERNÁRIO ?:

Terminaremos este capítulo com uma breve discussão sobre o operador C que possui uma construção um pouco estranha, chamado *operador condicional.*

Ele nos dá uma maneira compacta de expressar uma simples instrução **if-else.**

Este operador consiste em dois operadores separando a expressão toda em três diferentes expressões e é o único operador ternário em C.

Forma geral:

condição ? expressão_1 : expressão_2

A condição consiste em uma expressão lógica avaliada como verdadeira ou falsa.

A condição é avaliada e se verdadeira (não zero) a expressão condicional toda assume o valor da expressão_1, se falsa a expressão assume o valor da expressão_2.

Eis alguns exemplos:

```
max = (num1 > num2) ? num1 : num2;
```

Esta expressão é equivalente à construção **if-else** seguinte, mas bem mais compacta.

```
if (num1 > num2)
   max = num1;
else
   max = num2;
```

Outro exemplo:

```
abs = (num < 0) ? -num : num;
```

Expressões com o operador condicional não são necessárias, visto que o comando **if-else** pode substituí-las. São, entretanto, mais compactas e geram um código de máquina menor.

REVISÃO

1. As principais estruturas de decisão C são **if**, **if-else**, **switch** e o **operador condicional**.
2. O comando **if** oferece um meio de escolha entre executar ou não um bloco de instruções. O comando **if-else** oferece um meio de executar um ou outro bloco de instruções.

3. Operadores lógicos geralmente são aplicados a expressões relacionais como operandos.
4. Expressões lógicas são avaliadas da esquerda para a direita e a avaliação termina logo que é conhecida a veracidade ou falsidade da expressão toda.
5. O comando **switch** oferece um meio de escolher uma entre muitas opções. A expressão do comando **switch** deve ter um valor inteiro ou *char* constante. As expressões de cada caso devem ter um valor inteiro ou *char* constante e devem ser todas diferentes.
6. Num comando **switch** o controle do programa pula para o caso em que a expressão for igual à expressão do **switch** ou, se não houver nenhum caso igual, para o default. Um caso default é opcional.
7. O comando **break** provoca uma saída imediata da construção **switch**.
8. O operador condicional opera sobre três expressões. A primeira é avaliada e, se verdadeira, a expressão toda assume o valor da segunda expressão; caso contrário, assume o valor da terceira expressão.

EXERCÍCIOS

1. Numa simples construção **if** sem **else**, o que acontece se a condição seguida ao **if** for falsa?

   a) O controle procura pelo último **else** no programa.
   b) Nada.
   c) O controle passa para a instrução seguinte ao **if**.
   d) O corpo do comando **if** é executado.

2. O programa seguinte é correto?

```
main()
{
   if(getche() == 'a') then
      printf("\nVoce teclou a.");
}
```

3. O programa seguinte é correto?

```
main()
{
   int a,b,area;
   scanf("%d %d",&a,&b);
   if a == b
      area=a*a;
   else
      area=a*b;
   printf("\narea= %d.",area);
}
```

4. O programa seguinte é correto?

```
main()
{
   int a,b;

   scanf("%d %d",&a,&b);

   if( a != b) {
      a=1;
      b=2;
   };
   else
      a+=b;
   printf("\n%d %d.",a,b);
}
```

5. A principal diferença no modo de operação de um comando **if** e de um laço **while** é:

a) a expressão condicional seguida ao comando é avaliada diferentemente;

b) o corpo do laço **while** é sempre executado, e o corpo de um comando **if** somente é se a condição for verdadeira;

c) o corpo do laço **while** pode ser executado várias vezes e o corpo de um comando **if** somente uma;

d) a expressão condicional é avaliada antes do corpo do laço **while** ser executado, mas depois do corpo de um comando **if**.

6. A instrução **else** numa construção **if-else** é executada quando:

a) a expressão de condição seguida ao **if** for falsa;

b) a expressão de condição seguida ao **if** for verdadeira;

c) a expressão de condição seguida ao **else** for falsa;

d) a expressão de condição seguida ao **else** for verdadeira.

7. O programa C abaixo é correto?

```
main()
{
   if(getch()=='a') printf("e' a");else printf("nao e'");
}
```

8. *Verdadeiro ou Falso:* o compilador interpreta **else-if** de modo diferente de **if-else**.

9. A instrução seguinte a um particular **else-if** numa malha **else-if** é executada quando:

a) a expressão condicional que segue o **else-if** for verdadeira e todas as condições anteriores forem verdadeiras;

b) a expressão condicional que segue o **else-if** for verdadeira e todas as condições anteriores forem falsas;

c) a expressão condicional que segue o **else-if** for falsa e todas as condições anteriores forem verdadeiras;

d) a expressão condicional que segue o **else-if** for falsa e todas as condições anteriores forem falsas.

10. Num programa, o comando **else** fará par com qual **if** acima?

a) O último **if** com mesmos requisitos que o **else**.

b) O último **if** sem **else**.

c) O último **if** não envolto por chaves.

d) O último **if** não envolto por chaves e sem **else**.

11. Indique o valor de cada uma das seguintes expressões:

```
int i=1, j=2, k=3, n=2;
float x=3.3, y=4.4;
```

a) i < j + 3
b) 2 * i - 7 <= j - 8
c) -x + y >= 2.0 * y
d) i==y
e) i!=y
f) i + j + k == -2 * -k
g) !(n-j)
h) !n-j
i) !x * !x
j) i && j && k
k) i || j - 3 && 0
l) i < j && 2 >= k
m) i < j || 2 >= k
n) i == 2 || j == 4 || k == 5
o) i = 2 || j == 4 || k == 5
p) x <= 5.0 && x != 1.0 || i > j

12. Escreva expressões equivalentes sem usar o operador de negação (!).

a) !(i==j)
b) !(i + 1 < j - 2)
c) !(i < j && n < m)
d) !(i < 1 || j < 2 && n < 3)

13. Se **i** é uma variável inteira, descreva o efeito deste código:

```
while( i = 8) {
   printf("%d %d %d\n",i,i+2,i+3);
   i=0;
}
```

E agora compare com este

```
if(i=8)
   printf("%d %d %d\n",i,i+2,i+3);
```

Os dois estão logicamente errados. Explique.

14. Qual a saída deste programa?

```
main()
{
   int a=1, b=2;

   if(a==2)
      if(b==2)
         printf("%d\n",a=a+b);
   else
      printf("%d\n",a=a-b);
   printf("%d\n",a);
}
```

15. A vantagem de uma construção **switch** sobre uma **else-if** é:

a) a condição default pode ser usada em **switch**;
b) **switch** fornece uma leitura mais fácil de se entender;
c) diversas instruções diferentes podem ser executadas em cada caso de um **switch**;
d) diversas condições diferentes podem causar uma escolha a ser executada num **switch**.

16. A seguinte estrutura **switch** é correta?

```
switch(num) {
   case 1;
      printf("Numero e' 1");
   case 2;
      printf("Numero e' 2");
   default;
      printf("O numero e' diferente de 1 e 2");
}
```

17. *Verdadeiro* ou *Falso*: o comando **break** deve ser usado seguindo as instruções de cada **case** num **switch**.

18. A seguinte estrutura **switch** é correta?

```
switch(temp) {
   case temp < 60:
      printf("Esta verdadeiramente frio!");
      break;
   case temp < 80:
      printf("Que tempo agradavel!");
      break;
   default:
      printf("Certamente esta quente!");
}
```

19. A tarefa do operador condicional é:

a) selecionar o maior entre dois valores;
b) selecionar o mais igual entre dois valores;
c) selecionar um de dois valores alternativamente;
d) selecionar um de dois valores dependendo da condição.

20. Se *num* é -42, qual é o valor da expressão condicional?

```
( num < 0 ) ? 0 : num*num;
```

21. O valor da seguinte expressão é obscuro. Use parênteses para clareá-la.

```
a = x < y ? x < z ? x : z : y < z ? y : z;
```

22. Escreva um programa que leia caractere a caractere do teclado usando a função **getch()**; se o caractere lido for uma letra minúscula imprima-a em maiúsculo, caso contrário imprima o próprio caractere. O programa termina quando a tecla **[Esc]** for pressionada.

23. Escreva um programa que pergunte ao usuário com qual velocidade costuma dirigir seu carro e imprima a resposta que o guarda de trânsito daria conforme as seguintes velocidades: > 75 km/h, > 65 km/h, > 55 km/h, > 45 km/h, < 45 km/h. Use comandos **if-else** aninhados.

24. Modifique o programa *xadrez.c* para que imprima cada quadrado do tabuleiro com 3 linhas de altura e 6 colunas de largura ao invés de 1 linha de altura e 2 colunas de largura.

25. Modifique o programa *diag2.c* para que imprima 4 linhas: as primeiras duas como são impressas em *diag2.c*; a terceira, uma linha vertical passando pelo centro do retângulo (onde as duas primeiras se cortam); e a quarta, uma linha horizontal passando pelo mesmo centro. Use a construção **else-if** em malha. Não use operadores lógicos.

26. Modifique o programa do exercício anterior para que trabalhe com operadores lógicos eliminando o **else-if** em malha.

27. Escreva um programa que encontre o menor inteiro positivo **n** que aceite as seguintes condições:

n/3 = x inteiros e resto 2
n/5 = y inteiros e resto 3
n/7 = z inteiros e resto 4

MAKRON Books

CAPÍTULO 5

FUNÇÕES

- *Funções e Estrutura de um Programa*
- *Funções que Retornam um Valor*
- *O Comando return*
- *Funções não Inteiras*
- *Argumentos e Chamada por Valor*
- *Funções Recursivas*
- *Classes de Armazenamento*
  auto, extern, static, register
- *Uma Função que Gera Números Aleatórios*
- *Considerações Sobre Conflitos de Nomes de Variáveis*
- *O Pré-processador C*
- *A Diretiva #define*
- *Macros*
- *A Diretiva #include*
- *Outras Diretivas*
  #undef #if #ifdef #ifndef #else #endif

FUNÇÕES E ESTRUTURA DE UM PROGRAMA

Funções dividem grandes tarefas de computação em tarefas menores, e permitem às pessoas trabalharem sobre o que outras já fizeram, ao invés de partir do nada. Funções apropriadas podem freqüentemente esconder detalhes de operação de partes de programa que não necessitam conhecê-las, esclarecendo o todo, e facilitando mudanças. Você já usou a função **printf()** sem conhecer detalhes de sua programação.

O que é uma função ? Uma função é uma unidade de código de programa autônoma desenhada para cumprir uma tarefa particular.

C foi projetada com funções eficientes e fáceis de usar; programas em C geralmente consistem em várias pequenas funções ao invés de poucas de maior tamanho.

Provavelmente a principal razão da existência de funções é impedir que o programador tenha de escrever o mesmo código repetidas vezes. Suponha que você tenha, em seu programa, um parágrafo onde se calcula o quadrado de um número. Se, mais adiante, no programa, você precisar da mesma instrução, deverá escrevê-la novamente. Em vez disto você poderia saltar para uma seção, do código, que calcula o quadrado e voltar novamente à mesma posição. Trabalhando assim, uma simples seção do código pode ser usada repetidas vezes no mesmo programa.

FUNÇÕES SIMPLES

O uso de uma função pode ser comparado à forma de alugarmos a mão-de-obra de alguém para executar um trabalho específico. Algumas vezes a interação com semelhante pessoa é bem simples; outras vezes, mais complexa. Vamos começar com um caso simples: nosso exemplo cria uma função que imprime 20 caracteres sólidos numa linha. A seguir está a listagem completa que consiste na função **main()** e na função **linha()**.

```
/* moldtext.c */
/* envolve um texto por uma moldura */

main()
{
```

```
    linha();
    printf("\xDB UM PROGRAMA EM C \xDB\n");
    linha();
}

/* linha() */
/* desenha uma linha solida na tela, 20 caracteres */
linha()
{
    int j;
    for(j=1;j<=20;j++)
        printf("\xDB");
    printf("\n");
}
```

A saída será:

```
UM PROGRAMA EM C
```

ESTRUTURA DAS FUNÇÕES EM C

Como você pode ver, a estrutura de uma função C é bastante semelhante à da função **main()**. A única diferença é que **main()** possui um nome especial. Como vimos no Capítulo 1, a função **main()** é a primeira a ser chamada quando o programa é executado.

No exemplo anterior, a função **main()** foi colocada no começo da listagem. Poderíamos colocar a função **linha()** primeiro e o programa trabalharia exatamente da mesma forma.

Todas as funções em C começam com um nome seguido de parênteses (que envolve ou não uma lista de argumentos) e, após isto, chaves que envolvem o corpo da função.

CHAMANDO FUNÇÕES

Do mesmo modo que chamamos uma função de biblioteca C (**printf()**, **getche()**, etc.) chamamos nossas próprias funções como **linha()**. Os parentêses que seguem o nome são necessários para que o compilador possa diferenciar a chamada a uma função de uma variável que você esqueceu de declarar.

Visto que a chamada a uma função constitui uma instrução de programa, deve ser encerrada por ponto-e-vírgula.

```
linha();
```

Entretanto, na definição de uma função, o ponto-e-vírgula não pode ser usado.

```
linha()
```

VARIÁVEIS LOCAIS

Cada função pode chamar outras funções. A função **linha()** chama a função **printf()** e a função **main()** chama **linha()** e **printf()**.

As variáveis que são declaradas dentro de uma função são chamadas *variáveis locais* e são conhecidas somente dentro de seu próprio bloco. Um bloco começa quando o computador encontra uma chave de abertura ({) e termina quando o computador encontra uma chave de fechamento (}).

Um dos aspectos mais importantes que você deve entender sobre as variáveis locais é que elas existem apenas durante a execução do bloco de código onde estão declaradas; isto é, uma variável local é criada quando se entra em seu bloco e destruída na saída.

A variável **j** criada na função **linha()** é conhecida somente em **linha()**; e é invisível à função **main()**. Se adicionarmos a instrução abaixo à função **main()** (sem declarar a variável **j**):

```
printf("%d", j);
```

teremos um erro de compilação, pois **main()** não conhece nada sobre esta variável.

Podemos declarar uma outra variável **j** em **main()**; ela será uma variável completamente diferente e será conhecida em **main()** e não em **linha()**.

Uma variável local é conhecida em C como variável "automática", pois é automaticamente criada quando a função é chamada e destruída na saída.

UM EXEMPLO SONORO

Este exemplo usará o caractere especial '\x7', chamado BELL na tabela ASCII. Este caractere provoca um "beep" sonoro no alto-falante do seu micro.

```
/* beepteste */
/* testa a funcao doisbeep */
main()
{
   doisbeep();
   printf("Digite um caractere: ");
   getche();
   doisbeep();
}
/* doisbeep() */
/* toca o auto-falante duas vezes */
doisbeep()
{
   int k;
   printf("\x7");
   for(k=1;k<5000;k++)
      ;
   printf("\x7");
}
```


FUNÇÕES QUE RETORNAM UM VALOR

Discutiremos, agora, uma espécie ligeiramente mais complicada de função: a que retorna um valor. Você já usou uma função deste tipo, **getche()** que devolve, à função que chama, o valor do primeiro caractere pressionado no teclado.

O próximo programa cria uma função que lê um caractere do teclado e, se for maiúsculo, converte-o em minúsculo.

```
/* menu.c */
/* testa a funcao minusculo() */

main()
{
   char ch;

   printf("Digite 'a' e depois 'b': ");
   ch=minusculo();
   switch(ch) {
      case 'a':
         printf("\nVoce pressionou 'a'.");
         break;
      case 'b':
         printf("\nVoce pressionou 'b'.");
         break;
      default:
         printf("\nVoce escolheu algo desconhecido.");
   }
}

/* minusculo() */
/* retorna um caractere */
/* converte para minusculo se for maiusculo */

minúsculo()
{
   char ch;
   ch=getche();              /* le um caractere */
```

```
   if(ch >='A' && ch <='Z')   /* se maiusculo ? */
      ch += 'a'-'A';          /* converte para minusculo */
   return(ch);                /* devolve valor do caractere */
}
```

O COMANDO return

Nos programas *moldtext.c* e *beeptest.c* mostramos funções que retornam ao programa que chamou quando encontram a chave (}) que termina a função. Não há necessidade de uma instrução **return**.

O comando **return** causa a atribuição de qualquer expressão entre parênteses à função contendo o **return**. Então quando **minúsculo()** é chamada por **main()** ela adquire o valor do caractere lido e convertido em minúsculo.

A variável **ch** é local a **minúsculo()**, mas o valor de **ch** é mandado para **main()** pelo comando **return**. Este valor pode, então, ser atribuído a uma variável, como no nosso exemplo, ou fazer parte de alguma expressão.

O comando **return** tem dois usos importantes. Primeiro, você pode usar **return()** para devolver um valor e retornar, imediatamente, para a próxima instrução do código de chamada. Segundo, você pode usá-lo, sem os parênteses, para causar uma saída imediata da função na qual ele se encontra; isto é, **return** fará com que a execução do programa volte para o código de chamada assim que o computador encontrar este comando, o que ocorre, em geral, antes da última instrução da função.

Você pode colocar mais de um comando **return** em suas funções. Como exemplo vamos reescrever a função **minúsculo()** colocando dois comandos **return**.

```
/* minusculo() */
/* retorna um caractere */
/* converte para minusculo se for maiusculo */

minusculo()
{
   char ch;
```



```
   ch=getche();               /* le um caractere */

   if(ch >= 'A' && ch <= 'Z')  /* se maiusculo ? */
      return(ch + 'a'-'A');    /* devolve convertido */
   else
      return(ch);              /* devolve sem converter*/
}
```

HORAS E MINUTOS

Este é um outro exemplo de função que retorna um valor. O programa chama a função **minutos()** que solicita a hora e minutos ao usuário e calcula a diferença entre dois tempos.

```
/* minuts.c */
/* calcula a diferenca entre dois tempos */

main()
{
   int mins1, mins2;

   printf("Digite a primeira hora (hora:min): ");
   mins1=minutos();
   printf("Digite a segunda hora (hora:min): ");
   mins2=minutos();
   printf("A diferenca e' %d minutos.",mins2-mins1);
}

/* minutos() */
/* solicita hora:minutos */
/* retorna hora em minutos */
minutos()
{
   int hora, min;
   scanf("%d:%d",&hora,&min);
   return(hora*60 + min);
}
```

UM NOVO USO DE scanf()

Se você prestou atenção deve ter notado algo novo no argumento de **scanf()**. Os dois pontos entre as especificações de formato **%d** e **%d**.

```
scanf("%d:%d",&hora,&minutos);
```

Quando um caractere está presente na expressão de controle de **scanf()** ele deve ser digitado pelo usuário na posição onde se encontra. Assim, você deverá digitar dois pontos entre os dois números. Se você pressionar qualquer outra tecla, **scanf()** terminará imediatamente sua execução sem esperar que o outro argumento (minutos) seja digitado.

LIMITAÇÕES DE return()

O comando **return()** pode retornar somente um único valor à função que chama.

PASSANDO DADOS PARA A FUNÇÃO CHAMADA

O mecanismo usado para transmitir informações para uma função é chamado *argumento*. Você já usou *argumentos* nas funções **printf()** e **scanf()**.

Vamos construir uma função que retorna o valor absoluto de um número. O valor absoluto de um número é o próprio número quando o sinal é ignorado. Por exemplo, o valor absoluto de 5 é 5 e o valor absoluto de -5 é também 5.

```
/* testabs.c */
/* testa funcao abs() */
main()
{
   printf("%d %d %d\n",abs(0),abs(-3),abs(10));
}
```



```
/* abs() */
/* retorna o valor absoluto de um numero */
abs(x)
int x;
{
   return((x < 0)? -x:x); /* lembra o operador ternario? */
}
```

A definição da nossa função começa com duas linhas:

```
abs(x)
int   x;
```

A primeira linha informa ao compilador que **abs()** requer um argumento, e este argumento será chamado **x**. A segunda linha é uma declaração que informa ao compilador que **x** é do tipo **int**. Observe que argumentos são declarados antes da chave que marca o início do corpo da função. Estas duas linhas podem ser condensadas em uma:

```
abs(int x)
```

A variável **x** é na verdade uma nova variável e é chamada de argumento "formal" funcionando exatamente como uma variável local da função; isto é, é criada quando a função inicia sua execução e destruída quando a função retorna.

O próximo exemplo chama várias vezes a função **bar()** que recebe um argumento e desenha uma barra horizontal do tamanho do valor do argumento recebido.

```
/* bargraf.c */
/* desenha um grafico de barras */

main()
{
   printf("Luiza\t");
   bar(27);
   printf("Chris\t");
   bar(41);
   printf("Regina\t");
   bar(34);
```

```
   printf("Cindy\t");
   bar(22);
   printf("Harold\t");
   bar(15);
}

/* bar() */
/* funcao grafico de barras horizontal */

bar(pontos)
int pontos;              /* declaracao de argumento formal */
{
   int j;

   for(j=1;j<=pontos;j++)
      printf("\xCD");        /* traco duplo */
   printf("\n");
}
```

A saída será:

```
Luiza    ═══════════════════════
Chris    ═══════════════════════════════════
Regina   ═══════════════════════════
Cindy    ═══════════════
Harold   ═══════════
```

PASSANDO VARIÁVEIS COMO ARGUMENTOS

Nos exemplos anteriores nós passamos constantes como argumentos para as funções **bar()** e **abs()**. Podemos, entretanto, passar variáveis da função que chama, como mostra uma variante do programa *bargraf.c*:

```
/* bargraf1.c */
/* desenha um grafico de barras */
main()
{
   int enpontos;
   while(1) {
```



```
      printf("(0 para terminar) Pontos = ");
      scanf("%d",&enpontos);
      if(!enpontos)
         break;
      else
         bar(enpontos);
   }
}

/* bar() */
/* funcao grafico de barras horizontal */

bar(pontos)
int pontos;              /* declaracao de argumento formal */
{
   int j;

   for(j=1;j<=pontos;j++)
      printf("\xCD");   /* traco duplo */
   printf("\n");
}
```

```
(0 para terminar) pontos = 10
══════════
```

PASSANDO VÁRIOS ARGUMENTOS

Se mais de um argumento é necessário, podem ser passados na lista de argumentos separados por vírgulas. Podemos passar quantos argumentos desejarmos para uma função. A seguir está um exemplo de um programa que passa dois argumentos para a função **retang()**, cujo propósito é desenhar retângulos de vários tamanhos na tela. Os dois argumentos são a largura e altura do retângulo; onde cada retângulo representa um cômodo de uma casa.

```
/* descomod.c */
/* testa a funcao retang() */
```

```
main()
{
   printf("\nSala\n");
   retang(22,12);
   printf("\nCozinha\n");
   retang(16,16);
   printf("\nBanheiro\n");
   retang(6,8);
   printf("\nQuarto\n");
   retang(12,12);
}

/* retang() */
/* desenha retangulo */

retang(largura,altura)
int largura,altura;
{
   int j,k;

   largura /=2;
   altura /=4;
   for(j=1;j <= altura; j++) {
      printf("\t\t");
      for(k=1;k <= largura; k++)
         printf("\xDB");
      printf("\n");
   }
}
```

Este programa imprime o nome do cômodo e desenha o retângulo correspondente às suas dimensões.

Sala

Cozinha

Banheiro

Quarto

USANDO MAIS DE UMA FUNÇÃO

Você pode usar quantas funções você quiser num programa, e qualquer função pode chamar qualquer outra.

Em C não é permitido definir uma função dentro de outra e todas as funções são visíveis por todas as outras.

O programa *moldtext.c* imprime um texto, envolto por uma moldura no canto esquerdo da tela. Vamos escrever uma função que imprime espaços para centralizar a sua saída.

```
/* moldtex1.c */
/* centraliza um texto com moldura */

main()
{
   espacos(30);
   linha();
   espacos(30);
   printf("\xDB UM PROGRAMA EM C \xDB\n");
   espacos(30);
   linha();
}

/* linha() */
/* desenha uma linha solida na tela, 20 caracteres */
linha()
{
   int j;
   for(j=1;j<=20;j++)
      printf("\xDB");
   printf("\n");
}
```

```
* espacos() */
/* imprime espacos */
espacos(x)
int x;
{
   for(i=0;i<x;i++)
      printf(" ");
}
```

A saída será:

```
UM PROGRAMA EM C
```

O próximo programa calcula o quadrado de dois inteiros fornecidos pelo usuário. A função **main()** chama três funções. A primeira retorna o resultado já calculado. A segunda calcula o quadrado dos argumentos e a terceira retorna a soma de dois números.

```
/* multifun.c */
/* testa funcoes soma, sqr e somasqr */

main()
{
   int num1,num2;
   printf("Digite dois numeros: ");
   scanf("%d %d",&num1,&num2);
   printf("A soma dos quadrados e' %d",somasqr(num1,num2));
}

/* somasqr() */
/* retorna a soma dos quadrados de dois argumentos */

somasqr(j,k)
int j,k;
{
   return(soma(sqr(j),sqr(k)));
}
```



```
/* sqr() */
/* retorna o quadrado do argumento */

sqr(z)
int z;
{
   return(z*z);
}

/* soma() */
/* retorna a soma de dois argumentos */

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

FUNÇÕES NÃO INTEIRAS

O tipo de uma função é determinado pelo tipo de valor que ela retorna e não pelo tipo de seus argumentos.

Até agora temos trabalhado somente com funções inteiras. Uma função é dita do tipo inteira quando retorna um inteiro.

Se uma função for do tipo não inteira ela deve obrigatoriamente ser declarada. Se você omitir a declaração de uma função, C assume por default que a função é **int**; isto é, a função retorna um **int**. Uma função que retorna um **char** ou não retorna nada pode ser considerada como inteira.

Vamos analisar funções que retornam um valor diferente de **int**, por exemplo, um tipo **float**.

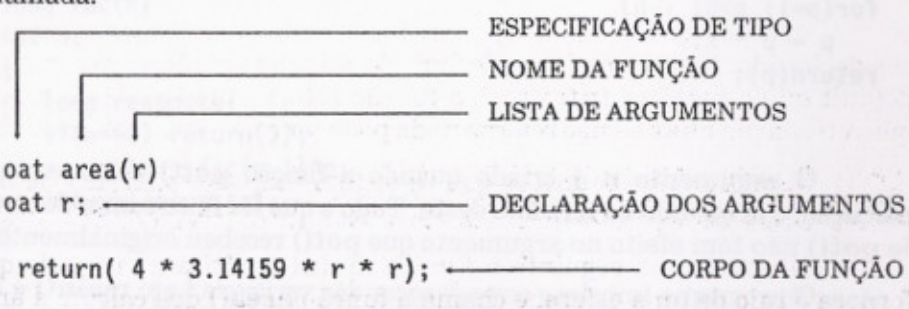
No programa seguinte a função **main()** solicita ao usuário que forneça o raio de uma esfera, e chama a função **area()** que calcula a área da esfera na forma de ponto flutuante.

```
/* esfera.c */
/* calcula a area da esfera */
```

```
main()
{
float area();              /* declaracao da funcao */
float raio;

printf("Digite o raio da esfera: ");
scanf("%f",&raio);
printf("A area da esfera e' %.2f",area(raio));
}

/* area() */
/* retorna a area da esfera */

float area(r) /* definicao da funcao */
float r;
{
return( 4 * 3.14159 * r * r); /* retorna float */
}
```

Note que a única diferença entre a declaração de uma função e a de uma variável é a inclusão dos parênteses depois do nome da função, que revelam ao compilador reconhecer que é função.

O tipo de uma função não inteira deve constar, precedendo o nome, na sua definição. C assume por default que todas as funções são do tipo **int**, a não ser que você indique outro tipo. Se uma função faz uma chamada a uma função não inteira, a função que chama deve declarar a função chamada.



FUNÇÕES DO TIPO int

A declaração de funções do tipo **int** não é necessária mas é um bom estilo de programação.

Vamos reescrever *minuts.c* declarando a função **minutos()**.

```
/* minuts.c */
/* calcula a diferenca entre dois tempos */

main()
{
   int minutos();        /* declaracao da funcao */
   int mins1, mins2;

   printf("Digite a primeira hora (hora:min): ");
   mins1=minutos();
   printf("Digite a segunda hora (hora:min): ");
   mins2=minutos();
   printf("A diferenca e' %d minutos.",mins2-mins1);
}

/* minutos() */
/* solicita hora:minutos */
/* retorna hora em minutos */

int minutos()         /* definicao da funcao */
{
   int hora, min;

   scanf("%d:%d",&hora,&min);
   return(hora*60 + min);
}
```

FUNÇÕES DO TIPO void

Se uma função retorna um valor do tipo **int** pode ser declarada, no seu programa, como tipo **int**; como declarar funções que não retornam nada?

O comitê de padronização da linguagem do American National Standards Institute (**ANSI**), em suas novas propostas, batizou funções que não retornam nada como tendo um novo tipo: **void**.

Vamos reescrever *moldtext.c* declarando a função **linha()**:

```
/* moldtext.c */
/* envolve um texto por uma moldura */

main()
{
   void linha();  /* declaracao da funcao */
   linha();
   printf("\xDB UM PROGRAMA EM C \xDB\n");
   linha();
}

/* linha() */
/* desenha uma linha solida na tela, 20 caracteres */

   void linha()   /* definicao da funcao */
{
   int j;

   for(j=1;j<=20;j++)
      printf("\xDB");
   printf("\n");
}
```

ARGUMENTOS – CHAMADA POR VALOR

Em C, todos os argumentos de funções são passados "por valor".

Isto significa que à função chamada é dada uma cópia dos valores dos argumentos, e ela cria outras variáveis temporárias para armazenar estes valores.

A diferença principal é que, em C, uma função chamada *não pode* alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária.

Vamos escrever uma função **pot**(m,n) que eleva um inteiro **m** à potência positiva inteira **n**. Isto é, o valor de **pot**(2,5) é 32. Por exemplo:

```
/* testpot.c */
/* mostra chamada por valor */
/* teste da funcao de potencia */
main()
{
   int i;

   printf("\n0 endereco de i e': %u\n",&i);
   for(i=0; i<10; ++i)
      printf("%d %d %d \n",i , pot(2,i),pot(-3,i));
}

/* pot() */
/* eleva x a potencia n; n>0 */
pot(x,n)
int x, n;
{
   int p;

   printf("\n0 endereco de n e': %u\n",&n);
   for(p=1; n>0; --n)
      p = p * x;
   return(p);
}
```

O argumento **n** é criado quando a função **pot()** inicia a sua execução, e destruído ao término desta. Tudo o que for feito com **n** dentro de **pot()** não tem efeito no argumento que **pot()** recebeu originalmente.

O programa imprime os endereços das variáveis **i** em **main()** e **n** em **pot()** e você pode verificar que são realmente duas variáveis diferentes.

FUNÇÕES RECURSIVAS

Uma função é dita recursiva se é definida em termos de si mesma. Isto é, uma função é recursiva quando dentro dela está presente uma chamada a ela própria.

Como exemplo, vamos reescrever o programa que calcula o fatorial de um número agora com uma função recursiva.

```
/* fator1.c */
/* encontra o fatorial de um numero */
/* chama funcao recursiva */
main()
{
   int num;
   long fac();
   while(1) {
      printf("\nDigite um numero: ");
      scanf("%d",&num);
      printf("\nO fatorial de %d e' %ld",num, fac(num));
   }
}

/* fac() */
/* calcula fatorial */
/* recursiva */
long fac(n)
int n;
{
   long resposta;
   if(n==0) return(1);
   resposta = fac(n-1)*n;
   return(resposta);
}
```

O código gerado por uma função recursiva exige a utilização de mais memória e é o que torna a execução mais lenta. Não é difícil criar funções que façam chamadas a elas mesmas. O difícil é reconhecer situações apropriadas às chamadas recursivas. O objetivo de uma função

recursiva é fazer com que ela passe por uma seqüência de chamadas até que um certo ponto seja atingido.

A função fatorial é um dos exemplos mais conhecidos de recursividade. Alguns pontos de sua montagem devem ser observados:

A definição de fatorial sugere recursão.

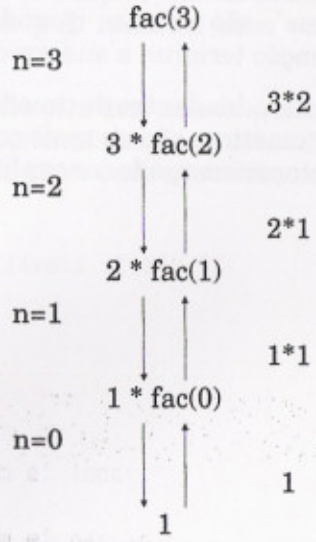
fatorial(n) = n * fatorial(n - 1)

A função **fac()**, quando chamada, verifica se foi satisfeita a condição básica para interromper a recursão.

if(n==0)

Cada vez que a função é chamada recursivamente o seu argumento está mais próximo da condição básica. Isto garante que o programa não girará em uma seqüência infindável de chamadas.

O diagrama seguinte mostra a série de chamadas à função **fac()** para n=3.



CLASSES DE ARMAZENAMENTO

Todas as variáveis e funções C têm dois atributos: um tipo e uma classe de armazenamento. Os tipos nós já conhecemos. As classes de armazenamento serão vistas a seguir.

São quatro as classes de armazenamento de variáveis C:

- auto (automáticas)
- extern (externas)
- static (estáticas)
- register (em registradores)

CLASSE DE ARMAZENAMENTO – auto

As variáveis que temos visto em todos os nossos exemplos estão confinadas nas funções que as usam; isto é, são "visíveis" ou "acessíveis" somente às funções onde estão declaradas. Tais variáveis são chamadas "locais" ou "automáticas", são criadas quando a função é chamada e destruídas quando a função termina a sua execução.

As variáveis declaradas dentro de uma função são automáticas por "default". Variáveis automáticas são as mais comuns dentre as 4 classes. A classe de variáveis automáticas pode ser explicitada usando-se a palavra **auto**. O código

```
main()
{
   auto int n;
   . . .
}
```

é equivalente a

```
main()
{
   int n;
   . . .
}
```


CLASSE DE ARMAZENAMENTO – extern

Todas as funções C e todas as variáveis declaradas fora de qualquer função têm a classe de armazenamento **extern**. Variáveis com este atributo serão conhecidas por todas as funções declaradas depois delas. A declaração de variáveis externas é feita da mesma maneira como declaramos variáveis dentro do bloco de uma função.

Enquanto variáveis locais são preferidas para várias tarefas, é às vezes desejável o uso de variáveis conhecidas de todas as funções do programa. O exemplo seguinte mostra o seu uso.

```
/* extern.c */
/* testa o uso de variaveis extern */

int teclanum; /* variavel externa */

main()
{
   extern teclanum;

   printf("Digite teclanum: ");
   scanf("%d",&teclanum);
   parimpar();
   negativo();
}

/* parimpar() */
/* checa se teclanum e' par ou impar */
parimpar()
{
   extern teclanum;

   if(teclanum % 2)
      printf("teclanum e' impar.\n);
   else
      printf("teclanum e' par.\n);
}
```

```
/* negativo() */
/* checa se teclanum e' negativo */

negativo()
{
   extern teclanum;
   if(teclanum < 0)
      printf("teclanum e' negativo.\n);
   else
      printf("teclanum e' positivo.\n);
}
```

Uma variável definida fora de qualquer função é dita **externa**. A palavra **extern** indica que a função usará uma variável externa. Este tipo de declaração, entretanto, não é obrigatória se a definição original ocorre no mesmo arquivo fonte.

A próxima versão de *extern.c* é completamente correta:

```
/* extern.c */
/* testa o uso de variaveis extern */

int teclanum; /* variavel externa */

main()
{
   printf("Digite teclanum: ");
   scanf("%d",&teclanum);
   parimpar();
   negativo();
}

/* parimpar() */
/* checa se teclanum e' par ou impar */
parimpar()
{
   if(teclanum % 2)
      printf("teclanum e' impar.\n);
   else
      printf("teclanum e' par.\n);
}
```



```
/* negativo() */
/* checa se teclanum e' negativo */
negativo()
{
   if(teclanum < 0)
      printf("teclanum e' negativo.\n);
   else
      printf("teclanum e' positivo.\n);
}
```

Variáveis externas retêm seus valores durante toda a execução do programa, mantendo sua posição de memória alocada, portanto só devem ser usadas se você tiver um ótimo motivo para isto. O uso de argumentos de funções é um método preferível para passar informações para a função.

CLASSE DE ARMAZENAMENTO – static

Variáveis **static** de um lado se assemelham às automáticas, pois são conhecidas somente as funções que as declaram e de outro lado se assemelham às externas pois mantém seus valores mesmo quando a função termina.

Declarações **static** têm dois usos importantes e distintos. O mais elementar é permitir a variáveis locais reterem seus valores mesmo após o término da execução do bloco onde são declaradas.

```
/* estatic.c */
/* mostra o uso de variaveis static */
main()
{
   soma();
   soma();
   soma();
}

/* soma() */
/* usa variavel static */
soma()
{
```

```
    static int i=0;
    printf("i = %d\n",i++);
}
```

A saída será:

```
i = 0
i = 1
i = 2
```

Observe que **i** não é inicializada a cada chamada de **soma()**.

VARIÁVEIS ESTÁTICAS EXTERNAS

O segundo e mais poderoso uso de **static** é associado a declarações externas. Junto a construções externas, permite um mecanismo de "privacidade" muito importante à programação modular.

A diferença entre variáveis externas e externas estáticas é que variáveis externas podem ser usadas por qualquer função abaixo de suas declarações, enquanto que variáveis externas estáticas somente podem ser usadas pelas funções do mesmo arquivo-fonte e abaixo de suas declarações.

Como exemplo deste mecanismo de privacidade, escreveremos duas funções que geram números aleatórios.

UMA FUNÇÃO QUE GERA NÚMEROS ALEATÓRIOS

Você já usou uma função de biblioteca C que gera números aleatórios, **rand()**.

Agora vamos criar a nossa própria função **rand()**, visto que ela não é implementada em todos os compiladores.

A primeira versão criará números "pseudo-aleatórios". O esquema começa com um número chamado "semente" que é usado para produzir um novo número que se tornará a nova semente. A nova semente é usada para produzir uma nova semente e assim por diante. Com este esquema

a função de número aleatório deve se lembrar da semente usada na última chamada e para isto variáveis de classe **static** se adaptam perfeitamente. O método é baseado em congruência linear que não explicaremos aqui.

```
/* rand() */
/* gera numeros pseudo-aleatorios */
/* versao 1 */
rand()
{
   static int semente=1;

   semente=(semente*25173+13849)%65536; /* formula magica */
   return(semente);
}
```

A variável estática **semente** começa com o valor 1 e este é alterado pela fórmula mágica a cada chamada a esta função. O resultado é um número no intervalo de -32768 a 32767.

O programa seguinte chama esta função cinco vezes:

```
/* testrand.c */
/* chama rand() */
main()
{
   int c;
   for (c=1;c<=5;c++)
   printf("%d\n",rand());
}
```

Ao executar este programa você terá a seguinte saída:

```
-26514
-4449
20196
-20531
3882
```

Agora vamos executar novamente o programa e o resultado será:

```
-26514
-4449
20196
-20531
3882
```

Observe que a saída é exatamente a mesma nas duas execuções do programa. Daí o nome "pseudo-aleatório". O problema é que a cada execução a variável **semente** começa com o valor 1. Para resolvê-lo vamos introduzir uma segunda função que chamaremos de **sement()**, que recompõe a semente a cada execução.

```
/* arquivo fonte sement() e rand() */
static int semente=1;
/* rand() */
/* gera numeros pseudo-aleatorios */
/* versao 2 */

rand()
{
   semente=(semente*25173+13849)%65536; /* formula magica */
   return(semente);
}

/* sement() */
/* recompoe a semente */
sement(n)
unsigned int n;
{
   semente=n;
   return(semente);
}
```

O programa seguinte testa estas funções.

```
/* testrand.c */
/* chama rand() */
/* versao 2 */
main()
{
   int c;
   int semente;

   printf("Digite a sua semente:\n");
   scanf("%d",&semente);
   sement(semente);
   for(c=1;c<=5;c++)
      printf("%d\n",rand());
}
```


Analise agora duas execuções da versão dois:

Digite a sua semente:

```
1

-26514
-4449
20196
-20531
3882
```

Digite a sua semente:

```
3

23832
20241
-1858
-30417
-16204
```

A nossa função de geração de números aleatórios produz números de -32768 a 32767, mas você pode fazer vários ajustes em suas chamadas. Aqui vão alguns exemplos:

Gerando números do tipo **float**:

1. Dividindo o número aleatório por 32768 resultará um número x do tipo **float** tal que -1 < = x < 1.
2. Adicionando 1 resultará x tal que 0 < = x < 2.
3. Dividindo por 2 teremos 0 < = x < 1.

4. Multiplicando por 6 teremos 0 < = x < 6.
5. Adicionando 1 teremos 1 < = x < 7.

E assim por diante.

Gerando números inteiros:

1. Tomando o resto da divisão do número aleatório sem sinal por 101 resultarão números de 0 a 100.
2. Tomando o resto da divisão do número aleatório sem sinal por 11 resultarão números de 0 a 10.

E assim por diante.

O último uso de **static** é para especificar a classe de armazenamento de funções. Funções **static** são conhecidas somente dentro do arquivo-fonte em que foram declaradas, ao contrário das outras funções que podem ser acessadas por outros arquivos compilados juntos.

CLASSE DE ARMAZENAMENTO – register

A classe de armazenamento **register** indica que a variável associada deve ser guardada fisicamente numa memória de acesso muito mais rápido chamada registrador. Um registrador da máquina é um espaço de 16 bits (no IBM-PC) onde podemos armazenar um **int** ou um **char**. Em outras palavras, variáveis da classe **register** são semelhantes às automáticas mas se aplicam apenas às variáveis do tipo **int** e **char**.

Cada máquina oferece um certo número de registradores que podem ser manuseados pelos usuários. Por exemplo, o computador IBM-PC oferece somente dois registradores, portanto só podemos ter duas variáveis em registradores ao mesmo tempo. Mas isto não nos impede de declarar quantas variáveis **register** quisermos. Se os registradores estiverem ocupados o computador simplesmente ignora a palavra **register** das nossas declarações. No segundo volume deste livro entraremos em mais detalhes sobre registradores.

Basicamente variáveis **register** são usadas para aumentar a velocidade de processamento. Assim, o programador deve escolher as variáveis que são mais freqüentemente acessadas e declará-las como da classe

register. Fortes candidatas a este tratamento são as variáveis de laços e argumentos de funções.

Eis um exemplo que mostra a diferença na velocidade entre variáveis de memória e em registradores.

```
/* mostra variaveis register */
main()
{
   int i,j;                /* guardadas na memoria */
   register int m,n;       /* guardadas em registradores */
   long t;

   t = time(0);
   for(j=0;j<=1000;j++)
      for(i=0;i<=1000;i++)
         ;
   printf("Tempo dos lacos nao register: %ld\n",time(0)-t);

   t = time(0);
   for(m=0;m<=1000;m++)
      for(n=0;n<=1000;n++)
         ;
   printf("Tempo dos lacos register: %ld\n",time(0)-t);
}
```

Este programa usa a rotina **time()** presente na biblioteca do compilador Turbo C da Borland.

CONSIDERAÇÕES SOBRE CONFLITO DE NOMES DE VARIÁVEIS

Sempre que duas variáveis tiverem o mesmo nome mas diferentes endereços de memória elas não são as mesmas variáveis. Eis um exemplo:

```
main()
{
   int i=5;          /* PRIMEIRA VARIAVEL */
```

```
   printf("Em main() - endereco de i=%u\n",&i);
   funcl(i);
   printf("Em main() - valor de i= %d\n",i);
}

funcl(i)
int i;                  /* SEGUNDA VARIAVEL */
{
   i=i+2;
   printf("Em funcl() - endereco de i=%u\n",&i);
   printf("Em funcl() - valor de i= %d\n",i);
}
```

Este conceito se estende a blocos dentro de uma mesma função. Por exemplo:

```
main()
{
   int i=5;             /* PRIMEIRA VARIAVEL */

   printf("Em main() - endereco de i=%u\n",&i);
   printf("Em main() - valor de i= %d\n",i);

   if(i==5) {
      int i;   /* SEGUNDA VARIAVEL */
      i=2;
      printf("No bloco - endereco de i=%u\n",&i);
      printf("No bloco - valor de i= %d\n",i);
   }
   printf("Fora do bloco - valor de i= %d\n",i);
}
```

As duas variáveis **i** não são as mesmas e você reconhece isto, pois elas têm endereços de memória diferentes.

A segunda variável **i** é conhecida somente dentro das chaves do comando **if**. Uma vez executado o bloco do comando **if**, ela é destruída.

Uma variável local é limitada ao bloco em que ela foi declarada. Se duas variáveis partilham o mesmo nome, a que foi declarada no bloco atual tem precedência sobre a que está declarada num bloco diferente.

O que acontece quando declaramos uma variável local com o mesmo nome de uma variável externa?

O próximo exemplo mostra esta situação.

```
int i=5;

main()
{
   int i;
   i=10;
   printf("\nEm main() i=%d",i);
   funcl();
}

funcl()
{
   printf("\nEm funcl() i=%d",i);
}
```

Estas duas variáveis são diferentes e você pode verificar isto modificando o programa para que imprima seus endereços.

A variável declarada em **main()** tem precedência sobre a externa no bloco onde foi declarada.

O PRÉ-PROCESSADOR C

O **pré-processador** C é um programa que examina o programa-fonte em C e executa certas modificações nele, baseado em instruções chamadas diretivas.

O **pré-processador** faz parte do compilador e pode ser considerado uma linguagem dentro da linguagem C. Ele é executado automa-

ticamente antes da compilação. Diretivas do **pré-processador** seriam instruções desta linguagem.

As instruções desta linguagem são executadas antes do programa ser compilado e têm a tarefa de alterar os programas-fontes, na sua forma de texto.

Instruções para o **pré-processador** devem fazer parte do texto que criamos, mas não farão parte do programa que compilamos, pois são retiradas do texto pelo compilador antes da compilação.

Para entendermos como funcionam as diretivas do **pré-processador**, vamos primeiro revisar como funciona o compilador.

Quando você escreve uma linha de programa

```
num = 40;
```

você está solicitando ao compilador transformar este código para instruções em linguagem de máquina que podem ser executadas pelo "chip" microprocessador do computador.

Conseqüentemente, a maior parte da sua listagem consiste em instruções para o microprocessador. Diretivas do **pré-processador** são instruções para o compilador propriamente dito. Mais precisamente, elas operam diretamente no compilador antes do processo de compilação ser iniciado; por isso o nome **pré-processador**.

Linhas normais de programa são instruções para o microprocessador; diretivas do pré-processador são instruções para o compilador.

Todas as diretivas do **pré-processador** são iniciadas com o símbolo (**#**). As diretivas podem ser colocadas em qualquer parte do programa, mas é costume serem colocadas no início do programa, antes de **main()**, ou antes do começo de uma função particular.

A DIRETIVA #define

A diretiva **#define** pode ser usada para definir constantes simbólicas com nomes apropriados. Como exemplo vamos modificar o programa *esfera.c* para que a constante **3.14159** seja definida com o nome **PI**.


```
/* esfera.c */
/* calcula a area da esfera */

#define PI 3.14159

main()
{
   float area();              /* declaracao da funcao */
   float raio;
   printf("Digite o raio da esfera: ");
   scanf("%f",&raio);
   printf("A area da esfera e' %.2f",area(raio));
}

/* area() */
/* retorna a area da esfera */
float area(r)  /* definicao da funcao */
float r;
{
   return( 4 * PI * r * r);   /* retorna float */
}
```


Quando o compilador encontra **#define**, ele substitui cada ocorrência de **PI** no programa por **3.14159**.

```
#define PI 3.14159
```

A frase à esquerda (**PI**), que será procurada, é chamada "identificador". A frase à direita (**3.14159**), que será substituída, é chamada "texto". Um ou mais espaços separam o identificador do texto.

Por convenção, o identificador (neste caso **PI**) é escrito em letras maiúsculas.

Note que só podemos escrever um comando destes por linha.

Observe também que não há ponto-e-vírgula após qualquer diretiva do pré-processador.

POR QUE USAR #define?

Talvez você se pergunte o que ganhamos substituindo **PI** por **3.14159** em nosso programa? Suponhamos que você tenha um programa em que a constante **3.14159** apareça muitas vezes. Suponha, ainda, que agora você queira uma precisão maior e queira trocar **3.14159** por **3.141592**. Você deverá ir passo a passo através do programa e trocar cada ocorrência que encontrar da constante. Entretanto, se você tiver definido **3.14159** como sendo **PI** na diretiva **#define**, você deverá somente fazer uma troca.

A diretiva **#define** pode ser usada não só para definir constantes, como mostra o exemplo seguinte:

```
#define ERRO printf("\nERRO.\n")
main()
{
   int i;
   printf("\nDigite um numero de 0 a 100: ");
   scanf("%d",&i);
   if(i<0 || i>100)
      ERRO;
}
```

Neste caso ERRO será substituído pela frase:

printf("\nERRo.\n)

antes da compilação.

MACROS

A diretiva **#define** tem a habilidade de usar *argumentos*. Uma diretiva **#define** com argumentos é chamada *macro* e é bastante semelhante a uma função. Eis alguns exemplos que ilustram como macros são definidas e usadas.


```
/* macroprn */
/* mostra macro, usando instrucao printf() */

#define PRN(n) printf("%.2f\n",n)

main()
{
   float num1 = 27.25;
   float num2;

   num2=1.0/3.0;
   PRN(num1);
   PRN(num2);
}
```

Quando você usar a diretiva **#define** nunca deve haver espaço em branco no identificador. Por exemplo, a instrução

#define PRN (n) printf("%.2f\n",n)

não trabalhará corretamente, porque o espaço entre PR e (n) é interpretado como o fim do identificador.

Para definir um texto "muito grande" devemos delimitar a linha anterior com uma barra invertida \ e prosseguir com a definição em outra linha.

Toda ocorrência de **PRN(n)** em seu programa é substituída por **printf("%.2f\n",n)**. O **n** na definição da macro é substituído pelo nome usado na chamada a macro em seu programa, portanto **PR(num1)** será substituído por **printf("%.2f\n",num1)**. Assim, **n** age realmente como um argumento.

Macros aumentam a clareza do código, pois permitem a utilização de nomes sugestivos para expressões.

MACROS E FUNÇÕES

Vamos modificar o programa *esfera.c* para que use uma macro ao invés da função **area()** :

```
/* esfera.c */
/* calcula a area da esfera */

/* define constante */
#define PI 3.14159
/* define macro */
#define AREA(x) ( 4 * PI * x * x)

main()
{
   float raio;

   printf("Digite o raio da esfera: ");
   scanf("%f",&raio);
   printf("A area da esfera e' %.2f",AREA(raio));
}
```

USO DE PARÊNTESES EM MACROS

O uso liberal de parênteses em macros pode provocar fracassos consideráveis.

Suponha que seu programa contém as seguintes linhas:

```
#define SOMA(x,y) x+y
.....
.....
ans = 10 * SOMA(3,4);
```

Qual é o valor que será atribuído a **ans** quando rodarmos o programa?

Você deve estar pensando que 3 será adicionado a 4, produzindo 7, e este resultado, quando multiplicado por 10 produzirá 70. ERRADO!!

O compilador substituirá SOMA(3,4) por 3+4

```
ans = 10 * 3+4
```

e o resultado será 34.

A solução é colocar parênteses envolvendo o texto todo:

```
#define SOMA(x,y) (x+y)
```

Colocando o texto entre parênteses, entretanto, você não soluciona todos os problemas. Considere o exemplo abaixo:

```
#define PRODUTO(x,y) (x*y)
.....
.....
ans = PRODUTO(2+3,4);
```

Depois do compilador efetuar a substituição, a expressão ficará:

```
ans = (2+3*4);
```

e o resultado não será o desejado.

A solução é envolver cada argumento por parênteses.

```
#define PRODUTO(x,y) ((x)*(y))
```

POR SEGURANÇA, COLOQUE PARÊNTESES ENVOLVENDO O TEXTO TODO DE QUALQUER DIRETIVA #define QUE USE ARGUMENTOS, BEM COMO ENVOLVENDO CADA ARGUMENTO.

VANTAGENS E DESVANTAGENS DO USO DE MACROS *VERSUS* FUNÇÕES

Várias funções de biblioteca C são atualmente implementadas como macros. Para ver estas macros, examine o programa *ctype.h* que acompanha o seu compilador. Certamente, estas funções poderiam estar implementadas na biblioteca padrão de funções em vez de macros. Qual é a diferença entre funções de biblioteca e macros ?

Como macros são simples substituições dentro dos programas, o seu código aparecerá em cada ponto do programa onde forem usadas. Assim, a execução do programa será mais rápida que a chamada a uma função toda vez que se fizer necessário. Em contrapartida o código do

programa será aumentado, pois o código da macro será duplicado cada vez que a macro for chamada.

Uma outra vantagem do uso de macros é a não necessidade de especificar o tipo do argumento. Por exemplo:

```
#define sqr(x) (x)*(x)

main()
{
   int i=2;
   float j=3.8;

   printf("Soma dos quadrados: %f\n",sqr(i)+sqr(j));
}
```

A macro **sqr()** pode ser usada com qualquer tipo de dado. No nosso exemplo chamamos a macro com um **int** e com um **float**.

Se uma função fosse escrita para calcular o quadrado de um número, o tipo do seu argumento deveria ser declarado. Por exemplo:

```
double sqr(x)
double x;
{
   return(x*x);
}
```

calculará o quadrado de uma variável **double**. Se desejarmos calcular o quadrado de um outro tipo de variável, precisaríamos escrever uma nova função. Uma macro é a solução eficiente quando diferentes tipos de dados podem ser usados.

É aconselhável, sempre que possível, para cada macro termos uma função equivalente numa biblioteca e fazer a escolha certa conforme o tamanho e o número de chamadas em seu programa.

A DIRETIVA #include

A diretiva **#include** causa a inclusão de um programa-fonte em outro.

Aqui está um exemplo de sua utilidade: suponha que você escreveu várias fórmulas matemáticas para calcular áreas de diversas figuras.

Você poderá colocar estas fórmulas em macros em um programa separado. No instante em que você precisar reescrevê-las para a utilização em seu programa use a diretiva **#include**.

```
#define PI 3.14159
#define AREA_CIRCULO(raio) (PI*raio*raio)
#define AREA_RETANG(base,altura) (base*altura)
#define AREA_TRIANG(base,altura) (base*altura/2)
#define AREA_ELIPSE(raio1,raio2) (PI*raio1*raio2)
#define AREA_TRAPEZ(alt,lado1,lado2) (alt*(lado1+lado2)/2)
```

Grave o programa acima como *areas.h*.

Quando você for escrever seu programa simplesmente inclua

```
#include <areas.h>
```

ou

```
#include "areas.h"
```

Dois exemplos de macros que já vimos e estão definidos no arquivo *stdio.h* são **getchar()** e **putchar()**, derivadas de duas outras funções, **getc()** e **putc()**.

```
#define getchar() getc(stdin)
#define putchar(c) putc((c),stdout)
```

A capacidade de **#define** e **#include** vai além da representação simbólica de constantes e inclusão de arquivos. Você pode até criar uma outra linguagem de programação usando **#define** e **#include**. No momento analise o programa a seguir.

```
#include "pascal.h"

program
   begin
      write("Isto e' linguagem C ??");
end
```

Humm, isto parece familiar, um pequeno programa em Pascal. O segredo está no arquivo *pascal.h* a seguir.

```
#define program main()
#define begin {
#define write(x) printf(x)
#define end }
```

OUTRAS DIRETIVAS #undef, #if, #ifdef, #ifndef, #else e #endif

Estas diretivas são geralmente usadas em grandes programas. Elas permitem suspender definições anteriores e produzir arquivos que podem ser compilados de mais de um modo.

A diretiva **#undef** suspende a mais recente definição de seu "texto".

```
#define GRANDE 3
#define ENORME 5
#undef GRANDE                   /* cancela definicao de GRANDE */
#define ENORME 10               /* ENORME redefinido como 10 */"
#undef ENORME                   /* ENORME volta a 5 */
#undef ENORME                   /* cancela definicao de ENORME */
```

FORÇANDO O COMPILADOR A USAR UMA FUNÇÃO

Suponhamos que você queira usar a função **getchar()** em vez de sua correspondente macro. Como forçar o compilador a abandonar a macro

e usar a função ? Certamente existem vários meios e o mais óbvio é não incluir o programa *stdio.h* pela diretiva **#include**.

Não incluir *stdio.h* não é a solução mais elegante para o problema, pois você pode querer usar outras informações deste arquivo. Uma solução melhor é suspender a definição da macro com a diretiva **#undef**.

As outras diretivas mencionadas permitem compilações condicionais.

```
#ifdef CORES
#include "cores.h"
#define TAB 10
#else
#include "cores1.h"
#define TAB 20
#endif
```

A diretiva **#ifdef** avalia se CORES já foi definida. Se positivo, executa todas as instruções seguintes até encontrar **#else** ou **#endif**, quando então pula para a próxima instrução após o **#endif**. Se não, executa as instruções seguintes ao **#else**.

Um dos usos da "compilação condicional" é para tornar o código mais portátil.

```
#if SYS=="IBM"
#include "ibm.h"
#endif
```

REVISÃO

1. Toda função C tem a mesma estrutura da função **main()** e oferece um meio de decompor repetidamente um problema em problemas menores.
2. Um programa grande deve ser escrito como uma coleção de funções onde cada uma não ocupa mais de uma página de comprimento e executa uma pequena tarefa do programa todo.

3. O comando **return** provoca o término imediato da execução de uma função e pode retornar, à função que chama, no máximo um valor.

4. Uma função é dita recursiva se, dentro dela, está presente uma instrução de chamada a ela própria.

5. Todas as variáveis e funções C têm dois atributos: um tipo e uma classe de armazenamento. O tipo de uma função é determinado pelo tipo do valor que ela retorna e é assumido o tipo **int** por default.

6. As classes de armazenamento em C são: **auto**, **extern**, **static** e **register**. Toda função C é da classe **extern**, isto é, é conhecida por todas as outras.

7. Existem 3 lugares, num programa C, onde você pode declarar variáveis: fora de qualquer função (serão conhecidas por todas as funções definidas depois delas), logo após a abertura da chave, indicando o início de um bloco (serão conhecidas somente dentro deste bloco) e antes da abertura da chave que inicia uma função (declarações dos argumentos da função).

8. Sempre que duas variáveis tiverem o mesmo nome, mas diferentes endereços de memória, elas não são as mesmas variáveis.

9. O pré-processador C é um programa contido no compilador que altera o programa-fonte antes da compilação.

10. Linhas normais de programa são instruções para o microprocessador; diretivas do pré-processador são instruções para o compilador.

11. A diretiva **#define** define constantes simbólicas ou macros. A diretiva **#include** inclui um programa-fonte no seu programa.

12. As diretivas **#undef**, **#if**, **#ifdef**, **#ifndef**, **#else** e **#endif** são geralmente usadas em grandes programas.

EXERCÍCIOS

1. Quais das seguintes razões são válidas para o uso de funções?

   a) Usam menos memória do que se repetirmos o mesmo código várias vezes.

   b) Rodam mais rápido.

   c) Fornecem um meio de encapsular alguma computação em uma caixa preta, que pode ser usada sem preocupação quanto a seus detalhes internos.

   d) Mantêm variáveis protegidas das outras partes do programa.

2. *Verdadeiro ou Falso:* Uma função pode ainda ser útil mesmo se você não enviar nada a ela e ela não lhe devolver qualquer informação.

3. A instrução abaixo é uma chamada correta à função **abs()** que necessita de um argumento? Por quê?

```
ans=abs(num)
```

4. *Verdadeiro ou Falso:* Para retornar de uma função, você deve usar o comando **return**.

5. *Verdadeiro ou Falso:* Você pode retornar quantos dados desejar, de uma função, ao programa que chama, usando o comando **return**.

6. A função abaixo é correta? Por quê?

```
abs(num);
{
   int num;
   if(num < 0)
      num = -num;
      return(num);
}
```

7. A função abaixo é correta? Por quê?

```
acha()
{
   int proximo;

   procura(proximo++);
   imprima();
}
```

8. Este programa é correto? Por quê?

```
main()
{
   float x,y;

   scanf("%f %f",&x,&y);
   printf("%f\n",mul(x,y));
}

float mul(a,b)
float a,b;
{
   return(a*b);
}
```

9. *Verdadeiro ou Falso:* Funções podem ser definidas dentro de outras funções, conforme as necessidades do programa.

10. *Verdadeiro ou Falso:* As variáveis habitualmente usadas em funções C são acessíveis a todas as outras funções.

11. Quais das seguintes razões são válidas para o uso de argumentos em funções?

a) Para indicar à função onde localizar ela mesma na memória.
b) Transmitir informações à função para que ela possa operá-las.
c) Para retornar informações provenientes da função ao programa que chama.
d) Para especificar o tipo da função.

12. Quais dos seguintes itens podem ser passados para uma função como argumentos?

a) constantes;
b) variáveis contendo algum valor;
c) diretivas do pré-processador;
d) expressões que depois de avaliadas assumem algum valor;
e) funções que retornam algum valor.

13. O programa seguinte é correto?

```
main()
{
   int tres=3;
   tipo(tres);
}

tipo(num)
float num;
{
   printf("%f",num);
}
```

14. Uma variável externa é definida numa declaração:

a) somente em **main()**;
b) na primeira função que a use;
c) em qualquer função que a use;
d) fora de qualquer função.

15. Uma variável externa pode ser referenciada numa instrução:

a) somente em **main()**;
b) na primeira função que a use;
c) em qualquer função que a use;
d) fora de qualquer função.

16. O que é diretiva do pré-processador?

a) Uma mensagem do compilador para o programa.
b) Uma mensagem do compilador para o linkeditor.
c) Uma mensagem do programa para o compilador.
d) Uma mensagem do programa para o microprocessador.

17. A diretiva **#define** causa a _____ de uma frase por outra.

18. Esta é uma forma correta de instrução **#define**?

```
#define CM POR POLEGADA 2.54
```

19. Nesta diretiva **#define**, qual é o "identificador" e qual é o "texto"?

```
#define EXP 2.71828
```

20. O que é macro?

a) Uma diretiva **#define** que funciona como uma função.
b) Uma diretiva **#define** que admite argumentos.
c) Uma diretiva **#define** que retorna um valor.
d) Uma diretiva **#define** que simula **scanf()**.

21. Uma variável não será usada para conter um valor que nunca muda porque:

a) o programa rodará mais devagar;
b) o programa será mais difícil de entender;
c) não é permitido semelhante tipo de uso;
d) o valor armazenado pode ser alterado.

22. O código abaixo é correto para calcular o custo de um pacote postal? Tal custo é igual a uma taxa fixa vezes a soma da altura, largura e comprimento do pacote:

```
#define SOMA3(alt,larg,comp) alt+larg+comp
.....
custo=taxa * SOMA3(a,l,c);
```

23. A diretiva **#include** causa a_____ de um programa em outro.

24. Execute o programa *esfera.c* para calcular a área do globo terrestre sabendo que seu raio é 4000 milhas.

25. Escreva um programa que solicite dois números e imprima o maior deles. Use uma função que faça a comparação dos dois números e retorne o maior deles. Esta função deve ter somente uma instrução.

26. Escreva um programa que troque o valor de duas variáveis externas. As variáveis devem ser fornecidas pelo usuário, impressas na tela, trocadas e impressas na tela novamente. Use uma função para a troca.

27. Escreva uma macro que retorne o cubo de seu argumento.

28. Escreva uma macro que toca o sinal sonoro do computador.

29. Escreva uma função que recebe um caractere como argumento e, se for uma letra minúscula, retorna-a em maiúsculo, caso contrário retorna o próprio caractere.

30. Escreva uma macro correspondente à função do exercício anterior.

31. Escreva uma macro que retorne 1 se o seu argumento é um número ímpar e 0 se for par.

32. Escreva uma macro que retorne 1 se o seu argumento for um caractere entre '0' e '9' e 0 se não for.

33. Escreva uma função recursiva, **pot(i, j)**, que aceite dois argumentos inteiros positivos e retorna **i** elevado à potência **j**. Por exemplo: pot(2,3) = 8. Use a seguinte definição:

```
pot(i,j) = i * pot(i,j-1)
```

34. Um número primo é qualquer inteiro positivo que é divisível apenas por si próprio e por 1. Escreva uma função que recebe um inteiro positivo e, se este número for primo, retorna 1, caso contrário, retorna 0.

35. A famosa conjetura de Goldbach diz que todo inteiro par maior que 2 é a soma de dois números primos. Testes extensivos foram feitos sem contudo ser encontrado um contra-exemplo. Escreva um programa mostrando que a afirmação é verdadeira para todo número par entre 700 e 1100. O programa deve imprimir cada número e os seus correspondentes primos. Use a função do exercício anterior.

MAKRON Books

CAPÍTULO 6

TECLADO E CURSOR

- *Teclado do IBM-PC*
- *As Teclas ASCII*
- *As Teclas Especiais*
- *O Código Estendido*
- *Controle do Cursor e o Programa ANSI.SYS*
- *Controle da Tela com ANSI.SYS*
- *Os Atributos de Caracteres*
- *Redefinição das Teclas de Função com ANSI.SYS*
- *Redirecionamento*

A LINGUAGEM C E O IBM-PC

Neste capítulo mudaremos o nosso enfoque sobre a linguagem C para estudarmos a interação do C com o computador IBM-PC. Isto não significa que já cobrimos tudo sobre C.

Levantaremos outros aspectos sobre a linguagem em outros capítulos.

Você agora conhece bastante a respeito de C para poder explorar algumas características do IBM-PC, a fim de, realmente, desenvolver programas em C, bem elaborados.

Cobriremos dois grandes tópicos:

Primeiro: O código ASCII estendido do IBM-PC que permite ao programador identificar as teclas de funções, teclas de controle de cursor e as teclas com combinações especiais.

Segundo: O programa de controle de teclado e tela ANSI.SYS que permite ao programador controlar a posição do cursor na tela e várias outras operações.

Quase todos os programas de aplicação, como processadores de texto, gerenciadores de banco de dados e até jogos, precisam destas capacidades para produzir uma interação de nível mais sofisticado com o usuário.

Mostraremos também como o redirecionamento pode ser usado para que os programas tenham a habilidade de ler e gravar arquivos.

O material a ser visto estenderá sua capacidade de escrever programas interessantes e poderosos.

O TECLADO

O teclado do PC é formado pelo controlador de teclado 8048, cuja principal tarefa é vigiar as teclas e informar às rotinas internas do IBM-PC sempre que uma delas for pressionada ou solta. Se alguma tecla for pressionada por mais de meio segundo, o 8048 começa a repetir a ação por

um determinado período. O modelo AT usa um controlador diferente, o 8042, que executa essencialmente as mesmas funções do 8048.

O 8048 possui ainda um "buffer" que pode armazenar até vinte caracteres até que sejam lidos. (O "buffer" do teclado é uma área temporária onde são armazenados os caracteres pressionados até que o programa os leia).

O teclado do PC, em conjunto com as suas rotinas internas, nos dá um modo de transmitirmos à memória praticamente todos os caracteres de códigos ASCII. O único código ASCII que não pode ser transmitido diretamente é o 0, pois ele é reservado para indicar caracteres não ASCII.

AS TECLAS ASCII

O teclado gera letras, números e caracteres de pontuação através do código ASCII de 1 a 255 decimal. Cada um destes caracteres é enviado como um byte pelas rotinas internas para a memória. Entretanto, existe um grande número de teclas e combinações de teclas que não são representadas por este 1 byte. Como exemplo, as teclas de função, F1 a F10, e a combinação de teclas para controle do cursor.

AS TECLAS ESPECIAIS

O IBM proporciona uma segunda coleção de 256 teclas e combinação de teclas através do uso do código estendido.

Este código consiste em 2 bytes; o primeiro sendo sempre 0 e o segundo um número indicando uma tecla particular ou combinação de teclas.

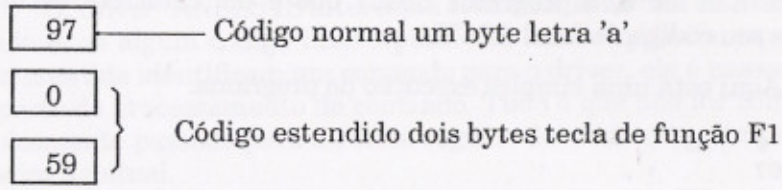
Quando uma tecla, que não pertence ao conjunto normal de caracteres, F1 por exemplo, é pressionada, ela primeiro manda um zero para o "buffer" do teclado e depois seu código específico. Assim, quando uma tecla não ASCII é pressionada, 2 caracteres são mandados.

O programa que espera ler caracteres de códigos estendido deve verificar se o primeiro caractere lido tem valor 0, e então reconhecer que

o próximo caractere é um código estendido com uma interpretação completamente diferente da do código normal.

O CÓDIGO ESTENDIDO DO TECLADO USA 2 CARACTERES, O PRIMEIRO SEMPRE TERÁ O VALOR ASCII ZERO (0).

Como nenhum caractere é representado por 0 no conjunto normal de caracteres do IBM, não há confusão. E, quando este caractere é recebido, ele sempre indica que o caractere seguinte pertence ao código estendido.



EXPLORANDO O CÓDIGO ESTENDIDO

Eis um programa que permite explorar o código estendido do teclado. Basicamente este programa imprime o código de qualquer tecla pressionada: tanto o código de um caractere normal de um byte como os códigos dos dois bytes de caractere estendido.

```
/* testecla.c */
/* imprime codigo das teclas */

main()
{
   char tecla1,tecla2;

   while((tecla1=getch())!='X')
   if(tecla1==0) {
      tecla2=getch();
      printf("%3d %3d\n",tecla1,tecla2);
   }else
      printf("%3d\n",tecla1);
}
```

Para ler o código do teclado usamos a função **getch()** que é equivalente à função **getche()**, salvo que o caractere não é impresso na tela.

Na expressão de teste de laço **while**, o programa lê o primeiro código e o compara com a letra 'X' que indica fim de laço. O corpo do laço compara este caractere com 0 e, se for 0, o programa reconhece que se trata de código estendido; então lê a segunda parte do código usando **getch()** novamente, e imprime o valor numérico das duas partes. Se o primeiro caractere não for 0, o programa deduz que é um caractere normal e imprime seu código decimal ASCII.

Aqui está uma simples execução do programa:

```
 0   59
97
 0   75
```

O primeiro código mostrado aqui é o da tecla **F1**, o segundo da tecla **'a'** e o terceiro da seta esquerda.

A tabela seguinte mostra o código estendido que pode ser obtido pressionando-se uma das teclas. A tabela apresenta o segundo byte do código; o primeiro byte é sempre 0.

Vários outros códigos podem ser acessados usando-se combinações de teclas com **ALT**, **CTRL** ou **SHIFT** e são mostrados em outra tabela.

UMA TECLA – CÓDIGO ESTENDIDO

SEGUNDO BYTE (decimal)	TECLA QUE GERA O CÓDIGO
59	F1
60	F2
61	F3
62	F4
63	F5
64	F6
65	F7
66	F8
67	F9
68	F10
71	HOME
72	UP ARROW

(continua na próxima página)

SEGUNDO BYTE (decimal)	TECLA QUE GERA O CÓDIGO
73	PgUp
75	LEFT ARROW
77	RIGHT ARROW
79	END
80	DOWN ARROW
81	PgDn
82	INS
83	DEL

DUAS TECLAS – CÓDIGO ESTENDIDO

SEGUNDO BYTE (DECIMAL)	TECLAS QUE GERAM O CÓDIGO
15	SHIFT TAB
16 A 25	ALT Q,W,E,R,T,Y,U,I,O,P
30 A 38	ALT A,S,D,F,G,H,J,K,L
44 A 50	ALT Z,X,C,V,B,N,M
84 A 93	SHIFT F1 A F10
94 A 103	CTRL F1 A F10
104 A 113	ALT F1 A F10
114	CTRL PrtSc (começa termina impressão)
115	CTRL LEFT ARROW
116	CTRL RIGHT ARROW
117	CTRL END
118	CTRL PgDn
119	CTRL HOME
120 A 131	ALT 1,2,3,4,5,6,7,8,9,0,-,=
132	CTRL PgUp

INTERPRETANDO CÓDIGO ESTENDIDO

Uma maneira comum de escrever funções C que imprimem o código estendido é usar o comando **switch**, como será mostrado no programa seguinte:

```
/* codesten.c */
/* testa codigo estendido */

main()
```

```
{
   int tecla1,tecla2;
   while((tecla1=getch())!='X')
      if(tecla1==0) {
         tecla2=getch();

         switch(tecla2) {

            case 59: printf("Tecla de funcao 1\n");break;
            case 60: printf("Tecla de funcao 2\n");break;
            case 75: printf("Seta esquerda\n");break;
            case 77: printf("Seta direita\n");break;
            default: printf("Algum outro codigo estendido\n");
         }
      }else
         printf("Codigo normal: %3d=%c\n",tecla1,tecla1);
}
```

O programa usa um comando **switch** para analisar, interpretar e imprimir o código das teclas.

CONTROLE DO CURSOR E ANSI.SYS

Quando o MS-DOS inicia a sua instalação, reconhece através das rotinas internas do IBM-PC várias coisas sobre o computador e os dispositivos a ele ligados como impressoras, dispositivos de discos, vídeo etc.

Algumas rotinas internas são acrescidas ao sistema operacional logo na sua inicialização, entretanto existem alguns dispositivos chamados "drivers" que não estão incluídos nas rotinas internas dos equipamentos e são fornecidos em arquivos separados junto com o disco de MS-DOS.

Para que o DOS possa incorporar estes arquivos ele procura sua especificação num arquivo-texto chamado CONFIG.SYS.

A família IBM-PC e outros vários computadores compatíveis com o sistema MS-DOS oferecem um dispositivo que controla a posição do cursor, a tela e ainda permite a redefinição das teclas do teclado. Este dispositivo não está incluído nas rotinas internas dos equipamentos. Está

num arquivo separado, junto ao disco do seu sistema, e é chamado ANSI.SYS.

ANSI significa American National Standards Institute e o arquivo ANSI.SYS utiliza o código padronizado, elaborado e aprovado pelo ANSI.

Este arquivo é um exemplo de "instalable device driver", isto é, uma secção de código, escrita para controlar os dispositivos de entrada e saída, que pode ser adicionada ao sistema operacional depois deste ter sido instalado.

O "driver" ANSI.SYS intercepta toda a saída na tela e a entrada do teclado e, se algum código interceptado for uma seqüência especial de caracteres que identificam um comando para o driver, ele é passado para sua parte de processamento de comando. Tudo o que não for comando é simplesmente passado para a rotina regular de vídeo que o imprime de maneira habitual.

Os comandos para o driver ANSI.SYS são identificados por uma seqüência especial de caracteres sempre começadas por um caractere "escape" ou 1B hexa seguido por um colchete esquerdo ou 5B hexa.

INSTALANDO ANSI.SYS

Para usar as facilidades oferecidas pelo driver ANSI.SYS ele deve ser instalado no sistema operacional. Para isto deve existir no disco de inicialização do sistema um arquivo chamado CONFIG.SYS que diz ao DOS quando há uma interface de dispositivo a ser carregada.

Muitos programas fazem você criar um arquivo CONFIG.SYS ou oferecem um. Se você tiver um, acrescente a linha

DEVICE = ANSI.SYS

Se não tiver, crie um

```
A>COPY CON: CONFIG.SYS
DEVICE = ANSI.SYS
<Ctrl> Z <Enter>
```

Além deste arquivo, o driver ANSI.SYS também deve estar presente no disco de inicialização.

Agora, quando você instalar seu sistema o arquivo ANSI.SYS estará incorporado ao DOS que ocupará uma memória acrescida de mais ou menos 1600 bytes.

CONTROLE DA TELA COM ANSI.SYS

O driver ANSI.SYS fornece um grupo de comandos de controle de tela que incluem mover o cursor, limpar a tela, definir os atributos de vídeo (cor, reverso, intensificado, piscante etc.), e alterar o modo texto para gráfico e vice-versa. Há comandos para guardar a posição atual do cursor, de modo que ele possa ser deslocado para imprimir alguma informação e depois retornar à posição original.

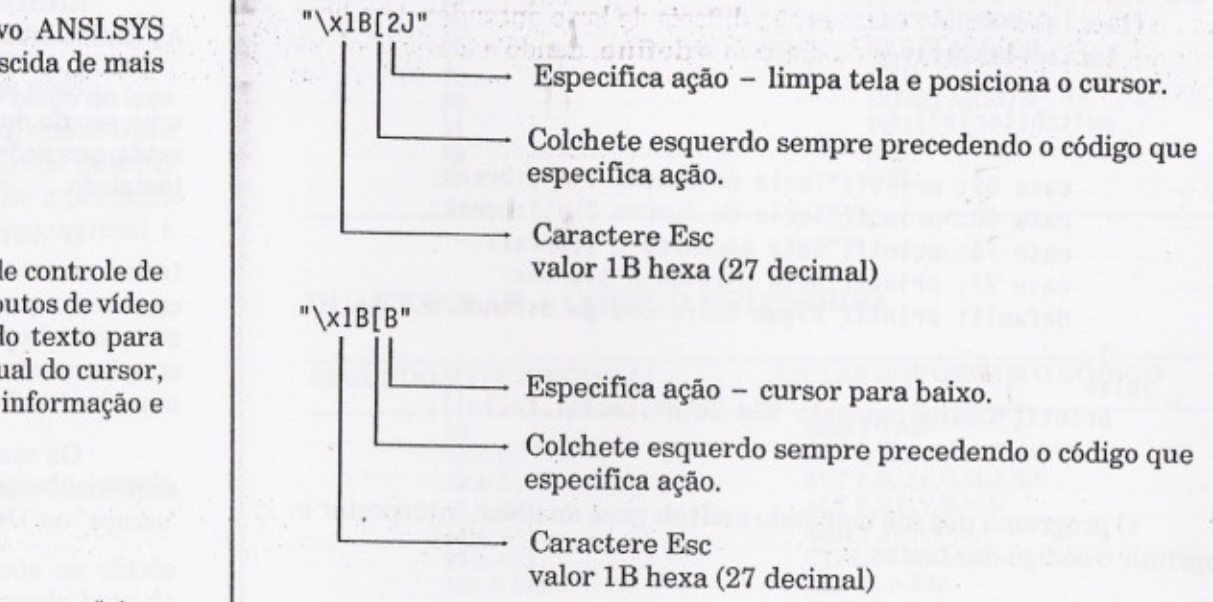
CONTROLE DO CURSOR COM ANSI.SYS

O controle do cursor é obtido através de "seqüências escape", isto é, o caractere **1B hexa** (*caractere escape*) seguido pelo colchete esquerdo '[' e acrescido de uma letra ou conjunto de letras, conforme a tabela seguinte:

CÓDIGOS DE CONTROLE DE CURSOR – ANSI.SYS

CÓDIGO	EFEITO
"\x1B[2J"	Limpa tela e posiciona o cursor no canto esquerdo
"\x1B[K"	Limpa até o final da linha
"\x1B[A"	Cursor uma linha acima
"\x1B[B"	Cursor uma linha abaixo
"\x1B[C"	Cursor uma coluna à direita
"\x1B[D"	Cursor uma coluna à esquerda
"\x1B[%d;%df"	Cursor na linha e coluna especificada
"\x1B[s"	Salva posição do cursor
"\x1B[u"	Restaura posição do cursor
"\x1B[%dA"	Cursor número de linhas especificadas para cima
"\x1B[%dB"	Cursor número de linhas especificadas para baixo
"\x1B[%dC"	Cursor número de colunas especificadas à direita
"\x1B[%dD"	Cursor número de colunas especificadas à esquerda

Eis um programa para mostrar o uso de controle do cursor através de duas seqüências:




```
/* diag.c */
/* move o cursor na diagonal */
main()
{
   printf("\x1B[2J");
   while(getche()!='.')
      printf("\x1B[B");
}
```

Uma execução do programa provocaria a seguinte saída em tela:

```
M
  A
    R
      I
        A
          N
            A
              .
```

USANDO #define PARA DEFINIR SEQÜÊNCIA ESCAPE

As seqüências de escapes são difíceis de ler e entender. Um bom costume é defini-las através da diretiva **#define**, dando a elas um nome mais significativo.

Vamos reescrever *diag.c*:

```
/* diag.c */
/* move o cursor na diagonal */
#define C_DESCE "\x1B[B"
main()
{
   while(getche()!='.')
      printf(C_DESCE);
}
```

Naturalmente que movimentar o cursor para baixo é uma das possibilidades que podemos ter.

A lista completa de comandos está no manual *IBM DOS Technical Reference*. Mostraremos vários destes comandos neste capítulo.

CONTROLE DO CURSOR ATRAVÉS DO TECLADO

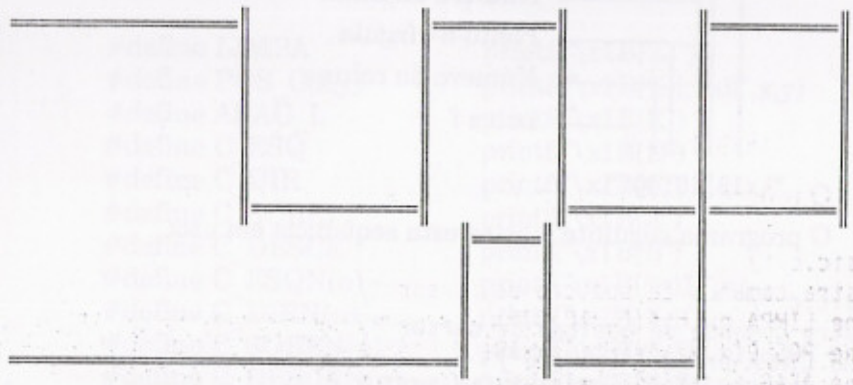
Agora, como já conhecemos o código estendido das teclas e como controlar o cursor, podemos juntar as duas coisas para controlar o cursor através das setas. Aqui está um programa que permite a você desenhar simples figuras na tela:

```
/* desenha.c */
/* move o cursor pela tela */
#define LIMPA "\x1B[2J"
#define C_ESQ "\x1B[D"
#define C_DIR "\x1B[C"
#define C_SOBE "\x1B[A"
#define C_DESCE "\x1B[B"
#define SETA_ESQ 75
#define SETA_DIR 77
#define SETA_SOBE 72
```



```
#define SETA_DESCE 80
#define DUPLO_H 205
#define DUPLO_V 186
main()
{
   int tecla;
   printf(LIMPA);
   while((tecla=getch())==0) {
      tecla=getch(); /* segundo byte */
      switch(tecla) {
      case SETA_ESQ:
         printf(C_ESQ);putch(DUPLO_H);break;
      case SETA_DIR:
         printf(C_DIR);putch(DUPLO_H);break;
      case SETA_SOBE:
         printf(C_SOBE);putch(DUPLO_V);break;
      case SETA_DESCE:
         printf(C_DESCE);putch(DUPLO_V);break;
      }
      printf(C_ESQ);
   }
}
```

Um exemplo de execução é:



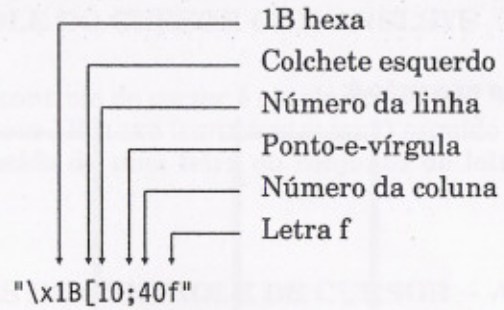
O programa inicia limpando a tela e posicionando o cursor no seu canto esquerdo, e então espera até que seja pressionada qualquer das quatro setas para começar o desenho.

O programa termina quando uma tecla de código normal é pressionada.

Este programa usa um comando **switch** para interpretar a tecla pressionada. Para cada uma das quatro teclas de controle do cursor, o cursor é primeiro movido na direção correspondente usando **printf()**; e o caractere gráfico apropriado é impresso usando a função **putch()**. Esta função é análoga à **getch()**, salvo que imprime o caractere na tela em vez de lê-lo do teclado.

MOVENDO O CURSOR PARA UMA POSIÇÃO ESPECÍFICA

Podemos tanto mover o cursor uma linha ou coluna por vez, como movê-lo diretamente para qualquer localização da tela usando uma seqüência de escape um pouco mais complexa. Esta seqüência é iniciada pela seqüência de escape usual '\x1B', seguida por colchete esquerdo e por um número representando a linha onde queremos que o cursor seja posicionado, um ponto-e-vírgula, um número representando a coluna desejada, e finalmente a letra minúscula 'f'.



O programa seguinte mostra esta seqüência em uso:

```
/* posic.c */
/* mostra comando de posicao de cursor */
#define LIMPA printf("\x1B[2J")
#define POS_C(x,y) printf("\x1B[%d;%df",x,y)
#define APAG_L printf("\x1B[K")
main()
{
   int lin=1,col=1;
   LIMPA;
   while(1) {
      POS_C(23,1);
      APAG_L;
      printf("Digite linha e coluna na forma 10,40 : ");
      scanf("%d,%d",&lin,&col);
      POS_C(lin,col);
      printf("*(%d,%d)",lin,col);
   }
}
```


Este programa limpa a tela com a seqüência "**\x1B[2J**" e entra no laço **while**, onde espera que o usuário entre com um par de coordenadas.

A instrução:

POS_C(23,1);

fixa a linha 23 para a entrada do par de coordenadas pelo usuário.

A seqüência escape "**\x1B[K**" apaga desde a posição do cursor até o fim da linha; finalmente é impresso um asterisco na posição fornecida e o número da linha e coluna correspondente.

Uma execução do programa terá a seguinte saída:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
1
2                                        *(2,35)
3              *(3,15)
4                                                *(4,40)
5     *(5,5)
6                                *(6,30)
7                    *(7,20)
8
9          *(9,10)
0*(10,2)
1
2
3
4
5
6
7
8
9
0
1
2
3 Digite linha e coluna na forma 10,40: 10,1
4
```

ATRIBUTOS DE CARACTERES

Todo caractere impresso na tela é guardado na memória do computador em 2 bytes. Um byte contém o código normal do caractere e o outro contém o seu atributo.

O "atributo" de um caractere é a sua aparência: piscante, intensificado, sublinhado, reverso ou combinações de atributo; preto em branco em vez de vídeo normal branco em preto.

TODO CARACTERE É GUARDADO NA MEMÓRIA COMO 2 BYTES: UM PARA O SEU CÓDIGO ASCII E O OUTRO PARA O SEU ATRIBUTO.

O atributo de um caractere pode ser escolhido usando seqüência de escape de ANSI.SYS. A seqüência, seguinte ao caractere escape usual e ao colchete, consiste em um número seguido pela letra 'm'. Aqui está a lista de números e o seu efeito em vídeo monocromático:

0	Cancela atributos. Vídeo normal branco sobre preto.
1	Intensificado.
4	Sublinhado.
5	Piscante.
7	Reverso (preto sobre branco).
8	Invisível (preto sobre preto).

Quando um atributo particular é impresso, todos os caracteres impressos em seguida terão o efeito daquele atributo até que seja escolhido algum outro.

O próximo programa mostra o uso de atributos de caracteres:

```
/* atrib.c */
/* troca atributos */


#define NORMAL "\x1B[0m"
#define INTEN "\x1B[1m"
#define SUBL "\x1B[4m"
#define PISCA "\x1B[5m"
#define REV "\x1B[7m"
```



```
main()
{
   printf("\n\n");
   printf("Normal %s Piscante %s Normal \n\n",PISCA,NORMAL);
   printf("Normal %s Negrito %s Normal \n\n",INTEN,NORMAL);
   printf("Normal %s Sublinhado %s Normal \n\n",SUBL,NORMAL);
   printf("Normal %s Reverso %s Normal \n\n",REV,NORMAL);
   printf(" %s %s Reverso e Piscante %s",PISCA,REV,NORMAL);
}
```

Vídeos com placas EGA ou CGA não possuem o atributo sublinhado, mas têm vários outros para definição de cores. Consulte o seu manual do DOS para mais detalhes.

Agora vamos juntar os conhecimentos que temos da linguagem C e do controle da tela e escrever um programa um pouco mais sofisticado. O programa *tiroalvo.c* simula um jogo de tiro ao alvo.

Antes de apresentar a listagem do programa, vamos criar um arquivo, para posterior inclusão, com o nome *ansi.h* do seguinte modo:

O ARQUIVO ANSI.H

```
#define LIMPA           printf("\x1B[2J")
#define POS_C(x,y)      printf("\x1B[%d;%df",x,y)
#define APAG_L          printf("\x1B[K")
#define C_ESQ           printf("\x1B[D")
#define C_DIR           printf("\x1B[C")
#define C_SOBE          printf("\x1B[A")
#define C_DESCE         printf("\x1B[B")
#define C_ESQN(n)       printf("\x1B[%dD",n)
#define C_DIRN(n)       printf("\x1B[%dC",n)
#define C_SOBEN(n)      printf("\x1B[%dA",n)
#define C_DESCEN(n)     printf("\x1B[%dB",n)
#define NORMAL          printf("\x1B[0m")
#define INTEN           printf("\x1B[1m")
#define SUBL            printf("\x1B[4m")
#define PISCA           printf("\x1B[5m")
#define REV             printf("\x1B[7m")
```

O PROGRAMA TIROALVO.C

Eis a listagem do nosso tiro ao alvo:

```
/* tiroalvo.c */
/* usa controle de tela ANSI.SYS */
#include <ansi.h>
#define BEEP printf("\x7")
#define SOBE 72
#define DESCE 80

main()
{
   int municao=15,disparos=0,pontos=0,i,j;

   LIMPA;
   POS_C(12,26);printf("*** TIRO AO ALVO ***");
   POS_C(15,15);
   printf("UTILIZE AS SETAS PARA MOVIMENTAR O ATIRADOR");
   POS_C(18,21);printf("E QUALQUER OUTRA PARA DISPARAR.");
   POS_C(23,20);printf("PRESSIONE <ENTER> PARA CONTINUAR");
   getch();LIMPA;
   POS_C(24,4);REV;printf("MUNICAO: ");
   POS_C(24,31);printf("DISPAROS: ");
   POS_C(24,59);printf("PONTOS: ");
   NORMAL;
   atualiza(municao,disparos,pontos);
   while(municao) {
      linha_dir();              /* imprime linha direita */
      i= alvo();                /* imprime alvo */
      j=22;posin(j);            /* posicao inicial */

      while(getch()==0) {       /* usuario joga ate atirar */
         switch(getch()) {
            case SOBE: j=(j<2) ? 22:j-1;break;
            case DESCE: j=(j>21) ? 1:j+1;break;
         }
         posin(j);
      }
```



```
      disparos++; municao--;
      dispara(j);               /* simula disparo */
      if(j==i+1) {              /* acertou */
         bum(i+1);
         pontos++;
         municao=((pontos%5)==0) ? municao+2:municao;
      }
      apagalvo(i);
      atualiza(municao,disparos,pontos);
   }
   POS_C(10,25);printf("* * * F I M * * *");
   getch();
}

atualiza(m,d,p)
int m,d,p;
{
   REV;PISCA;
   POS_C(24,13);printf(" %02d",m);
   if(!m) BEEP;
   POS_C(24,40);printf(" %03d",d);
   POS_C(24,66);printf(" %03d",p);
   NORMAL;
}

posin(j)
int j;
{
   REV;
   POS_C(j,78);printf("\xDE");
   NORMAL;
}

linha_dir()
{
   int i;
   for(i=1;i<23;i++) {
      POS_C(i,78);
      printf("\xDE");
   }
}
```

```
alvo()
{
   int i;
   i=rand()%21;
   POS_C(i,1);printf("\xDE");
   POS_C(i+2,1);printf("\xDE");
   return(i);
}

dispara(j)
int j;
{
   int i;
   for(i=76;i>0;i-=2) {
      POS_C(j,i);printf("\xDC");
      POS_C(j,i);printf(" ");
   }
}

bum(j)
int j;
{
   int i;
   for(i=0;i<15;i++) {
      POS_C(j,i);printf(" BUUUMMM !!!");
      POS_C(j,i);printf("             ");
   }
}

apagalvo(j)
int j;
{
   POS_C(j,1);printf(" ");
   POS_C(j+2,1);printf(" ");
}
```


O programa fornece uma interação com o usuário que será descrita em vários passos:

Passo 1: Limpa a tela e apresenta explicações sobre seu uso aguardando até que um tecla seja pressionada.

Passo 2: Imprime a tela inicial. A função **atualiza()** é chamada para imprimir a munição disponível, o número de disparos efetuados e o número de pontos. Se a munição for 0 um sinal sonoro é dado e o programa termina.

Passo 3 É o início do programa propriamente dito, composto por um laço **while** que controla a munição. Este laço inicia sua execução imprimindo uma linha na coluna 78 através da função **linha_dir()** e o alvo que é composto por dois caracteres e é impresso pela função **alvo()** numa linha aleatória e na coluna 1.

Passo 4: O programa entra no laço **while** aninhado que aguarda o usuário jogar e atualiza a posição do cursor pela função **posin()**.

Passo 5: O usuário já jogou. Incrementa disparos e decrementa munição e simula um disparo através da função **dispara()**. Se o alvo foi acertado, a função **bum()** é chamada e a variável pontos é incrementada. A cada vez que o usuário faz mais 5 pontos, a munição é acrescida de 2. O alvo é apagado pela função **apagalvo()** e a linha de dados é atualizada pela chamada a função **atualiza()**.

REDEFINIÇÃO DE TECLAS USANDO ANSI.SYS

Vamos voltar a nossa atenção para uma outra habilidade fornecida pelo arquivo *ANSI.SYS*: atribuir diferentes textos para as teclas de função.

A atribuição de textos às teclas de função capacita-lhe configurar seu teclado conforme suas necessidades mais rotineiras. Por exemplo, se você escreveu vários programas em C, pode querer listar o nome de seus fontes na tela com o seguinte comando:

```
C> dir *.c
```

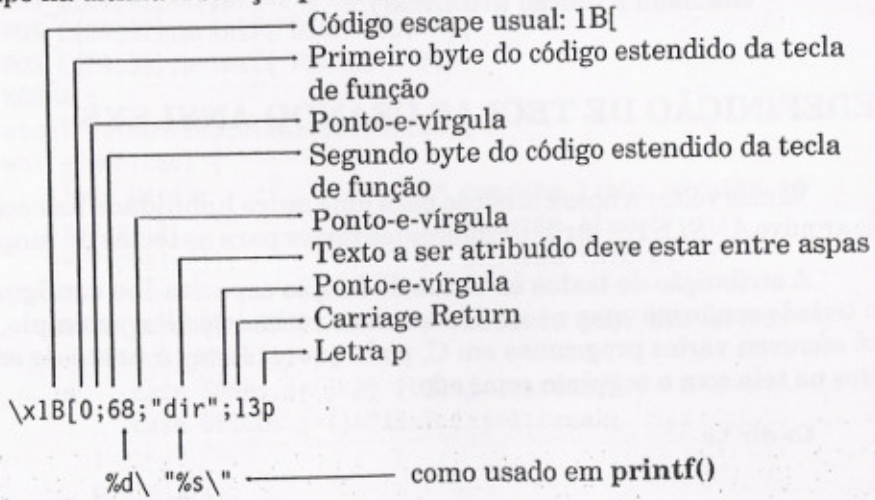
Este texto pode ser atribuído a uma tecla de função. Assim, se você precisar dele várias vezes, deverá somente pressionar a tecla de função reprogramada sem ter de redigitá-lo. A vantagem disto será bem maior para os textos maiores, como:

type \word\1987\marco\receita.dat

Aqui está um programa que redefine a tecla de função que você escolher com o texto "dir *.c" :

```
/* progfunc.c */
/* atribui um texto a tecla de funcao escolhida */

main()
{
   int tecla;

   printf("Digite o numero da tecla de funcao: ");
   scanf("%d",&tecla);
   printf("\x1B[0;%d;\"%s\";13p",tecla+58,"dir *.c");
}
```

Sem dúvida, a parte mais difícil deste programa é a seqüência de escape na última instrução **printf()**.



Como aspas não podem ser usadas num texto em C, elas devem ser representadas por (\").

Visto que queremos usar uma variável para representar o código da tecla de função, ela deve ser **tecla+58**.

O uso de ANSY.SYS não é a maneira mais rápida de movimentar o cursor, e requer que o arquivo ANSY.SYS esteja presente na memória.

Existem maneiras mais rápidas e convenientes de se obter as capacidades oferecidas por ANSY.SYS; entretanto, são um pouco mais difíceis de serem programadas. No Volume II deste livro estudaremos como fazê-lo.

REDIRECIONAMENTO

Os sistemas operacionais PC-DOS, MS-DOS, UNIX e outros, quando entram em operação, inicializam dois dispositivos acoplados ao seu micro: o teclado, tido como entrada padrão, e o vídeo, tido como saída padrão.

Por "default", um programa em C terá seus dados digitados através do teclado e suas saídas impressas em vídeo.

Você pode informar ao DOS que altere a entrada e/ou saída padrão para qualquer dispositivo acoplado ao seu micro. Este processo é chamado "redirecionamento". Por exemplo: você pode informar ao DOS que o seu programa pegará os dados de entrada de um arquivo em disco ao invés do teclado.

Há duas maneiras de fazer um programa trabalhar com arquivos em disco. Uma é o uso explícito de funções que abrem, fecham, lêem ou gravam em arquivos. Outra é a de usar redirecionamento em programas desenhados para trabalhar com teclado e tela. O redirecionamento é mais limitado em vários aspectos, mas muito mais simples de usar.

Em outras palavras, o DOS assume certos periféricos de entrada e saída. O redirecionamento permite trocar os periféricos assumidos.

A IMPRESSÃO PODE SER REDIRECIONADA PARA UM ARQUIVO AO INVÉS DA TELA E A ENTRADA DE UM PROGRAMA PODE SER REDIRECIONADA DE UM ARQUIVO AO INVÉS DO TECLADO.

IMPRESSÃO REDIRECIONADA

Considere o programa seguinte:

```
/* espelho.c */
/* ecoa a entrada na tela */

main()
{
   char ch;
   do{
      ch=getch();
      printf("%c",ch);
   } while(ch != 'X');
}
```

Quando executamos este programa, a função **printf()** causa a impressão em tela dos caracteres datilografados no teclado até que a tecla **'X'** seja pressionada.

Vamos ver o que ocorre quando chamamos o programa pelo DOS usando redirecionamento:

C>espelho > arq.txt

O símbolo **>** é um operador de redirecionamento. Ele causa a criação de um novo arquivo em disco chamado *arq.txt* que conterá a sua entrada de dados. Quando você chama o programa e datilografa alguma coisa, nada aparece na tela. A sua entrada foi guardada num arquivo chamado *arq.txt*.

Para ver o arquivo criado, digite o seguinte comando:

C>type arq.txt

O operador (**>**), conecta um programa executável com um arquivo-texto. Ele não pode ser usado para conectar um arquivo-texto com outro

ou um programa com outro. O nome do programa executável deve estar à esquerda do operador, e o nome do arquivo-texto à direita.

A entrada não poderá ser redirecionada para mais de um arquivo usando este operador.

O DOS predefine alguns nomes para os periféricos. Um deles é PRN para a impressora. A saída pode ser redirecionada para a impressora usando o seu nome. Por exemplo:

C>espelho > PRN

Tudo que você datilografar será impresso na impressora toda vez que você pressionar **[Return]**.

Tome o programa do Capítulo 2 que imprime uma árvore de natal e digite o seguinte comando:

C>ARVNATAL > PRN

e você verá sua árvore sendo impressa pela impressora.

INDICANDO FIM DE ARQUIVO

Nós usamos o caractere **X** para terminar a entrada no programa *espelho.c* e através do comando TYPE podemos notar que este caractere estará presente no arquivo.

Como não queremos este caractere impresso no nosso arquivo, vamos modificar o programa e colocar o caractere **'\x1A'** que pode ser obtido pressionando-se [Ctrl][Z] e indica fim de arquivo.

```
/* transfer.c */
/* ecoa a entrada */
/* para ser usado com redirecionamento */

main()
{
   char ch;

   do {
```

```
    ch=getch();
    printf("%c",ch);
  } while(ch != '\x1A')
}
```

O CARACTERE '\x1A' REPRESENTA FIM DE ARQUIVO.

ENTRADA DE DADOS REDIRECIONADA

Podemos redirecionar a entrada de dados do teclado para que o programa a leia de um arquivo.

Digite o seguinte comando:

C>transfer < arq.txt

O símbolo < é o segundo operador de redirecionamento. Ele causa a leitura dos dados do arquivo *arq.txt* pelo programa *transfer*. O programa *transfer* não sabe que a entrada vem de um arquivo em vez do teclado; ele simplesmente lê da entrada padrão indicada pelo DOS, que agora é o arquivo *arq.txt*.

OS DOIS CAMINHOS DE UMA VEZ

Os redirecionamentos da entrada e da saída podem ser usados juntos.

Para mostrar este processo vamos escrever dois programas, um para codificar uma mensagem e outro para decodificá-la.

```
/* code.c */
/* codifica um arquivo */
/* para usar com redirecionamento */
#define CTRL_Z '\x1A'
main()
{
   char ch;
```



```
   while ((ch=getch()) != CTRL_Z)
      putch(ch+1);
   putch(CTRL_Z);
}
```

Execute o programa *transfer.c* e gere o arquivo *arq1.txt* contendo a sua mensagem, usando redirecionamento.

Agora, com o arquivo *arq1.txt* em disco digite:

C>type arq1.txt

e para codificá-lo, vamos redirecionar a entrada para *arq1.txt* e a saída para *arq2.txt*.

C>code < arq1.txt > arq2.txt

Para ver o arquivo codificado digite:

C>type arq2.txt

Vamos agora escrever um programa para decodificar um arquivo codificado por *code.c*.

```
/* decode.c */
/* decodifica arquivo codif. por code.c */
/* para ser usado com redirecionamento */

#define CTRL_Z '\x1A'

main()
{
   char ch;

   while ((ch=getch()) != CTRL_Z)
      putch(ch-1);
   putch(CTRL_Z);
}
```

Digite:

C>decode < arq2.txt > arq3.txt

Outros operadores DOS podem ser encontrados no manual do seu sistema operacional.

REVISÃO

1. O conjunto normal de caracteres é formado pela tabela ASCII que consiste em 256 códigos de 1 byte cada um.
2. O código estendido do IBM oferece uma segunda coleção de 256 caracteres que consistem em 2 bytes cada um, onde o primeiro sempre terá o valor ASCII zero.
3. O arquivo ANSI.SYS é fornecido junto ao seu sistema operacional e pode ser incorporado ao DOS através do arquivo-texto CONFIG.SYS.
4. ANSI.SYS é um exemplo de dispositivo instalável que, quando acrescido ao DOS, permite o controle da posição do cursor, da tela e do teclado.
5. A palavra ANSI significa American National Standards Institute e o arquivo ANSI.SYS utiliza o código padronizado elaborado e aprovado pelo ANSI.
6. O redirecionamento é um meio de alterar os dispositivos tidos como "default" pelo DOS.
7. A impressão de um programa pode ser redirecionada para outro dispositivo que não a tela, e a entrada de dados pode ser redirecionada de outro dispositivo que não o teclado.

EXERCÍCIOS

1. A finalidade do código estendido do teclado é:
   - *a)* ler caracteres de línguas estrangeiras;
   - *b)* ler caracteres datilografados com a combinação de uma tecla normal com [Alt] ou [Ctrl];
   - *c)* ler teclas de função e teclas de controle de cursor;
   - *d)* ler caracteres gráficos.

2. Quantos códigos estendidos existem (incluindo os que não são usados)?
3. Quantos bytes são usados para representar um código estendido do teclado?
4. *Verdadeiro ou Falso:* os códigos estendidos do teclado representam somente teclas únicas como F1.
5. Qual dos seguintes códigos representa a tecla F1?
   - *a)* 97
   - *b)* 1 78
   - *c)* '\xDB'
   - *d)* 0 59
6. ANSI.SYS é:
   - *a)* um raro arquivo infectado;
   - *b)* um arquivo instalável para controlar os dispositivos de entrada e saída;
   - *c)* um arquivo que permite a utilização do código estendido do teclado e as capacidades de movimentação do cursor;
   - *d)* um arquivo que sempre procura pelo DOS quando inicia sua execução.
7. CONFIG.SYS é:
   - *a)* um arquivo que sempre procura pelo DOS quando inicia sua execução;
   - *b)* um arquivo que contém instruções para modificar o DOS;
   - *c)* um arquivo que pode pedir ao DOS para que instale ANSI.SYS;
   - *d)* nenhuma das anteriores.
8. Descreva de que um sistema necessita antes que ANSI.SYS possa ser usado.
9. Todas as seqüências de escapes de ANSI.SYS começam por:
   - *a)* '\x['
   - *b)* '['
   - *c)* '\x1B'
   - *d)* '\x1B['

10. A seqüência de escapes que limpa a tela é _____.

11. *Verdadeiro ou Falso:* O cursor pode ser movido para qualquer localização da tela somente se incrementarmos uma linha ou uma coluna.

14. *Verdadeiro ou Falso:* Usando seqüências de escapes de ANSI.SYS, qualquer frase pode ser atribuída a qualquer tecla de função.

17. Redirecionamento é:

a) enviar a saída do programa para algum lugar sem ser a tela;
b) obter um programa de algum lugar sem ser de arquivos .EXE;
c) obter a entrada de um programa de algum lugar sem ser do teclado;
d) substituir a entrada ou saída padrão por outro periférico.

18. Descreva o que acontecerá após esta instrução:

```
C>prog1 < f1.c > f2.c
```

19. Escreva um programa que solicita ao usuário a digitação de uma frase, ecoando-a na tela. Se o usuário pressionar a seta esquerda, o programa deve apagar um caractere à esquerda do cursor, assim toda a frase pode ser apagada um caractere por vez.

20. Escreva um programa que lê um programa-fonte em C usando redirecionamento, e determina se o programa está com as chaves balanceadas; isto é, se tem o mesmo número de chaves direita e esquerda.

```
C>chaves < prog.c
ERRO: chaves não balanceadas
C>
```

21. Escreva um programa que copia um programa-fonte em C em outro arquivo em disco, trocando toda ocorrência de letras minúsculas por maiúsculas. Use redirecionamento.

22. Escreva um programa que copia um programa-fonte em C em outro arquivo em disco, trocando todo caractere de tabulação pelos espaços correspondentes. Use redirecionamento.

MAKRON Books

CAPÍTULO 7

MATRIZES E STRINGS

- *Matrizes*
- *Checando Limites*
- *Mais de uma Dimensão*
- *Matrizes como Argumento de Funções*
- *Chamada por Valor e por Referência*
- *A Ordenação Bolha*
- *Strings*
- *As Funções para Manipulação de Strings*
  gets( ) puts( ) strlen( ) strcat( ) strcmp( ) strcpy( )

MATRIZES

Uma matriz é um tipo de dado usado para representar uma certa quantidade de variáveis de valores homogêneos.

Imagine o seguinte problema: calcular a média das notas da prova do mês de junho de 5 alunos. Você poderia declarar:

```
int nota0,nota1,nota2,nota3,nota4;
```

e o programa deverá ler os valores separadamente:

```
printf("Digite a nota do aluno 0: ");
scanf("%d",&nota0);
printf("Digite a nota do aluno 1: ");
scanf("%d",&nota1);
......
......
......
printf("Digite a nota do aluno 4: ");
scanf("%d",&nota4);
```

Imagine agora se você pretendesse encontrar a média aritmética das notas de uma classe de 50 alunos ou da escola toda com 2000 alunos? Seria uma tarefa bastante volumosa!

É evidente que precisamos de uma maneira conveniente para referenciar tais coleções de dados similares. Matriz é o tipo de dado oferecido por C para este propósito.

Uma matriz é uma série de variáveis do mesmo tipo referenciadas por um único nome, onde cada variável é diferenciada através de um número chamado "subscrito" ou "índice". Colchetes são usados para conter o subscrito.

A declaração

```
int notas[5];
```

aloca memória para armazenar 5 inteiros e anuncia que notas é uma matriz de 5 membros ou "elementos".

Vamos escrever um programa que soluciona o nosso problema usando uma matriz.

```
/* notas.c */
/* média das notas de 5 alunos */
main()
{
   int notas[5];
   int i,soma;
   for(i=0;i<5;i++) {
      printf("Digite a nota do aluno %d: ",i);
      scanf("%d",&notas[i]);
}
   soma=0;
   for(i=0;i<5;i++)
      soma=soma+notas[i];
   printf("Media das notas: %d.",soma/5);
}
```

Eis um exemplo da execução do programa:

```
Digite a nota do aluno 0: 75
Digite a nota do aluno 1: 55
Digite a nota do aluno 2: 60
Digite a nota do aluno 3: 80
Digite a nota do aluno 4: 90
Media das notas: 70.
```

O programa apresenta várias construções novas que analisaremos passo a passo.

DECLARAÇÃO DA MATRIZ

Em C, as matrizes precisam ser declaradas, como quaisquer outras variáveis, para que o compilador conheça o tipo de matriz e reserve espaço de memória suficiente para armazená-la.

Os elementos da matriz são guardados numa seqüência contínua de memória, isto é, um seguido ao outro.

O que diferencia a declaração de uma matriz da declaração de qualquer outra variável é a parte que segue o nome, isto é, os pares de colchetes ([e]) que envolvem um número inteiro, que indica ao compilador o tamanho da matriz.

```
int notas[5];
```

A palavra **int** declara que todo elemento da matriz é do tipo **int**, **notas** é o nome dado à matriz e **[5]** indica que a nossa matriz terá 5 elementos do tipo "int". Por definição uma matriz é composta por elementos de um único tipo.

REFERENCIANDO UM ELEMENTO DA MATRIZ

Uma vez declarada a matriz, precisamos de um modo de referenciar seus elementos individualmente. Isto é feito através do número entre colchetes seguindo o nome da matriz.

Observe que este número tem um significado diferente quando referencia um elemento da matriz e na declaração da matriz, onde indica o seu tamanho.

Quando referenciamos um elemento da matriz este número especifica a posição do elemento na matriz.

Os elementos da matriz são sempre numerados por índices iniciados por 0 (zero).

O elemento referenciado pelo número 2

```
notas[2]
```

não é o segundo elemento da matriz e sim o terceiro, pois a numeração começa em 0. Assim o último elemento da matriz possui um índice, uma unidade menor que o tamanho da matriz.

Em nosso programa utilizamos uma variável inteira, **i**, como índice da matriz. Esta possibilidade torna as matrizes verdadeiramente úteis.

```
notas[i]
```


ARMAZENANDO DADOS NA MATRIZ

A seção do código que coloca dados na matriz é:

```
for(i=0;i<5;i++){
printf("Digite a nota do aluno %d:",i);
scanf("%d",&notas[i]);
}
```

O laço **for** causa um processo de solicitação ao usuário e leitura das notas, repetido 5 vezes.

Na instrução **scanf()** nós usamos o operador (**&**) junto ao elemento da matriz (**&notas[i]**) pois **notas[i]** é uma variável como qualquer outra (**&num**, como exemplo).

LENDO DADOS DA MATRIZ

Vamos analisar a seção do código que lê os dados já armazenados nos elementos da matriz:

```
soma=0;
for(i=0;i<5;i++)
   soma = soma + notas[i];
printf("Media das notas: %d.",soma/5);
```

O laço **for** adiciona os valores contidos nos elementos da matriz em uma variável chamada soma. Quando todas as notas forem adicionadas, o resultado é então dividido pelo número de elementos (5), para encontrar a média.

USANDO OUTROS TIPOS DE VARIÁVEIS

Agora que você está familiarizado com o fato de uma matriz ser composta por uma série de elementos de um dado tipo, podemos então escolher qualquer tipo de dado para a nossa matriz.

Suponhamos que você queira que as notas dos seus alunos fiquem no intervalo de 0 a 10 e que não seja desprezada a parte fracionária. A solução seria usar uma matriz do tipo **float**.

```
/* fnotas.c */
/* media das notas de 5 alunos */
main()
{
   float notas[5];
   float soma;
   int i;

   for(i=0;i<5;i++) {
      printf("Digite a nota do aluno %d: ",i);
      scanf("%f",&notas[i]);
   }
   soma=0.0;
   for(i=0;i<5;i++)
      soma=soma+notas[i];
   printf("Media das notas: %.2f",soma/5.0);
}
```

O programa *fnotas.c* opera da mesma forma que *notas.c*, exceto no fato de poder aceitar números com ponto decimal e calcular a média com precisão maior.

Eis um exemplo da execução do programa:

```
Digite a nota do aluno 0: 7.5
Digite a nota do aluno 1: 5.5
Digite a nota do aluno 2: 6.0
Digite a nota do aluno 3: 8.0
Digite a nota do aluno 4: 9.0
Media das notas: 7.20.
```

Observe que a declaração da matriz notas e da variável soma como ponto flutuante acarreta a alteração do formato especificado em **scanf()** e **printf()**.

Em programação sempre devemos tentar otimizar um programa depois do seu desenvolvimento inicial. É o que faremos com *fnotas.c*.

Preste atenção às modificações:

```
/* flnotas1.c */
/* media das notas de 5 alunos */

main()
{
   float notas[5],soma;
   int i;

   for(i=0,soma=0.0;i<5;soma+=notas[i++]){
      printf("Digite a nota do aluno %d: ",i);
      scanf("%f",&notas[i]);
   }
   printf("Media das notas: %.2f",soma/5.0);
}
```

Esta versão é bem mais compacta que a anterior, e o mais importante é a eliminação de um laço **for**, o que tornará o programa mais rápido.

LENDO UM NÚMERO DESCONHECIDO DE ELEMENTOS

Nos exemplos anteriores utilizamos um número fixo de notas. Como faríamos se não conhecessemos de antemão quantos itens entrariam na matriz ?

O programa seguinte aceita até 40 notas e pode ser facilmente modificado para aceitar qualquer número de notas.

```
/* notas2.c */
/* media de um numero arbitrario de notas */

#define LIM 40

main()
{
   float notas[LIM],soma=0.0;
```

```
   int i=0;
   do {
      printf("Digite a nota do aluno %d: ",i);
      scanf("%f",&notas[i]);
      if(notas[i]>0)
         soma+=notas[i];
   } while(notas[i++]>0);

   printf("Media das notas: %.2f",soma/(i-1));
}
```

A seguir está um exemplo da execução do programa:

```
Digite a nota do aluno 0: 71.3
Digite a nota do aluno 1: 80.9
Digite a nota do aluno 2: 89.2
Digite a nota do aluno 3: 0
Media das notas: 80.5.
```

O laço **for** foi substituído pelo laço **do-while**. Este laço repete a solicitação da nota ao usuário, armazenando-a na matriz **notas[]**, até que seja fornecido um valor menor ou igual a zero (0). Obviamente este não é um programa para alunos não aplicados.

Quando o último item for digitado, a variável **i** terá alcançado um valor acima do valor total de itens. Isto é verdadeiro, pois é contado o zero ou o número negativo, que o usuário fornece para finalizar a entrada.

Portanto, para encontrar o número de itens válidos devemos subtrair 1 do valor contido em **i**.

Uma outra modificação neste programa foi o uso da diretiva **#define** para declarar **LIM** como uma constante de valor 40.

```
#define LIM 40
```

Nós usamos **LIM** na declaração da matriz. O uso de constantes, definidas pela diretiva **#define**, para dimensionar matrizes é comum em C pois, se com o passar do tempo quisermos aumentar ou diminuir este valor, precisamos somente trocar o 40 da diretiva **#define** pelo número desejado e a troca refletirá em qualquer lugar do programa onde **LIM** aparece.

CHECANDO LIMITES

Nós dimensionamos a matriz **notas[]** em 40 e este número permite a entrada de até 40 notas.

Suponha, agora, que o usuário decida entrar 60 notas. Como expulsar os dados excedentes?

A linguagem C não realiza verificação de limites em matrizes; por isto, nada impede que você vá além do fim de uma matriz. Se você transpuser o fim da matriz durante uma operação de atribuição, então atribuirá valores a outros dados ou até mesmo a uma parte do código do programa.

Isto acarretará resultados imprevisíveis e nenhuma mensagem de erro do compilador avisará o que está ocorrendo.

C NÃO AVISA VOCÊ QUANDO O LIMITE DE DIMENSIONAMENTO DE UMA MATRIZ FOI EXCEDIDO.

Como programador você tem a responsabilidade de providenciar a verificação dos limites, sempre que necessário. Assim a solução é não permitir que o usuário digite dados, para elementos da matriz, acima do limite. Para isto modificaremos o laço **do-while** como segue:

```
do {
   if(i>=LIM){
      printf("Buffer lotado.\n");
      i++;
      break; /* sai do laco do-while */
   }

printf("Digite a nota do aluno %d: ",i);
scanf("%f",&notas[i]);

if(notas[i]>0)
   soma+=notas[i];

} while(notas[i++]>0);
```

Agora se **i** atingir 40, que é um acima do fim do buffer de 39, a mensagem "buffer lotado." será impressa e o comando **break** fará com que a execução saia do laço **do-while** e passe para a segunda parte do programa.

INICIALIZANDO MATRIZES

Muitas linguagens de programação permitem que variáveis (incluindo matrizes) sejam inicializadas em tempo de compilação e/ou em tempo de execução e a linguagem C não é uma exceção. Inicializações em tempo de compilação aumentam o tempo de compilação mas não penalizam o programa em tempo de execução. Inicializações em tempo de execução têm um efeito oposto isto é, a compilação é mais rápida, a execução é mais lenta e o tamanho do programa é aumentado.

Em C a inicialização de uma matriz é feita em tempo de compilação. As classes de variáveis para este fim são a **classe extern** e a **classe static**.

MATRIZES DAS CLASSES EXTERN OU STATIC PODEM SER INICIALIZADAS.
MATRIZES DA CLASSE AUTO NÃO PODEM SER INICIALIZADAS.

Lembre-se de que a inicialização de uma variável é feita na instrução de sua declaração e é uma maneira de especificarmos valores iniciais prefixados.

O programa seguinte exemplifica a inicialização de uma matriz. Você fornece o preço em centavos e ele calcula, em cruzados novos, quantas notas de 50 centavos, 25 centavos, 10 centavos, 5 centavos e 1 centavo equivalem à quantidade fornecida.

```
/* troco.C */
/* programa que produz troco */

#define LIM 5

int tab[LIM]={50,25,10,5,1};
```



```
main()
{
   int d, valor, quant;

   printf("Digite o valor em centavos: ");
   scanf("%d",&valor);

   for(d=0;d<LIM;d++){
      quant=valor/tab[d];
      printf("Valor da nota = %2d, ",tab[d]);
      printf("Numero de notas = %2d\n ",quant);
      valor %=tab[d];
   }
}
```

Veja uma execução do programa:

```
C> troco
Digite o valor em centavos: 143
Valor da nota = 50, numero de notas = 2
Valor da nota = 25, numero de notas = 1
Valor da nota = 10, numero de notas = 1
Valor da nota =  5, numero de notas = 1
Valor da nota =  1, numero de notas = 3
```

A matriz **tab[]** contém os valores das várias moedas possíveis. A instrução que inicializa a matriz é:

```
int tab[LIM]={50,25,10,5,1}
```

A lista de valores é colocada entre chaves e os valores são separados por vírgulas.

Os valores são atribuídos na seqüência, isto é, tab[0] é 50, tab[1] é 25, tab[2] é 10 e assim por diante, até tab[4] que é 1.

Se em seu programa você desejar inicializar a sua matriz, deve criar uma variável que existirá durante toda a execução do programa. As classes de variáveis com esta característica são **extern** e **static**.

Vamos reescrever o programa *troco.C* para mostrar o uso de variável do tipo **static**:

```
/* troco2.C */
/* programa que produz troco */
#define LIM 5
main()
{
   int d, valor, quant;
   static int tab[]={50,25,10,5,1};
   printf("Digite o valor em centavos: ");
   scanf("%d",&valor);
   for(d=0;d<LIM;d++){
      quant=valor/tab[d]);
      printf("Numero de notas = %2d\n", quant);
      valor%=tab[d];
   }
}
```

O programa é similar ao anterior, mas a definição da matriz foi movida para dentro da função **main()** e a palavra **static** foi adicionada.

Os dois programas operam exatamente do mesmo modo, mas, se houvesse outras funções no programa, não poderiam acessar a matriz **static** em *troco2.C* e sim a matriz **extern** em *troco.C*.

DIMENSIONANDO UMA MATRIZ INICIALIZADA

Que outro tipo de modificação, na declaração da matriz, foi feita entre *troco.c* e *troco2.c*?

O programa *troco.c* usa o valor LIM na declaração da matriz:

```
int tab[LIM] = {50,25,10,5,1};
```

Entretanto, em *troco2.c* este valor é simplesmente extinto, restando apenas um par de colchetes vazios:

```
static int tab[] = {50,25,10,5,1};
```


O que nós podemos dizer sobre isto?

Se nenhum número for fornecido para dimensionar a matriz, o compilador contará o número de itens da lista de inicialização e o fixará como dimensão da matriz.

Na falta de inicialização explícita, variáveis **extern** e variáveis **static** são inicializadas com valor zero; variáveis automáticas têm valor indefinido (isto é, lixo). Se você declarar uma matriz **extern** ou **static** sem inicializá-la, deverá explicitar a sua dimensão.

Se há menos inicializadores que a dimensão especificada, os outros serão zero. É um erro ter-se mais inicializadores que o necessário.

Lamentavelmente, em C, não há como se especificar a repetição de um inicializador, nem de se inicializar um elemento no meio de uma matriz sem inicializar todos os elementos anteriores ao mesmo.

MAIS DE UMA DIMENSÃO

A linguagem C permite matrizes de qualquer tipo, incluindo matrizes de matrizes. Por exemplo, uma matriz de duas dimensões é uma matriz em que seus elementos são matrizes de uma dimensão.

Na verdade, em C, o termo duas dimensões não faz sentido pois todas as matrizes são de uma única dimensão. Usamos o termo mais de uma dimensão para referência a matrizes em que os elementos são matrizes.

Com dois pares de colchetes obtemos uma matriz de duas dimensões e com cada par de colchetes adicionais obtemos uma matriz com uma dimensão a mais.

O exemplo apresentado declara uma matriz de duas dimensões que é usada para imprimir uma grade na tela.

Inicialmente, a matriz é preenchida com pontos. Depois, o programa forma um ciclo através de uma laço **while** onde é impressa a matriz e é solicitado ao usuário que digite um par de coordenadas. Quando o usuário fornece as duas coordenadas (separadas por vírgula), o programa imprime um caractere gráfico na posição correspondente da tela.

Executando este programa você verá como trabalha um sistema de coordenadas de suas dimensões. Digite alguns pares de números como (0,0), (5,0), (0,5) etc... A coordenada horizontal, ou seja, x, é fornecida primeiro.

```
/* grade.c */
/* imprime coordenadas na tela */
#define A 5
#define L 10
main()
{
   char matriz[A][L];
   int x,y;

   for(y=0;y<A;y++)
      for(x=0;x<L;x++)
         matriz[y][x]='.';
   printf("Digite coordenadas na forma x,y (4,2).\n");
   printf("Use numero negativo para sair.\n");
   while(x>=0) {
      for(y=0;y<A;y++) {
         for(x=0;x<L;x++)
            printf("%c",matriz[y][x]);
         printf("\n");
      }

      printf("Coordenadas: ");
      scanf("%d,%d",&x,&y);
      matriz[y][x]='\xB0';
   }
}
```

```
Entre coordenadas na forma x,y (4,2).
Use numero negativo para sair.
.....
.....
.....
.....
.....
Coordenadas: _
```


Lembre-se de que se você fornecer coordenadas que excedem a dimensão da matriz os resultados podem ser desastrosos.

INICIALIZANDO MATRIZES DE DUAS DIMENSÕES

As matrizes de duas dimensões são inicializadas da mesma maneira que as de dimensão única, isto é, os elementos (matrizes) são colocados entre as chaves depois do sinal de igual e separados por vírgulas. Cada elemento é composto por chaves e seus elementos internos separados por vírgulas.

Como exemplo, modificaremos o programa *grade.c* para criar um jogo de batalha naval.

Neste jogo, um jogador (o computador) esconde um número de navios em diferentes localizações de uma grade 10 por 5.

O outro jogador (o usuário) tenta adivinhar onde estão escondidos os navios digitando coordenadas.

Se o usuário acertar, a coordenada é marcada com um bloco sólido e, se errar, será marcada com um bloco pontilhado.

```
/* bnaval.c */
/* jogo de batalha naval */
#define A 5
#define L 10

char inimigo[A][L]=
   {  {0,0,0,0,0,0,0,0,0,0},
      {0,1,1,1,1,0,0,1,0,1},
      {0,0,0,0,0,0,0,1,0,1},
      {1,0,0,0,0,0,0,1,0,0},
      {1,0,1,1,1,0,0,0,0,0}   };

main()
{
   char amigo[A][L];
   int x,y;
   for(y=0;y<A;y++)
```

```
    for(x=0;x<L;x++)
        amigo[y][x]='.';
  printf("Digite coordenadas na forma x,y (4,2).\n");
  printf("Use numero negativo para sair.\n");

  while(x>=0) {
     for(y=0;y<A;y++) {
        for(x=0;x<L;x++)
           printf("%c",amigo[y][x]);
        printf("\n");
     }
     printf("\nCoordenadas: ");
     scanf("%d,%d",&x,&y);
     if(inimigo[y][x]==1) amigo[y][x]='\xDB';
     else amigo[y][x]='\xB0';
     }
  }
```

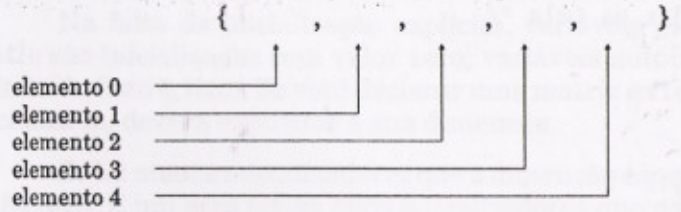
```
Entre coordenadas na forma x,y (4,2).
Use numero negativo para sair.
.........
.........
.........
.........
.........
Coordenadas: _
```

Os navios são estabelecidos na inicialização da matriz, no começo do programa. Como é a inicialização desta matriz?

Preste atenção à seguinte explicação: a matriz **inimigo[]** é uma matriz de 5 elementos. Cada elemento, por sua vez, é uma matriz de 10 elementos. Os elementos da matriz **inimigo[]** são:

```
inimigo[0]
inimigo[1]
inimigo[2]
inimigo[3]
inimigo[4]
```


Estes elementos são os nomes de outras matrizes. Para inicializar esta matriz devemos colocar a lista de elementos em ordem, entre chaves e separados por vírgula como foi feito para matrizes de uma dimensão. Então,



Como cada elemento é uma matriz, em cada posição devemos ter uma matriz de caracteres inicializada. Assim, no lugar de elemento 0 devemos ter

{0,0,0,0,0,0,0,0,0,0}

e no lugar de elemento 1

{0,1,1,1,1,0,0,1,0,1}

e assim por diante.

A instrução

```
char inimigo[A][L]=
{  {0,0,0,0,0,0,0,0,0,0},
   {0,1,1,1,1,0,0,1,0,1},
   {0,0,0,0,0,0,0,1,0,1},
   {1,0,0,0,0,0,0,1,0,0},
   {1,0,1,1,1,0,0,0,0,0}   };
```

é na verdade uma inicialização de uma matriz de uma dimensão em que os elementos são matrizes.

A LISTA DE VALORES, USADA PARA INICIALIZAR UMA MATRIZ, É UMA LISTA DE CONSTANTES SEPARADAS POR VÍRGULAS E ENVOLTA POR CHAVES.

INICIALIZANDO MATRIZES DE TRÊS DIMENSÕES

Como exemplo, mostraremos um fragmento de programa que declara e inicializa uma matriz de 3 dimensões.

```
int tresd[3][2][4]=
{  {  {1,2,3,4},
      {5,6,7,8}   },
   {  {7,9,3,2},
      {4,6,8,3}   },
   {  {7,2,6,3},
      {0,1,9,4}   }  };
```

Esta é uma matriz de matrizes de matrizes.

A matriz de fora tem 3 elementos, cada um destes elementos é uma matriz de duas dimensões de dois elementos onde cada um dos elementos é uma matriz de uma dimensão de quatro números.

Como você representaria o único elemento 0 da declaração?

O primeiro índice é [2], pois é o terceiro grupo de duas dimensões. O segundo índice é [1], pois é o segundo de duas matrizes de uma dimensão. O terceiro índice é [0], pois é o primeiro elemento da matriz de uma dimensão.

Então:

```
tresd[2][1][0] = = 0
```

MATRIZES COMO ARGUMENTOS DE FUNÇÕES

Temos visto exemplos da passagem de vários tipos de variáveis como argumentos de funções.

C permite também passar uma matriz para uma função.

O programa *maxinum.c* usa uma função chamada **max()** para encontrar o elemento da matriz que possui o maior valor.


```
/* maxnum.c */
/* imprime o elemento da matriz de maior valor */
#define VALMAX 20
main()
{
   int list[VALMAX];
   int tamanho=0;
   int num;
   do {
      printf("Digite numero: ");
      scanf("%d",&list[tamanho]);
   } while(list[tamanho++] != 0);

   tamanho--;
   num=max(list,tamanho);
   printf("O maior numero e' %d",num);

}

/* max() */
/* retorna o maior numero */
max(list2,tam2)
int list2[],tam2;
{
   int dex,maxi;
   maxi=list2[0];
   for(dex=1;dex<tam2;dex++)
      if(maxi<list2[dex])
      maxi=list2[dex];
   return(maxi);
}
```

O usuário fornecerá alguns números (não mais que 20) e o programa imprimirá o maior deles.

Aqui está um exemplo da execução do programa:

```
Digite numero: 42
Digite numero:  1
Digite numero: 64
Digite numero: 33
```

```
Digite numero: 27
Digite numero:  0
O maior numero e' 64
```

A primeira parte deste programa é bastante familiar: o laço **do-while** para ler a lista de números. O único elemento novo desta parte do programa é a instrução:

```
num = max(list,tamanho);
```

ela é uma chamada à função **max()** que retorna o maior número. A função **max()** tem dois argumentos: o primeiro é a matriz **list** e o segundo é a variável tamanho.

A parte crítica a ser considerada aqui é como foi passada a matriz para a função: usamos unicamente o nome da matriz.

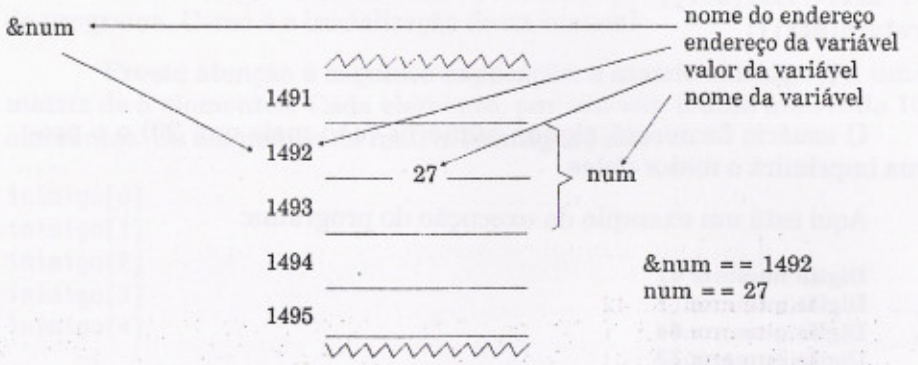
Para referenciar um elemento da matriz, já sabemos que a forma é **list[indice]**; mas o que representa o nome da matriz sem colchetes?

O nome de uma matriz desacompanhado de colchetes é equivalente ao endereço da matriz.

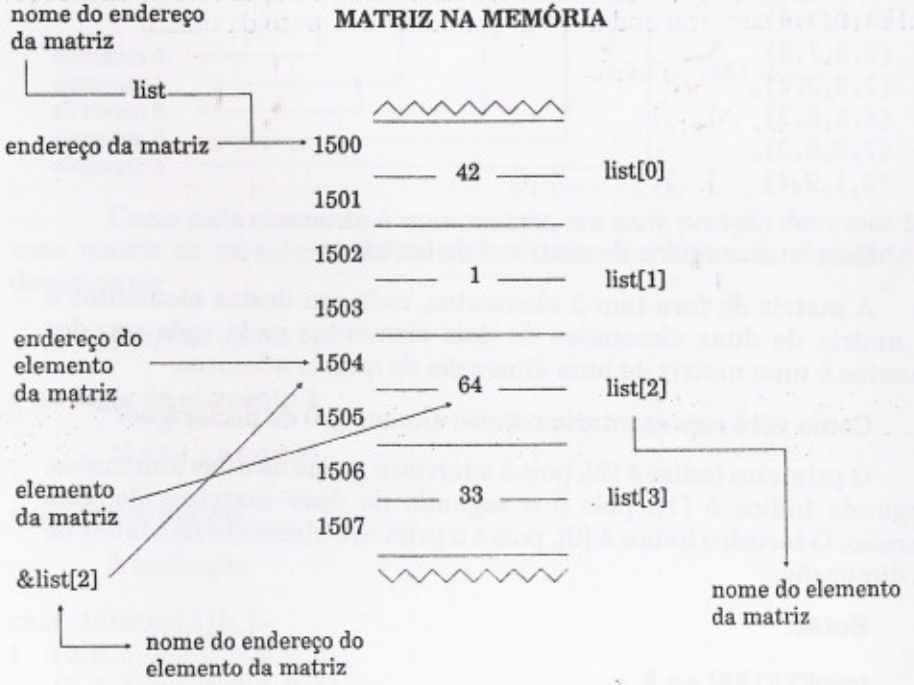
Considerações sobre endereços e valores podem trazer alguma confusão; vamos recapitular o que conhecemos sobre endereços de simples variáveis.

Imagine uma variável inteira **num** com valor de 27. Sua inicialização compreende a seguinte instrução:

```
int num = 27;
```



Temos quatro objetos a considerar sobre variável: seu nome (**num**), seu valor (**27**), seu endereço (**1492**) e o nome do seu endereço (**&num**). Vamos fazer agora uma representação similar com a nossa matriz **list[]**. Suponhamos que a matriz **list[]** esteja localizada na memória começando no endereço 1500.



Note que um elemento da matriz (64) tem nome (list[2]). O endereço da matriz (1500) tem nome (list).

Por que o endereço da matriz não é chamado **&list**?

Em C, esta forma de chamada não será compatível com o modo de endereçamento de variáveis. O nome **list**, entretanto, não é usado para nome de endereço em variável simples. Assim **list** se refere ao endereço se **list** for uma matriz, e a um valor se **list** for uma variável simples.

A propósito, você pode imaginar um outro meio de representar o endereço **list**?

O que você pensa sobre a instrução seguinte?

&list[0]

Visto que **list[0]** é o primeiro elemento da matriz, ele terá o mesmo endereço da própria matriz, e visto que **&** é o operador de endereço, **&list[0]** vai conter o endereço do primeiro elemento da matriz.

Em outras palavras:

list = = &list[0]

O bom entendimento de endereços tornar-se-á imprescindível quando, no Módulo II, estudarmos ponteiros, visto que endereços são particularmente relacionados com ponteiros.

O JOGO DE ADIVINHA NÚMERO

O exemplo seguinte simula um jogo de adivinhações de números. O usuário escolhe o número de dígitos do número a adivinhar e, a cada tentativa, o programa responderá com 2 números onde o primeiro indica número de dígitos corretos nas posições corretas e o segundo, número de dígitos corretos em posições erradas.

Eis a listagem.

```
/* adivnum.c */
/* jogo de adivinhacao de numeros */
#define TAMAX 20
main()
{
   char A[TAMAX], B[TAMAX], C[TAMAX];
   int n, tentativas, k, c;
   printf("\nVoce quer instrucoes? (s/n): ");
   if(getche()=='s') instrucoes();
   do{
      n=numdig();          /* usuario escolhe o num. de digitos */
      preenchA(A,n);
      tentativas=0;
      do{
         if(!(k=preenchB(B,n))) break;
```



```
         tentativas++;
         c=certo1(A,B,C,n);
         printf("\t%d\t%d\n",c,certo2(B,C,n));
      }while(c!=n);
      if(k)
         printf("\n\nCorreto em %d tentativas.",tentativas);
      printf("\nQuer jogar novamente ? (s/n): ");
   } while(getche()=='s');
}

instruções()
{printf("\n");
printf("\n\t\tEste programa toma um numero aleatorio ");
printf("\n\t\tde N digitos (voce escolhe N < 21). Voce ");
printf("\n\t\tdevera digitar um numero de N digitos. O ");
printf("\n\t\tprograma ira responder com 2 numeros, o ");
printf("\n\t\tprimeiro indica digitos corretos nas ");
printf("\n\t\tposicoes corretas, o segundo digitos ");
printf("\n\t\tcorretos em posicoes erradas. Para parar ");
printf("\n\t\tpressione a tecla [Esc].\n\nBOA SORTE ! !");
}

numdig()
{
   int n;

   do {
      printf("\n\nEscolha o numero de digitos (1 a 20):");
      scanf("%d",&n);
   } while(n < 1 || n > 20);
   return(n);
}

preenchA(a,n)
char a[];
int n;
{
   int i;
   for(i=0;i<n;i++) a[i]=rand()%10+'0';
}

preenchB(b,n)
char b[];
```

```
int n;
{
   int i;
   printf("Digite %d digitos: ",n);
   printf("\n?");
   for(i=0;i<n;i++){
      b[i]=getche();
      if(b[i]==27)return(0);
   }
   return(1);
}

certo1(a,b,c,n)
char a[],b[],c[];
int n;
{
   int i,j=0;
   for(i=0;i<n;i++) {
      c[i]=a[i];
      if(c[i]==b[i]) {
         j++;
         b[i]='x';
         c[i]='y';
      }
   }
   return(j);
}

certo2(b,c,n)
char b[],c[];
int n;
{
   int i,j,k=0;
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         if(c[i]==b[j]) {
            k++;
            b[j]='x';
            c[i]='y';
         }
   return(k);
}
```


O programa inicia perguntando se o usuário quer instruções e, se positivo, chama **instruções()**.

Três matrizes são declaradas para armazenar dígitos. A matriz A[] é preenchida com dígitos aleatórios pela função preenchA() onde a instrução

```
a[i] = rand()%10 + '0';
```

cria caracteres entre '0' e '9'.

A matriz B[] é preenchida com os dígitos do usuário, ou seja, a tentativa de acertar o que está em A[], através da função preenchB().

A matriz C[] é usada como auxiliar pelas funções **certo1()**, que devolve o número de dígitos certos nas posições certas, e **certo2()**, que devolve o número de dígitos certos em posições erradas.

CHAMADA POR VALOR E CHAMADA POR REFERÊNCIA

Quando o nome de uma simples variável é usado como argumento na chamada a uma função, a função toma o valor contido nesta variável e o instala em uma nova variável e em nova posição de memória criada pela função. Portanto as alterações que forem feitas nos parâmetros da função não terão nenhum efeito nas variáveis usadas na chamada. Este método é conhecido como **chamada por valor**.

O que ocorre quando o endereço de uma matriz é passado para uma função como argumento?

Como matrizes podem ser bastante grandes, os criadores da linguagem C determinaram que seria mais eficiente termos uma única cópia da matriz na memória para que não seja relevante o número de funções que a acessam. Assim não são passados os valores contidos na matriz, somente o endereço da matriz.

A função usa o endereço para acessar a matriz real usada na chamada.

Isto significa que as alterações que forem feitas na matriz pela função afetarão a própria matriz.

Esta maneira de se passar argumentos é conhecida como **chamada por referência**.

PASSANDO O NOME DE UMA MATRIZ PARA UMA FUNÇÃO, NÃO É CRIADA UMA NOVA CÓPIA DA MATRIZ.

ORDENANDO UMA MATRIZ

Vamos analisar uma função que ordena os valores de uma matriz. A ordenação é uma tarefa importante em várias aplicações, particularmente em programas de banco de dados em que o usuário deseja rearranjar a lista de itens em ordem numérica ou alfabética.

A listagem do programa está a seguir:

```
/* ordnum.c */
/* ordena os valores da matriz */
#define TAMAX 20
main()
{
   int list[TAMAX], tam=0, d;

   do {
      printf("Digite numero: ");
      scanf("%d",&list[tam]);
   } while(list[tam++] != 0);

   ordena(list,--tam);
   for(d=0; d<tam ; d++)
      printf("%d\n",list[d]);
}

/* ordena() */
/* ordena matriz de inteiros */
ordena(list,tam)
int list[],tam;
```



```
{
   int j, i, temp;
   for(j=0; j<tam-1; j++)
      for(i=j+1;i<tam;i++)
         if(list[j] > list[i]) {
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
      }
}
```

Em seguida está um exemplo da execução do programa:

```
Digite numero: 46
Digite numero: 25
Digite numero: 73
Digite numero: 58
Digite numero: 33
Digite numero: 18
Digite numero:  0

18
25
33
46
58
73
```

O programa primeiro solicita a lista de números (você não deve digitar mais de 20). Enquanto o usuário fornecer números, eles serão colocados na matriz **list[]**. Quando o usuário terminar a entrada pressionando 0 (que não é colocado na lista), o programa chama a função **ordena()**, que ordena os valores da lista.

A estrutura deste programa é similar à estrutura de *maxnum.c* e usamos o mesmo nome de matriz, **list[]**.

A ORDENAÇÃO BOLHA

O processo de ordenação utilizado na função **ordena()** merece alguma explanação.

A função começa considerando a primeira variável da matriz, **list[0]**. O objetivo é colocar o menor item da lista nesta variável. Assim a função percorre todos os itens restantes da lista, de **list[1]** a **list[tam-1]**, comparando cada um com o primeiro item. Sempre que encontrar um item menor, eles são trocados. Terminada esta operação, é tomado o próximo item, **list[1]**. Este item deverá conter o próximo menor item. E novamente são feitas as comparações e trocas. O processo continua até que a lista toda esteja ordenada.

Este método é chamado de **Ordenação Bolha**. Sua popularidade vem de seu nome fácil e de sua simplicidade. Porém é um dos algoritmos de ordenação menos eficiente.

A ordenação bolha é feita através de dois laços. O laço mais externo, da variável **j**, determina qual elemento da matriz será usado como base de comparação (**list[0]** é o primeiro). O laço mais interno, da variável **i**, compara cada item com o de base e, quando encontrar um item menor que o de base, faz a troca.

O processo de troca usa uma variável temporária para guardar o valor de **list[i]**, depois coloca em **list[i]** o valor de **list[j]** e finalmente em **list[j]** o valor armazenado na variável temporária.

Lembre-se de que todas as trocas são feitas na matriz original do programa chamador.

MATRIZES DE DUAS DIMENSÕES COMO ARGUMENTO

Antes de iniciarmos o estudo de "strings", vamos exemplificar o uso de uma matriz de 2 dimensões como argumento de função através de um programa para avaliar a eficiência de funcionários de uma loja, quanto ao número de peças vendidas por cada um.

O primeiro índice da matriz indica o número de funcionários da loja e o segundo índice, o número de meses a serem avaliados.

```
/* histogr.c*/
/* histograma horizontal */
#define FUNC 10          /* numero de funcionarios */
#define MES 3            /* numero de meses */
#define MAXBARRA 50      /* tamanho maximo da barra */
```



```
main(){
   int pecas[FUNC][MES];
   int i,j;
   for(i=0;i<FUNC;i++)
      for(j=0;j<MES;j++) {
         printf("Numero de pecas vendidas pelo funcionario");
         printf("%d no mes %d: ",i+1,j+1);
         scanf("%d",&pecas[i][j]);
      }

   grafico(pecas);
   getche();
}

/* grafico() */
/* desenha histograma */
grafico(p)
int p[][MES];
{
   int i,j,max,tam,media[FUNC];

   for(i=0;i<FUNC;i++) {
      media[i]=0;
      for(j=0;j<MES;j++)
         media[i]+=p[i][j];
      media[i]/=MES;
   }
   max=0;
   for(i=0;i<FUNC;i++)
      if(media[i] > max)
         max=media[i];
   printf("\n\n\n\n FUNC - MEDIA\n---------------\n");
   for(i=0;i<FUNC;i++) {
      printf("%5d - %5d : ",i+1,media[i]);
      if(media[i] > 0) {
         if((tam=(media[i]*MAXBARRA/max)) <= 0)
            tam=1;
      } else tam=0;

      while(tam>0) {
```

```
            printf("%c",'\xDB');
            --tam;
        }
        printf("\n");
    }
}
```

A passagem de uma matriz de duas dimensões para uma função é similar à passagem de uma matriz de uma dimensão.

O método de passagem do endereço da matriz para a função é idêntico, não importando quantas dimensões tem a matriz, visto que sempre passamos o endereço da matriz.

```
grafico(pecas)
```

Entretanto, a declaração da matriz na função é um tanto misteriosa:

```
int p[][mes];
```

Não precisamos informar à função quantas linhas tem a matriz. Por que não?

Como não há verificação de limites, uma matriz com qualquer número de linhas pode ser passada para a função chamada.

Uma função, que recebe uma matriz bidimensional como parâmetro, deverá conhecer o comprimento da segunda dimensão para poder operar com declarações do tipo

```
p[2][1]
```

pois, para encontrar a posição de memória onde **p[2][1]** está guardado, a função multiplica o índice da linha (2) pelo número de elementos por linha (mês que é 3) e adiciona o índice da coluna (1).

O resultado é: 2 * 3 + 1 = 7

Caso o comprimento das linhas não seja conhecido, será impossível saber onde inicia, por exemplo, a terceira linha.

Neste ponto você já está apto a trabalhar com matrizes. Você sabe como declarar matrizes de diferentes dimensões e tamanhos, como inicializar matrizes, como referenciar um elemento particular da matriz e como passar uma matriz para uma função. A próxima parte deste capítulo mostrará como trabalhar com "strings" que são, simplesmente, um tipo especial de matriz.

STRINGS

String é uma das mais úteis e importantes formas de dados em C e é usada para armazenar e manipular textos como palavras, nomes e sentenças.

Em C, string não é um tipo de dado formal como em outras linguagens como Pascal e Basic.

String é uma matriz do tipo **char** terminada pelo caractere null ('\0').

Como uma matriz é um conjunto de dados de mesmo tipo e string é uma série de caracteres, a definição de string como sendo uma matriz do tipo **char** é bastante razoável.

Cada caractere de uma string pode ser acessado como um elemento de uma matriz do tipo **char**, o que proporciona uma grande flexibilidade aos programas que processam textos.

STRINGS CONSTANTES

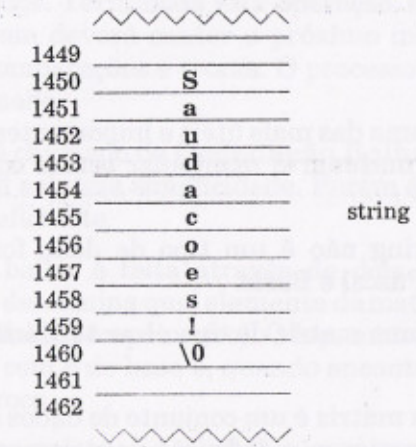
Sempre que o compilador encontrar qualquer coisa entre aspas duplas, ele reconhece que se trata de uma string constante, isto é, os caracteres entre as aspas mais o caractere null ('\0').

Ao longo deste livro já mostramos exemplos de **strings** constantes, como na instrução seguinte:

```
printf("%s","Saudacoes!");
```

"Saudacoes !" é uma string constante.

Cada caractere de uma string ocupa 1 byte de memória e o último caractere é sempre '\0' (null). O caractere **null** ou '\0' tem o valor 0 (zero) decimal. Note que isto não é o mesmo que o caractere 0 que tem valor 48 decimal.



String constante armazenada da memória

TODA "STRING" DEVE TERMINAR PELO CARACTERE NULL, '\0', QUE TEM VALOR 0 DECIMAL.

A terminação '\0' é importante, pois é a única maneira que as funções possuem para poderem reconhecer onde é o fim da string.

VARIÁVEIS STRINGS

Uma das maneiras de receber uma string do teclado é através da função **scanf()** pelo formato %s. Eis um exemplo:

```
/* string1.c */
/* le "string" do teclado e imprime-a */
main()
{
   char nome[15];

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



```
   scanf("%s",nome);
   printf("Saudacoes, %s.",nome);
}
```

A instrução

```
scanf("%s",nome);
```

lê cada caractere não branco e os armazena a partir do endereço **nome**. O processo termina quando um caractere branco é encontrado. Neste ponto é incluído automaticamente o caractere '\0' na próxima posição livre da matriz. Assim, se a sua matriz tiver 15 caracteres de comprimento, você só poderá digitar até 14 caracteres, deixando livre pelo menos uma posição para o caractere null.

Você deve ter notado que não usamos o operador **&** precedendo o segundo argumento de **scanf()**. Visto que o nome de uma matriz é o seu endereço inicial, seria errado utilizar o operador de endereço (**&**) junto a ele. A expressão nome é equivalente a **&nome[0]**.

LENDO STRINGS

Ler uma string consiste em dois passos: reservar espaço de memória para armazená-la e usar alguma função que permita a sua entrada.

O primeiro passo é declarar a string especificando o seu tamanho.

```
char nome[15];
```

Uma vez reservado o espaço necessário você pode usar a função **scanf()** ou a função **gets()** de biblioteca C para receber a string.

A FUNÇÃO scanf()

A função **scanf()** é bastante limitada para a leitura de strings. Por exemplo, considere a seguinte execução de *string1.c*.

```
Digite seu nome: Hamilton Gomes
Saudacoes, Hamilton
```

O programa subitamente eliminou o último nome. Lembre-se de que **scanf()** usa qualquer espaço em branco para terminar a entrada. O resultado é que não existe uma forma de digitar um texto de múltiplas palavras numa única variável usando **scanf()**.

A função **scanf()** é principalmente usada para ler uma mistura de tipos de dados numa mesma instrução. Por exemplo, se em cada linha de entrada quisermos ler o nome de um material do estoque, o seu número do estoque e o seu preço **scanf()** se adaptam perfeitamente.

A FUNÇÃO gets()

A função **gets()** é bastante conveniente para a leitura de strings. O seu propósito é unicamente ler uma string da sua entrada padrão que por "default" é o teclado.

Visto que uma string não tem um tamanho predeterminado, **gets()** lê caracteres até encontrar o de nova linha ('\n') que é gerado pressionando-se a tecla **[Enter]**. Todos os caracteres anteriores ao '\n' são armazenados na string e é então incluído o caractere '\0'.

Caracteres brancos como espaços e tabulações são perfeitamente aceitáveis como parte da string. Eis um exemplo simples:

```
/* getnome1.c */
/* mostra o uso de gets() */
main()
{
   char nome[81];

   printf("Digite seu nome: ");
   gets(nome);
   printf("Saudacoes, %s.",nome);
}
```


A saída:

```
Digite seu nome: Hamilton Gomes
Saudacoes, Hamilton Gomes
```

A função **gets()** é exatamente o que procurávamos: agora o programa lembrará todo o nome.

IMPRIMINDO STRINGS

As duas principais funções para imprimir strings são **printf()** e **puts()**.

A FUNÇÃO puts()

A função **puts()** é o complemento de **gets()** e é bem fácil de usar.

O propósito de **puts()** é o de imprimir uma única string por vez. O endereço desta string deve ser mandado para **puts()** como argumento.

O próximo exemplo ilustra algumas das muitas possibilidades de seu uso.

```
/* puts1.c */
/* mostra o uso de puts() */
main()
{
   char nome[81];

   puts("Digite seu nome: ");
   gets(nome);
   puts("Saudacoes, ");
   puts(nome);
   puts("puts() pula de linha sozinha");
   puts(&nome[4]);
}
```

A saída:

```
Digite seu nome:
Hamilton Gomes
Saudacoes,
Hamilton Gomes
puts() pula de linha sozinha
lton Gomes
```

A função **puts()** reconhece o '\0' como fim da string. Observe que cada string impressa por **puts()** termina por um caractere de nova linha. Portanto, se você desejar imprimir duas strings na mesma linha, a solução é usar **printf()**.

As duas instruções seguintes têm o mesmo efeito:

```
printf("%s\n",nome);
puts(nome);
```

INICIALIZANDO STRINGS

Lembre-se de que qualquer matriz pode ser inicializada contanto que seja da classe **extern** ou **static**. Esta característica vale também para strings que são matrizes do tipo **char**.

```
char nome[]={'A','n','a','\0'};
```

O compilador oferece uma forma de inicialização de strings equivalente, mas consideravelmente mais simples:

```
char nome[]="Ana";
```

Enquanto os caracteres simples são colocados entre aspas simples e o conjunto é envolto por chaves, a string é colocada entre aspas duplas. O uso da segunda forma causa a inclusão automática do caractere '\0'.

```
/* inistr.c */
/* le e imprime string, mostra inicializ. de "string" */
main()
{
   static char salute[]="Saudacoes, ";
   char nome[81];
```



```
   puts("Digite seu nome: ");
   gets(nome);
   puts(salute);
   puts(nome);
}
```

Como somente matrizes de classe extern ou static podem ser inicializadas, usamos static.

```
Digite seu nome:
Hamilton Gomes
Saudacoes,
Hamilton Gomes
```

OUTRAS FUNÇÕES DE MANIPULAÇÃO DE STRINGS

Vários compiladores oferecem, em suas bibliotecas, funções para manipular strings. Aqui veremos 4 delas: **strlen()**, **strcat()**, **strcmp()** e **strcpy()**. No Módulo II deste livro mostraremos como escrever estas funções, caso o seu compilador não as ofereça.

A FUNÇÃO strlen()

A função **strlen()** aceita um endereço de string como argumento e retorna o tamanho da string armazenada a partir deste endereço até um caractere antes do '\0'.

O programa seguinte examina cada posição de memória ocupada pela "string" e imprime o que encontrou nela.

```
/* examem.c */
/* ver "string" na memoria */

main()
{
   char nome[81];
   int d;
```

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

   for(d=0;d<strlen(nome)+4;d++)
      printf("End=%5u caractere='%c'=%3d\n",
         &nome[d],nome[d],nome[d]);
}
```

Analise um exemplo da execução do programa:

```
Entre seu nome:
Helio
End=3486   caractere=   'H' =  72
End=3487   caractere=   'e' = 101
End=3488   caractere=   'l' = 108
End=3489   caractere=   'i' = 105
End=3490   caractere=   'o' = 111
End=3491   caractere=   ''  =   0
End=3492   caractere=   '<' =  60
End=3493   caractere=   '#' =  35
End=3494   caractere=   'u' =  17
```

Para mostrar o que há depois do fim da **string**, imprimimos 4 caracteres além. O primeiro é o caractere **null**, que não é impresso pois tem valor 0. Os outros são caracteres considerados lixo que podem ter qualquer valor dependendo do que o computador armazenou naquelas posições. Se tivéssemos declarado a matriz como **static** ou **extern** estes espaços teriam 0 armazenado em vez de lixo.

Em nosso programa a instrução

```
strlen(nome)
```

retorna o valor 5 se nome for "Helio". Observe que **strlen()** não conta o caractere **null**.

A FUNÇÃO strcat()

A função **strcat()** concatena duas strings, isto é, junta uma string ao final de outra. Ela toma dois endereços de strings como argumento e copia a segunda string no final da primeira e esta combinação gera uma nova primeira string. A segunda string não é alterada.

Cuidado! Esta função não verifica se a segunda string cabe no espaço livre da primeira string. Certamente você poderia usar **strlen()** antes de usá-la.

```
/* juntastr.c */
/* junta duas strings */
main()
{
   static char salute[81] = "Saudacoes, ";
   char nome[20];
   puts("Digite seu nome:");
   gets(nome);
   strcat(salute,nome);
   puts(salute);
   puts(nome);
}
```

A saída

```
Digite seu nome:
Hamilton
Saudacoes, Hamilton
Hamilton
```

A FUNÇÃO strcmp()

Suponhamos que você queira comparar a resposta do usuário com uma string interna:

```
/* errado.c */
main()
{
   static char resp[]="branco";
   char r1[40];
   puts("Qual e' a cor do cavalo branco de Napoleao?");
   gets(r1);
```

```
   while(r1 != resp) {
      puts("Resposta errada. Tente de novo.");
      gets(r1);
   }
   puts("Correto! ");
}
```

Este programa não trabalhará corretamente, pois **r1** e **resp** são endereços, então a expressão

r1!= resp

realmente não pergunta se as duas strings são iguais e sim se os dois endereços são iguais. Como **r1** e **resp** estão em endereços distintos, nunca serão os mesmos.

Como fazer para comparar o conteúdo de strings e não seus endereços?

A solução é usar a função **strcmp()**.

```
/* certo.c */
main()
{
   static char resp[]="branco";
   char r1[40];
   puts("Qual e' a cor do cavalo branco de Napoleao ?");
   gets(r1);

   while(strcmp(r1,resp) != 0) {
      puts("Resposta errada. Tente de novo.");
      gets(r1);
   }
   puts("Correto! ");
}
```

A forma geral de **strcmp()** é

strcmp(string1,string2)

e o valor retornado é

<0	se string1 < string2
0	se string1 = string2
>0	se string1 > string2

Neste contexto, "menor que" (<) ou "maior que" (>) indica que, se você colocar string1 e string2 em ordem alfabética, o que aparecerá primeiro será menor que o outro.

O exemplo seguinte testa e imprime o valor retornado por **strcmp()** em várias situações.

```
/* compstr.c */
/* imprime o valor que strcmp() retorna */
main()
{
   printf("%d\n",strcmp("A","A"));
   printf("%d\n",strcmp("A","B"));
   printf("%d\n",strcmp("B","A"));
   printf("%d\n",strcmp("C","A"));
   printf("%d\n",strcmp("casas","casa"));
}
```

A saída:

```
 0
-1
 1
 2
115
```

A função retorna a diferença em ASCII entre os primeiros dois caracteres diferentes.

UMA MATRIZ DE STRINGS

Ao longo deste capítulo vimos vários exemplos de matrizes bidimensionais. Como uma string é uma matriz, uma matriz de string é na realidade uma matriz de matrizes, ou uma matriz de duas dimensões.

O nosso programa exemplo solicita a você que digite o seu nome. Quando você faz isto, ele checa seu nome contra uma lista mestre, para ver se você é digno ou não de entrar no palácio.

```
/* compara.c */
/* compara palavra fornecida com palavra no programa */
#define MAX 5
#define LEN 10

main()
{
int d;
int entra=0;
char nome[40];
static char list[MAX][LEN]=
{  "Katarina",
   "Nigel",
   "Gustavo",
   "Francisco",
   "Airton" };
printf("Digite seu nome: ");
gets(nome);
for(d=0;d<MAX;d++)
   if(strcmp(&list[d][0],nome)==0)
   entra=1;
if(entra==1)
   printf("Voce pode entrar, oh honrado Sr.!");
else
   printf("Guardas! Removam este sujeito!");
}
```

Em seguida estão as duas possíveis saídas do programa:

Digite seu nome: Gustavo
Voce pode entrar, oh honrado Sr.!

ou

Digite seu nome: Roberto
Guardas! Removam este sujeito!

Repare como a matriz de strings é inicializada. Como a frase entre aspas é uma matriz de uma dimensão e não um conjunto de elementos, não é necessário usar chaves envolvendo cada nome.

Na instrução

strcmp(&list[d][0],nome)==0

utilizamos novamente a função **strcmp()**.

A FUNÇÃO strcpy() E APAGANDO CARACTERES

Freqüentemente é vantajoso ser capaz de apagar caracteres do interior de uma string (se você for desenvolver um editor de textos, por exemplo).

A biblioteca de C não fornece funções para isto, então vamos desenvolver uma rotina para esta tarefa.

Deixaremos como exercício a função complementar que insere caracteres no interior de uma string.

```
/* delete.c */
/* apaga caractere de "string" */

main()
{
   char string[81];
   int posicao;

   printf("Digite string, <return>, posicao\n");
   gets(string);
   scanf("%d",&posicao);
   strdel(string,posicao);
   puts(string);
}

/* strdel() */
/* apaga caractere de "string" */
```

```
strdel(str,n)
char str[];
int n;
{
   strcpy(&str[n],&str[n+1]);
}
```

Este programa solicita uma string e a posição do caractere a ser apagado (lembre que o primeiro caractere é 0). Em seguida chama a função **strdel()** para apagar o caractere.

Aqui está uma execução do programa:

```
Digite string, <RETURN>, posicao
carrta
2
carta
```

A função **strdel()** move, um espaço à esquerda, todos os caracteres que estão à direita do caractere sendo apagado.

Para mover os caracteres, a função usa uma função de biblioteca C **strcpy()**. Esta função simplesmente copia uma string em outra.

O PROGRAMA QUE IMPRIME CARTÃO DE NATAL

O próximo programa usa as funções de manipulação de strings para imprimir um lindo cartão de natal. Você pode usar redirecionamento para obter uma cópia impressa.

```
C>CARTAO > PRN
```

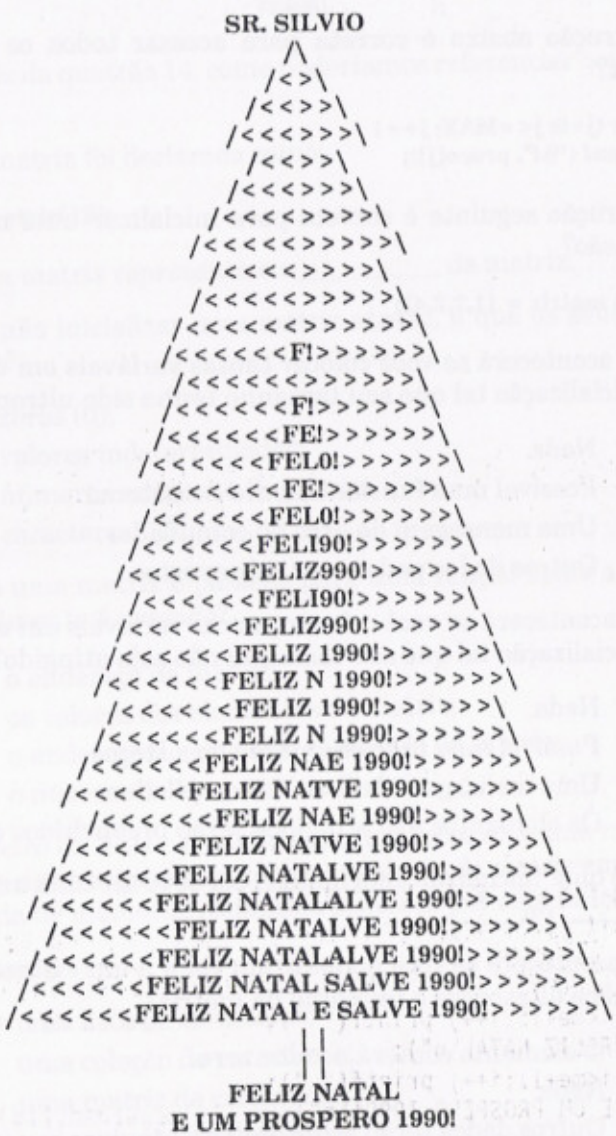
Eis a listagem:

```
/* cartao.c */
/* imprime cartao de natal */
main()
{
   char nome[81],mensagem[39],se,sd,b[42];
   int t,ll=0,me=20,d,ns,i,fim,l,j;
```



```
   printf("\nDestinatario: "); gets(nome);
   printf("\nMensagem dentro da arvore: "); gets(mensagem);
   t=strlen(mensagem);
   printf("\nSinal interno direito:"); sd=getche();
   printf("\nSinal interno esquerdo:");se=getche();
   d=38-t; b[0]='/'; ns=d/2;
   for(i=1;i<=ns;i++) b[i]=se;
   strcpy(&b[i],mensagem);
   fim=strlen(b);
   for(i=fim;i<fim+ns;i++) b[i]=sd;
   b[i]='\\'; b[i+1]='\0'; fim=strlen(b);

   /* impressao */

   printf("\n");
   for(i=0;i<(80-strlen(nome))/2;i++) printf(" ");
   printf("%s\n\n\n",nome);

   for(i=1;i<=36;i++) {
      l=i-11;
      for(j=1;j<=me+20-l;j++) printf(" ");
      for(j=0;j<l;j++) printf("%c",b[j]);
      printf("%s\n",&b[fim-l]);
      if(!(i%4)) ll=ll+2;
   }

   for(i=0;i<me-1;i++) printf(" ");
   printf("%c",'/');
   for(i=0;i<20;i++) printf("%c",se);
   for(i=0;i<20;i++) printf("%c",sd);
   printf("%c\n",'\\');

   for(i=0;i<4;i++) {
      for(j=0;j<me+18;j++) printf(" ");
      printf("| |\n");
   }
   printf("\n\n");
   for(i=0;i<me+15;i++) printf(" ");
   printf("FELIZ NATAL\n");
   for(i=0;i<me+11;i++) printf(" ");
   printf("E UM PROSPERO 1990!\n");
}
```

Destinatario: SR. SILVIO
Mensagem dentro da arvore: FELIZ NATAL E SALVE 1990!
Sinal interno direito: >
Sinal interno esquerdo: <



REVISÃO

1. Uma matriz é uma coleção de itens de tipos idênticos tratados como uma unidade.
2. Os elementos de uma matriz são diferenciados através de um índice sempre iniciado em 0 (não 1, como em outras linguagens).
3. Matrizes são armazenadas numa seqüência contínua de memória e podem ter qualquer tipo.
4. C não avisa você quando o limite de dimensionamento de uma matriz foi excedido.
5. Matrizes podem ser das classes **auto**, **extern** e **static** mas não **register**. Matrizes **extern** e **static** podem ser inicializadas. Matrizes **auto** não podem.
6. Todas as matrizes em C são, na verdade, de uma única dimensão. O termo mais de uma dimensão indica matrizes em que os elementos são matrizes.
7. O nome de uma matriz é o seu endereço.
8. Uma string é uma matriz do tipo **char**.
9. A função **gets()** lê uma string e a função complementar **puts()** imprime uma string.
10. A biblioteca C oferece várias funções para a manipulação de strings, dentre elas as mais usadas são: **strlen()**, **strcat()**, **strcmp()** e **strcpy()**.

EXERCÍCIOS

1. Uma matriz é uma coleção de variáveis de:
   a) diferentes tipos de dados distribuídos pela memória;
   b) tipos de dados similares distribuídos pela memória;
   c) tipos de dados similares em seqüência na memória;
   d) diferentes tipos de dados em seqüência na memória.

2. Em que uma string é semelhante a uma matriz?

a) São duas matrizes de caracteres.
b) A matriz é um tipo de string.
c) Acessam funções do mesmo modo.
d) String é um tipo de matriz.

3. Em uma declaração de matriz devem ser especificados o t_____, o n_____ e o t_____ da matriz.

4. A declaração de matriz é correta?

```
int num(25);
```

5. Qual é o elemento da matriz referenciado por esta expressão?

```
num[4]
```

6. Qual é a diferença entre os números "3" destas duas instruções?

```
int num[3];
num[3] = 5;
```

a) O primeiro especifica um elemento particular e o segundo, um tipo.
b) O primeiro especifica tamanho e o segundo, um elemento particular.
c) O primeiro especifica um elemento particular e o segundo, o tamanho da matriz.
d) Os dois especificam elementos da matriz.

7. O que faz a combinação das instruções seguintes?

```
#define LIM 50
char coleta [LIM];
```

a) Torna LIM um índice.
b) Torna LIM uma variável tipo float.
c) Torna coleta[] uma matriz do tipo LIM.
d) Torna coleta[] uma matriz de tamanho LIM.

8. Se uma matriz é declarada como:

```
float preco[MAX];
```

a instrução abaixo é correta para acessar todos os elementos da matriz?

```
for (j=0; j<=MAX; j++)
scanf ("%f", preco[j]);
```

9. A instrução seguinte é correta para inicializar uma matriz de uma dimensão?

```
int matriz = {1,2,3,4};
```

10. O que acontecerá se você colocar tantas variáveis em uma matriz na sua inicialização tal que seu tamanho tenha sido ultrapassado?

a) Nada.
b) Possível mau funcionamento do sistema.
c) Uma mensagem de erro do compilador.
d) Outros dados podem ser sobrepostos.

11. O que acontecerá se você colocar poucas variáveis em uma matriz na sua inicialização tal que seu tamanho não seja atingido?

a) Nada.
b) Possível mau funcionamento do sistema.
c) Uma mensagem de erro do compilador.
d) Os elementos não atingidos serão preenchidos com zeros.

12. Se você quer inicializar uma matriz, ela deve ser uma matriz da classe _____ ou_____.

13. O que acontecerá se você atribuir um valor a um elemento da matriz cujo índice ultrapassa o tamanho da matriz?

a) O elemento conterá o valor zero.
b) Nada.
c) Outros dados serão sobrepostos.
d) Mau funcionamento do sistema.

14. A inicialização abaixo é correta?

```
int matriz[3][3] = {    {1,2,3},
                        {4,5,6},
                        {7,8,9}     };
```

15. Na matriz da questão 14, como poderíamos referenciar o elemento de valor 4?

16. Se uma matriz foi declarada como:

```
int matriz[12];
```

A palavra matriz representa o e_____ da matriz.

17. Se você não inicializar uma matriz **static**, o que os seus elementos conterão?

a) zeros (0);
b) valores indeterminados;
c) números em ponto flutuante;
d) caracteres '\0'.

18. Quando uma matriz é passada para uma função como argumento, o que realmente é passado?

a) o endereço da matriz;
b) os valores dos elementos da matriz;
c) o endereço do primeiro elemento da matriz;
d) o número de elemento da matriz.

19. *Verdadeiro ou Falso:* Quando uma função recebe uma matriz inteira passada como argumento, coloca os valores da matriz em uma posição separada de memória, conhecida somente por esta função.

20. Uma string é:

a) uma lista de caracteres;
b) uma coleção de caracteres;
c) uma matriz de caracteres;
d) um conjunto de caracteres.

21. "A" é um _____ enquanto 'A' é um_____ .

22. O que é a expressão seguinte?

"Mesopotamia\n"

a) uma variável string;
b) uma string matriz;
c) uma string constante;
d) uma string de caracteres.

23. Uma string é terminada pelo caractere _____ , que é chamado _____ .

24. A função _____ é projetada especificamente para ler uma string do teclado.

25. Se você tem declarado uma string como:

```
char nome [10];
```

e em seu programa você solicitar ao usuário fornecer a string, o máximo de caracteres que ele deverá fornecer é _____ .

26. *Verdadeiro ou Falso:* A função **puts()** sempre adiciona um '\n' no final da string sendo impressa.

27. Qual das funções é mais apropriada para ler uma string composta por várias palavras?

a) **gets()**
b) **printf()**
c) **scanf()**
d) **puts()**

28. Assuma a seguinte inicialização:

```
char string[] = "Brasileira";
```

Como você se refere à string "leira" (as últimas 5 letras da string)?

29. Qual é o erro sutil que esta instrução exibe?

```
nome = "GEORGE";
```

30. Qual a expressão que você usará para encontrar o comprimento da string nome?

31. Escreva uma função que insira um caractere em qualquer posição de uma string e escreva um programa que teste esta função. A chamada à função deve ser de forma

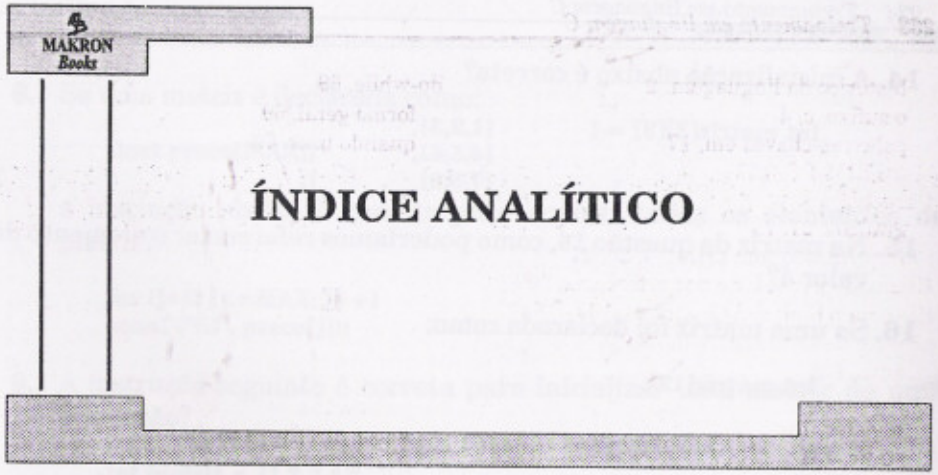
```
strins(string, caractere, posição);
```

32. Escreva a função **strlen()**.

33. Escreva a função **strcat()**.

34. Escreva a função **strcmp()**.

35. Escreva a função **strcpy()**.



ÍNDICE ANALÍTICO

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X