APOSTILA DO CURSO

LINGUAGEM C

Prof. Erico Fagundes Anicet Lisboa, M. Sc.
erico@ericolisboa.eng.br

Versão digital disponível na internet
http://www.ericolisboa.eng.br

RIO DE JANEIRO, RJ - BRASIL
NOVEMBRO DE 2001

ÍNDICE

CAPÍTULO 1

INTRODUÇÃO

1.1 História

A origem do nome da linguagem C é muito simples. É a linguagem que sucede a linguagem B. Por sua vez, a linguagem B teve seu nome retirado da inicial do local onde ela foi desenvolvida: Laboratórios Bell. A primeira versão da linguagem C foi escrita e implementada por D.M. Ritchie. Foi inicialmente publicada no livro "The C Programming Language", por B.W. Kernighan & D.M. Ritchie em 1978.

Diversas versões de C, incompatíveis, foram criadas. Estas versões funcionavam somente com um determinado compilador, rodando apenas em uma única plataforma, o que tornava os códigos computacionais muito restritos a determinadas condições. Em 1983, a ANSI (American National Standards Institute) fundou uma comissão para definir uma versão padronizada para a linguagem C. Esta versão chamou-se ANSI C. Desta forma, simplesmente compilando o código fonte em qualquer sistema, um programa escrito em ANSI C funciona em praticamente qualquer computador.

1.2 Estruturação de um programa em C

A seguir é fornecido um programa exemplo, contendo algumas das principais funcionalidades da linguagem C. Cada item do programa será detalhado mais adiante no curso.

```
main()
{
    /* Programa exemplo */
    float a, b, c;

    printf("Digite dois numeros:\n");
    scanf("%f", &a);
    scanf("%f", &b);
    if(a > b)
        c = a * a;
    else
        c = b * b;
    printf("Quadrado do maior numero digitado: %f\n", c);
}
```

A primeira linha do programa, main(), indica que é a primeira função a ser executada, ou seja, é por onde o programa começa a execução.

O abre-chaves, {, na segunda linha começa o corpo da função.

A terceira linha, /* Programa exemplo */, é um comentário e é ignorada pelo compilador.

Na quarta linha são declaradas as três variáveis que serão utilizadas pelo programa.

A quinta linha é uma linha vazia. É ignorada pelo compilador e pode ser utilizada em qualquer lugar dentro de um código em C. Normalmente, utiliza-se para separar partes lógicas de um código.

As linhas 7 e 8 recebem dois valores do teclado, através da função scanf().

As próximas 4 linhas calculam o valor da variável *c*, dependendo da comparação entre *a* e *b*, feita pelo bloco **if-else**.

A linha 13 imprime o resultado na tela, através da função printf().

O fecha-chaves, }, na última linha encerra a função.

1.3 Tipos

A linguagem C possui 5 tipos básicos. São os tipos **char**, **int**, **float**, **double** e **void**. A tabela abaixo apresenta algumas propriedades de cada tipo.

Tipo	Descrição	Tamanho	Intervalo
char	caractere	1 bytes	-128 a 127 ou 0 a 255
int	inteiro	2 ou 4 bytes	-32768 a 32767 ou -214783648 a 214783647
float	ponto flutuante	4 bytes	-1.7E38 a 1.7E38 (precisão de 6 digitos)
double	ponto flutuante de dupla precisão	8 bytes	-1.7E38 a 1.7E38 (precisão de 16 digitos)
void	vazio	0 bytes	-

O tamanho do tipo inteiro varia com o compilador utilizado. Este tipo possui ainda três variações, a seguir:

Tipo	Descrição	Tamanho	Intervalo
long int	inteiro longo	4 bytes	-214783648 a 214783647
short int	inteiro curto	2 bytes	-32768 a 32767
unsigned int	inteiro sem sinal	2 ou 4 bytes	0 a 65535 ou 0 a 4294967295

1.4 Variáveis

Em C, todas as variáveis precisam ser declaradas. A declaração tem a forma

tipo nome-da-variável

ou

tipo nome-da-variável1, da-variável2, ...

onde *tipo* é o tipo da variável e *nomes-das-variáveis* são separadas por vírgulas.

Os nomes das variáveis devem começar com uma letra ou o sublinhado ('A' a 'Z', 'a' a 'z' e '_'). O restante do nome pode ser composto por letras, sublinhado ou números. Também não são permitidos como nomes de variáveis palavras reservadas pela linguagem C. A tabela abaixo fornece alguns nomes de variáveis válidos e não-válidos.

Nomes válidos	Nomes não válidos
ABC	/ABC
tomate	int
_g	g*
agua_do_mar	agua-do-mar
carro5000	5000carro

1.5 Constantes

Existem diversos tipos de constantes: inteira, ponto flutuante, caractere e cadeia de caracteres.

Constante numérica	Significado
10	constante inteira
017	constante octal
0xFF, 0XF0	constante hexadecimal
64L	constante longa
78678537	constante longa (implícito)
74.1, 1., .5	constante de ponto flutuante

Constantes de caractere são representadas entre apóstrofos (') e equivalem ao número pelo qual o caractere é representado na máquina. A maioria das máquinas utiliza a representação ASCII (American Standard Code for Information Interchange). Para permitir portabilidade, constantes de caractere devem ser utilizadas no lugar de seus equivalentes inteiros.

Constante de caractere	Valor ASCII
'A'	65
'Z'	90
'='	61

Caracteres especiais são representados com a barra invertida (\) seguida de um determinado caractere.

Constante de caractere	Caractere representado
'\n'	caractere de mudança de linha (LF)
'\r'	caractere de retorno de carro (CR)
'\t'	caractere de tabulação (TAB)
'\\'	caractere de barra invertida
'\0'	caractere nulo
'\''	caractere apóstrofo
'\"'	caractere aspas

O caractere nulo ('\0') é colocado à direita da cadeia, indicando o seu final.

Cadeia	Significado	Tamanho
"ABCDE"	Cadeia de caracteres armazenando os caracteres 'A', 'B', 'C', 'D', 'E' e '\0'	6 bytes
""	Cadeia de caracteres armazenando o caractere '\0'	1 byte

1.6 Entrada e Saída Básicas

Nesta seção são apresentadas duas funções que serão utilizadas ao longo dos exercícios propostos no curso. São as funções printf() e scanf(). O conceito de funções será detalhado no Capítulo 4.

1.6.1 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 da linguagem C, sendo incluída em uma biblioteca (*stdio.h*) fornecida juntamente com os compiladores. Esta função serve para apresentar na tela uma expressão definida pelo usuário, e segue a sintaxe

```
printf("expr. de controle", argumento1, argumento2, …),
```

onde *expr. de controle* é uma expressão definida, que pode conter alguns códigos, apresentados na tabela a seguir. Quando a função printf() encontra um destes códigos, ela o substitui pelo argumento fornecido. Os argumentos podem ser nenhum ou quantos argumentos se fizerem necessários.

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)

Exemplos:

```
printf("Teste geral");
```
Saída: Teste geral

```
printf("Esta casa tem %d quartos\n", 2);
```
Saída: Esta casa tem 2 quartos

```
printf("Nome: %s\nSexo: %c\nIdade: %d\n", "Pedro", 'M', 18);
```
Saída: Nome: Pedro
Sexo: M
Idade: 18

1.6.2 A Função scanf()

A função scanf() é outra das funções de E/S (entrada e saída) que podem ser usadas em C. Lê do teclado dados e coloca os valores fornecidos pelo usuário nas variáveis utilizadas como parâmetro da função. Sua sintaxe é

```
scanf("expr. de controle", &argumento1, &argumento2, …),
```

onde a expressão de controle utiliza os mesmos códigos da função printf().

Exemplo:

```
int i, j;
float f;
char c;

scanf("%d%d", &i, &j);
scanf("%f", &f);
scanf("%c", &c);
```

O operador de endereço (&), que precede os argumentos da função, retorna o primeiro byte ocupado pela variável na memória do computador, e será detalhado no capítulo de ponteiros.

CAPÍTULO 2

OPERADORES

2.1 Operadores aritméticos

Os operadores aritméticos são os seguintes: + (adição), - (subtração), * (multiplicação) e / (divisão). No caso de divisão entre números inteiros, o resultado é truncado. O operador % fornece o resto da divisão e só funciona como operador entre tipos inteiros. Então:

Expressão	Tem o valor
22 / 3	7
22 % 3	1
14 / 4	3
14 % 4	2
21 / 7	3
21 % 7	0

2.2 Operador de atribuição

O operador de atribuição (=) copia o valor do lado direito para a variável, ou endereço, do lado esquerdo. Além disso, o operador = também retorna este valor. Isso significa que, ao fazer x = y, por exemplo, o valor da variável y será copiado na variável x, sendo este valor o resultado da operação.

Em outras palavras, se fizermos x = y = 0, será processada inicialmente a operação y = 0, atribuindo o valor 0 à variável y. Esta expressão fornece o resultado 0, que é atribuído à variável x. No final do processamento, ambas as variáveis terão o valor 0.

Expressão	Operação	Valor da expressão
i = 3	Coloca o valor 3 em i	3
i = 3 + 4	O valor 7 é colocado em i	7
i = k = 4	O valor 4 é colocado em k; o valor da atribuição (4) é então colocado em i	4
i = (k = 4) + 3	O valor 4 é colocado em k; a adição é realizada e o valor 7 é colocado em i	4

2.3 Operadores relacionais

Os operadores relacionais em C são: == (igual a), != (diferente de), > (maior que), < (menor que), >= (maior ou igual a) e <= (menor ou igual a). O resultado de dois valores conectados por um operador relacional será 0 para falso ou 1 para verdadeiro.

Expressão	Valor
5 < 3	0
3 < 5	1
5 == 5	1
3 == 5	0
5 <= 5	1

Em C, não existem variáveis lógicas. Qualquer valor pode ser testado como verdadeiro ou falso. Se ele for zero, é falso. Se for qualquer valor diferente de zero, é verdadeiro.

2.4 Operadores lógicos

Os dois operadores lógicos binários são && (e) e || (ou). O resultado de suas operações também será 0 (falso) ou 1 (verdadeiro).

Expressão	Valor
5 || 3	1
5 || 0	1
5 && 3	1
5 && 0	0
(i > 5) && (i <= 7)	1 se i for maior que 5 e menor ou igual a 7 0, qualque outro caso

O operador negação (!) inverte o sentido do valor que o segue. Qualquer valor não zero será convertido para 0 e um valor 0 será convertido para 1.

Expressão	Valor
!5	0
!0	1
!(i > 5)	1 se i não for maior que 5 0, se i for maior que 5

2.5 Operadores bit a bit

Os operadores bit a bit operam apenas com números inteiros. Os operadores são: & (e, bit a bit), | (ou, bit a bit), ^ (ou exclusivo, bit a bit), ~ (negação, bit a bit), << (deslocamento à esquerda) e >> (deslocamento à direita). São utilizados normalmente para ligar ou desligar bits ou para testar bits específicos em variáveis inteiras.

Expressão	Valor	Expressão binária	Valor binário
1 | 2	3	00000001 | 00000010	00000011
0xFF & 0x0F	0x0F	11111111 & 00001111	00001111
0x33 & 0xCC	0xFF	00110011 | 11001100	11111111
0x0F << 2	0x3C	00001111 << 2	00111100
0x1C >> 1	0x0E	00011100 >> 1	00001110

2.6 Atribuições reduzidas

C oferece muitos operadores de atribuição que são redução de outros operadores. Eles tomam a forma de *op*=, onde *op* pode ser +, -, *, /, %, <<, >>, &, ^, |. A expressão *f op= g* é análoga a *f = f op g*. Por exemplo:

Expressão	É igual a
a += 2	a = a + 2
i <<= 1	i = i << 1
s /= 7 + 2	s = s / (7 + 2)

2.7 Operadores pré e pós-fixados

Os operadores pré e pós-fixados incrementam (++) ou decrementam (--) uma variável. Uma operação prefixada é realizada antes que o valor da variável seja utilizado. Uma operação pós-fixada é efetuada após a utilização da variável. Por exemplo, para uma variável i inteira com valor 5:

Expressão	Valor de i utilizado na avaliação	Valor da expressão	Valor final de i
5 + (i++)	5	10	6
5 + (i--)	5	10	4
5 + (++i)	6	11	6
5 + (--i)	4	9	4

2.8 Operadores condicionais

Operadores condicionais são uma maneira rápida de selecionar um entre dois valores, baseado no valor de uma expressão. A sintaxe é:

```
expr1 ? expr2 : expr3
```

Se *expr1* for verdadeira (não zero), então o resultado é o valor de *expr2*. Caso contrário é o valor de *expr3*. Por exemplo:

Expressão	Valor
5 ? 1 : 2	1
0 ? 1 : 2	2
(a > b) ? a : b	1
(a > b) ? ((a > b) ? a : c) : ((b > c) ? b : c)	1

2.9 Operador vírgula

Numa seqüência de expressões separadas por vírgulas, as expressões são processadas da esquerda para a direita sendo retornado o valor da expressão mais à direita.

Expressão	Valor
5, 1, 2	2
i++, j + 2	j + 2
i++, j++, k++	valor de k (antes do incremento)
++i, ++j, ++k	valor de k (depois do incremento)

2.10 Precedência de operadores

Os operadores têm uma ordem de precedência. Isto é, sem parênteses, certas operações são efetuadas antes de outras com menor precedência. Para operadores com precedência igual, a associatividade é da esquerda para direita, com algumas exceções (mostradas na tabela abaixo). Para se ter certeza da interpretação, as expressões que se deseja interpretar primeiro devem ser agrupadas entre parênteses.

Cada conjunto de operadores na tabela possui a mesma precedência. Os símbolos ainda não mencionados serão descritos mais adiante (nos capítulos sobre funções, matrizes e estruturas).

Operador	Descrição
()	chamada de função
[]	elemento de matriz
->	ponteiro para membro de estrutura
.	membro de estrutura
!	negação lógica
~	negação bit a bit
++	incremento
--	decremento
-	menos unário
(tipo)	conversão (cast)
*	ponteiro
&	endereço
sizeof	tamanho do objeto
*	multiplicação
/	elemento de matriz
%	resto da divisão
+	adição
-	subtração
<<	deslocamento à esquerda
>>	deslocamento à direita
<	menor que
<=	menor ou igual a
>	maior que
>=	maior ou igual a
==	igualdade
!=	desigualdade
&	e, bit a bit
^	ou exclusivo, bit a bit
|	ou, bit a bit
&&	e, lógico
||	ou, lógico
? :	condicional
=	atribuição
op=	atribuição
,	vírgula

CAPÍTULO 3

CONTROLE DE FLUXO

3.1 if

3.1.1 Sintaxe

```
if (expr) comando
```

Se *expr* for verdadeira (diferente de 0), *comando* é executado.

```
if (expr) comando1 else comando2
```

Se *expr* for verdadeira, *comando1* é executado; caso contrário, *comando2* é executado.

```
if (expr1) comando1 else if (expr2) comando2 else comando3
```

Se *expr1* for verdadeira, *comando1* é executado. Se for falsa, se *expr2* for verdadeira, *comando2* é executado; caso contrário, *comando3*

3.1.2 Exemplos

```
/* Exemplo 1 */
if (a == 3)
      b = 4;


/* Exemplo 2 */
if (a > b)
      c = a * a;
else
      c = b * b;


/* Exemplo 3 */
if (a == b)
      c = 0;
else if (a > b)
      c = a * a;
else
      c = b * b;
```

Atenção:

Exemplo	**Ação**
if (a == 3) b = 4;	Testa se o valor de *a* é 3. Se for, atribui o valor 4 a *b*
if (a = 3) b = 4;	Atribui 3 à variável *a*. Testa o valor 3 (verdadeiro); portanto, independente do valor inicial de *a* será atribuído 4 a *b*

3.2 while

3.2.1 Sintaxe

```
while (expr) comando
```

Enquanto *expr* for verdadeira (diferente de 0), *comando* é executado.

Quando o programa chega na linha que contém o teste do comando **while**, ele verifica se a expressão de teste é verdadeira. Se for, o programa executa o comando uma vez e torna a testar a expressão, até que a expressão seja falsa. Somente quando isso ocorrer, o controle do programa passa para a linha seguinte ao laço.

Se, na primeira vez que o programa testar a expressão, ela for falsa, o controle do programa passa para a linha seguinte ao laço, sem executar o comando nenhuma vez.

O corpo de um laço **while** pode ter um único comando terminado por ponto-e-vírgula, vários comandos entre chaves ou ainda nenhuma instrução, mantendo o ponto-e-vírgula.

3.2.2 Exemplo

```
int i;

i = 0;
while (i < 6)
{
    printf("%d\n", i);
    i++;
}
```

3.3 do-while

3.3.1 Sintaxe

```
do comando while (expr)
```

comando é executado enquanto *expr* for verdadeira (diferente de 0).

O laço **do-while** é bastante parecido com o laço **while**. A diferença é que no laço **do-while** o teste da condição é executado somente depois do laço ser processado. Isso garante que o laço será executado pelo menos uma vez.

3.3.2 Exemplo

```
int i;

i = 0;
do
{
    printf("%d\n", i);
    i++;
} while (i < 6)
```

3.4 for

3.4.1 Sintaxe

```
for (inicializacao, condicao, incremento) comando
```

O laço **for** é equivalente ao seguinte laço **while**:

```
inicializacao
while (condicao)
{
    comando
    incremento
}
```

3.4.2 Exemplos

```
int i;

for (i = 0; i < 6; i++)
{
    printf("%d\n", i);
}
```

Qualquer uma das expressões do laço **for** pode conter várias instruções separadas por vírgulas.

```
int i, j;

for (i = 0, j = 0; i + j < 100; i++, j+=2)
{
    printf("%d\n", i + j);
}
```

Qualquer uma das três partes de um laço **for** pode ser omitida, embora os ponto-e-vírgulas devam permanecer. Se a expressão de teste for omitida, é considerada verdadeira.

```
int i = 0;

for (; i < 100;)
{
    printf("%d\n", i++);
}
```

3.5 break

O comando **break** pode ser utilizado no corpo de qualquer estrutura de laço C (**while**, **do-while** e **for**). Causa a imediata saída do laço e o controle passa para o próximo estágio do programa.

3.5.1 Exemplo

```
int i = 0;

while(1)
{
    printf("%d\n", i++);
    if (i >= 6) break;
}
```

3.6 switch

3.6.1 Sintaxe

```
switch(expr)
{
    case constante1:
        comando1;       /*opcional*/
    case constante2:
        comando2;       /*opcional*/
    case constante3:
        comando3;       /*opcional*/
    default:            /*opcional*/
        comando4;       /*opcional*/
}
```

O comando **switch** verifica o valor de *expr* e compara seu valor com os rótulos dos casos. *expr* deve ser inteiro ou caractere.

Cada caso deve ser rotulado por uma constante do tipo inteiro ou caractere. 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 entre chaves.

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

Se nenhum caso for satisfeito, o controle do programa sai do bloco **switch**, a menos que exista um caso *default*. Se existir, a execução começa nele.

Os rótulos dos casos devem ser todos diferentes.

O comando **break** causa uma saída imediata do programa. Se não houver um **break** seguindo as instruções do caso, o programa segue executando todas as instruções dos casos abaixo, até encontrar um **break** ou o fim do corpo do **switch**.

3.6.2 Exemplo

```
int mes;

 ...

switch(mes)
{
    case 1:
        printf("Janeiro\n");
        break;
    case 2:
        printf("Fevereiro\n");
        break;
    case 3:
        printf("Marco\n");
        break;
    case 4:
        printf("Abril\n");
        break;
```

```
        case 5:
            printf("Maio\n");
            break;
        case 6:
            printf("Junho\n");
            break;
        case 7:
            printf("Julho\n");
            break;
        case 8:
            printf("Agosto\n");
            break;
        case 9:
            printf("Setembro\n");
            break;
        case 10:
            printf("Outubro\n");
            break;
        case 11:
            printf("Novembro\n");
            break;
        case 12:
            printf("Dezembro\n");
            break;
        default:
            printf("O numero nao equivale a nenhum mes\n");
            break;
    }
```

CAPÍTULO 4

FUNÇÕES

4.1 Definição de Função

Funções servem para dividir um grande programa em diversas partes menores. Além disso, permitem que sejam utilizadas partes de programa desenvolvidas por outras pessoas, sem que se tenha acesso ao código-fonte. Como exemplo, em capítulos anteriores foi utilizada a função printf() sem que fossem conhecidos detalhes de sua programação.

Programas em C geralmente utilizam diversas pequenas funções, ao invés de poucas e grandes funções. Ao dividir um programa em funções, diversas vantagens são encontradas, como impedir que o programador tenha que repetir o código diversas vezes, facilitar o trabalho de encontrar erros no código.

Diversas funções são fornecidas juntamente com os compiladores, e estão presentes na norma ANSI, como funções matemáticas (seno, coseno, potência, raiz quadrada etc.), funções de entrada e saída (scanf(), printf() e outras), entre outras.

Uma função tem a sintaxe

```
tipo nome(argumentos)
{
    declarações de variáveis
    comandos
}
```

onde:

tipo determina o tipo do valor de retorno (se omitido, é assumido int);
nome representa o nome pelo qual a função será chamada ao longo do programa;
argumentos são informações externas transmitidas para a função (podem não existir).

Todo programa é composto de funções, sendo iniciada a execução pela função de nome main().

4.1.1 Variáveis Locais

A declaração das variáveis, em C, deve vir no início da função, antes de qualquer comando. Uma variável declarada dentro do corpo de uma função é local, ou seja, só existe dentro da função. Ao ser iniciada a função, a variável é criada. Quando a função termina, a variável é apagada, sendo liberado seu espaço ocupado na memória.

4.1.2 Chamando Funções

Para executar uma função, ela deve ser chamada no corpo de uma outra função (à exceção da função main(), que é executada no início do programa). Uma chamada de função é feita escrevendo-se o nome da função seguido dos argumentos fornecidos, entre parênteses. Se não houver argumentos, ainda assim devem ser mantidos os parênteses, para que o compilador diferencie a chamada da função de uma variável. O comando de chamada de uma função deve ser seguido de ponto-e-vírgula.

As funções só podem ser chamadas depois de terem sido declaradas. Caso sejam chamadas sem que tenham sido declaradas, um erro de compilação ocorre.

4.1.3 Programa Exemplo

O programa abaixo é composto de duas funções: main() e linha().

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

main()
{
    linha();
    printf("Programa exemplo de funcoes \n");
    linha();
}
```

```
SAIDA
------------------------------
Programa exemplo de funcoes
------------------------------
```

A função linha(), apresentada neste exemplo, escreve uma linha na tela. Esta função chama uma outra função, printf(), da biblioteca C. Uma função pode conter em seu corpo chamadas a outras funções.

A função main() apresenta duas chamadas à função linha().

4.2 Argumentos

Argumentos são utilizados para transmitir informações para a função. Já foram utilizados anteriormente nas funções printf() e scanf().

Uma função pode receber qualquer número de argumentos, sendo possível escrever uma função que não receba nenhum argumento. No caso de uma função sem argumentos pode-se escrevê-la de duas formas: deixando a lista de argumentos vazia (mantendo entretanto os parênteses) ou colocando o tipo *void* entre parênteses.

O quinto tipo existente em C, *void* (vazio, em inglês), é um tipo utilizado para representar o nada. Nenhuma variável pode ser declarada como sendo do tipo void. A função main(), já utilizada em capítulos anteriores, é um exemplo de função sem argumentos.

Exemplo: o programa abaixo utiliza a função EscreveCaractere(). Esta função recebe como argumento uma variável caractere (ch) e uma variável inteira (n) e faz com que o caractere *ch* seja impresso *n* vezes.

```
EscreveCaractere(char ch, int n)
{
    int i;

    for(i = 0; i < n; i++)
    {
        printf("%c", ch);
    }
}
```

```
main()
{
    EscreveCaractere('-', 27);
    printf("\nPrograma exemplo de funcoes\n");
    EscreveCaractere('-', 27);
    EscreveCaractere('\n', 3);
    printf("Teste concluido\n");
}
```

Nota-se que no exemplo inicialmente é definida a função EscreveCaractere() para, somente depois, ser definida a função main(), que acessa a função EscreveCaractere(). Caso a função main() venha primeiro, quando o compilador tentar compilar a linha que chama a função EscreveCaractere(), o compilador não reconhecerá a função.

Caso deseje-se definir a função EscreveCaractere() antes da função main(), deve-se inicialmente declarar a função EscreveCaractere(). A declaração de uma função consiste em escrevê-la da mesma forma que na definição, sem o corpo da função e seguida por ponto-e-vírgula.

O exemplo anterior ficaria da seguinte maneira:

```
EscreveCaractere(char ch, int n);

main()
{
    EscreveCaractere('-', 27);
    printf("\nPrograma exemplo de funcoes\n");
    EscreveCaractere('-', 27);
    EscreveCaractere('\n', 3);
    printf("Teste concluido\n");
}

EscreveCaractere(char ch, int n)
{
    int i;

    for(i = 0; i < n; i++)
        printf("%c", ch);
}
```

Quando o compilador encontra a primeira linha do código, ele entende que a função EscreveCaractere() existe e tem a forma apresentada na declaração, mas ainda não está definida; será definida em algum lugar do código. Portanto, ele consegue compilar as chamadas à esta função no resto do programa.

4.3 Valor de Retorno

Valor de retorno é o valor que uma função retorna para a função que a chamou. Seu tipo é fornecido antes do nome da função na declaração.

Exemplo: a função Quadrado() recebe como argumento uma variável real (a) e retorna o quadrado dela.

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

Na função, são fornecidos o tipo do valor de retorno (float), o nome da função (Quadrado) e o argumento (a, do tipo float). No corpo da função, é encontrado o comando **return**. Este comando fornece o valor de retorno da função, fazendo com que ela termine. Uma função pode ter diversos comando **return**.

Exemplo: a função Maximo() retorna o maior valor entre dois números.

```
int Maximo(int a, int b)
{
    if(a > b)
        return a;
    else
        return b;
}
```

Esta outra função possui dois argumentos inteiros (*a* e *b*) sendo o valor de retorno também inteiro. Este é um exemplo de função que possui mais de um comando **return**.

A função EscreveCaractere(), do exemplo fornecido na seção 4.2, é um exemplo de função que não possui retorno. No caso, se for omitido o valor de retorno de uma função, este valor é assumido como int. Se uma função não retorna nada, seu tipo de retorno deve ser definido como **void**.

O comando **return** pode ser utilizado numa função com tipo de retorno **void**. neste caso, o comando não deve retornar nenhum valor, sendo chamado simplesmente seguido do ponto-e-vírgula.

4.4 Recursividade

Como foi visto nas seções anteriores, uma função pode conter em seu corpo chamadas a outras funções. Nesta seção veremos um caso particular em que uma função é chamada por ela própria. A este caso, dá-se o nome de *recursividade*.

Exemplo: Função fatorial(), utilizando recursividade.

Sabe-se que N! = N * (N - 1)! e que 0! = 1.

```
int fatorial(int n)
{
    if(n > 0)
        return n * fatorial(n - 1);
    else
        return 1;
}
```

Note que existe dentro da função recursiva uma possibilidade de o programa sair dela sem que ocorra a chamada à própria função (no caso de n menor ou igual a zero a função retorna 1). Se não existisse esta condição, a função se chamaria infinitamente, causando o travamento do programa.

4.5 Classes de Armazenamento

Todas as variáveis e funções C têm dois atributos: um tipo e uma classe de armazenamento. Os tipos, como visto anteriormente, são cinco: int, long, float, double e void. As classes de armazenamento são quatro: auto (automáticas), extern (externas), static (estáticas) e register (em registradores).

4.5.1 Classe de Armazenamento - auto

As variáveis declaradas até agora nos exemplos são acessíveis somente na função a qual ela pertence. Estas variáveis são automáticas caso não seja definida sua classe de armazenamento. Ou seja, o código

```
main()
{
    auto int n;
    . . .
}
```

é equivalente a

```
main()
{
    int n;
    . . .
}
```

4.5.2 Classe de Armazenamento - extern

Todas as variáveis declaradas fora de qualquer função são externas. Variáveis com este atributo serão conhecidas por todas as funções declaradas depois delas.

```
/* Testa o uso de variaveis externas */
int i;

void incrementa(void)
{
    i++;
}

main()
{
    printf("%d\n", i);
    incrementa();
    printf("%d\n", i);
}
```

4.5.3 Classe de Armazenamento - static

Variáveis estáticas possuem dois comportamentos bem distintos: variáveis estáticas locais e estáticas externas.

Quando uma variável estática é declarada dentro de uma função, ela é local àquela função. Uma variável estática mantém seu valor ao término da função. O exemplo a seguir apresenta uma função contendo uma variável estática.

```
void soma(void)
{
    static int i = 0;
    printf("i = %d\n", i++);
}

main()
{
    soma();
    soma();
    soma();
}
```

A saída será:

```
i = 0
i = 1
i = 2
```

Como a variável i é estática, não é inicializada a cada chamada de soma().

O segundo uso de **static** é associado a declarações externas. Declarar uma variável estática externa indica que esta variável será global apenas no arquivo fonte no qual ela foi declarada, e só a aprtir de sua declaração. Esta declaração é usada como mecanismo de privacidade e impede que outros arquivos tenham acesso à variavél.

4.5.4 Classe de Armazenamento - register

A classe de armazenamento **register** indica que a variável será guardada fisicamente numa memória de acesso ,uito mais rápido chamada registrador. Somente podem ser armazenadas no registrador variáveis do tipo **int** e **char**. Basicamente, variáveis **register** são usadas para aumentar a velocidade de processamento.

4.6 O Pré-processador C

O pré-processador C é um programa que examina o programa-fonte em C e executa certas modificações nela, baseado em instruções chamadas diretivas. É executado automaticamente antes da compilação.

4.6.1 A Diretiva #define

A diretiva **#define** substitui determinados códigos predefinidos pelo usuário. Possui a forma

```
#define nome xxxxx
```

onde *nome* é o símbolo a ser trocado e *xxxxx* é qualquer expressão. O comando pode continuar em mais de uma linha utilizando-se a barra invertida (\) para indicar a continuação na linha de baixo.

Ao ser criado um **#define**, o precompilador substitui todas as ocorrências de *nome* no código fonte pela expressão *xxxxx* fornecida.

Exemplo: a seguir definimos algumas constantes simbólicas.

```
#define PI 3.14159
#define ON 1
#define OFF 0
#define ENDERECO 0x378

void main()
{
...
}
```

No exemplo acima, definimos PI como 3.14159. Isto significa que em todas as ocorrências do texto, PI será trocado por 3.14159. Assim, a instrução

```
area = PI * raio * raio;
```

será interpretada pelo compilador como se fosse escrita assim:

```
area = 3.14159 * raio * raio;
```

Não é utilizado ponto-e-vírgula no comando #define. Se for utilizado, o precompilador incluirá o ponto-e-vírgula na substituição.

#define pode ter argumentos. Eles são fornecidos imediatamente após o nome:

```
#define nome(arg1,arg2,...) xxxxx
```

Dado:

```
#define MAIS_UM(x) x+1
```

então:

```
MAIS_UM(y)
```

será substituído por:

```
y+1
```

#defines podem referenciar **#define**s anteriores. Por exemplo:

```
#define PI    3.141592653589793
#define PI_2  2*PI
```

Um uso típico de **#define** e substituir algumas chamadas a funções por código direto. Vamos supor, por exemplo, que quiséssemos criar uma macro equivalente a uma função que multiplique dois números. A macro

```
#define PRODUTO(a,b) a*b
```

parece atender ao problema. Vamos testar:

Expressão original	**Expressão pré-compilada**
resposta = PRODUTO(a,b);	resposta = a*b;
resposta = PRODUTO(x,2);	resposta = x*2;
resposta = PRODUTO(a+b,2);	resposta = a+b*2;

Para as duas primeiras expressões, a macro funcionou bem. Já para a terceira, a pricípio desejaríamos o produto de a + b por 2. A expressão pré-compilada não fornece isso, já que o operador * é processado antes do operador +. A solução para evitar este problema é por os argumentos na definição entre parênteses, ou seja:

```
#define PRODUTO(a,b) (a)*(b)
```

Novamente, parece atender ao problema. Vamos testar:

Expressão original	**Expressão pré-compilada**
resposta = PRODUTO(a,b);	resposta = (a)*(b);
resposta = PRODUTO(x,2);	resposta = (x)*(2);
resposta = PRODUTO(a+b,2);	resposta = (a+b)*(2);
resposta = PRODUTO(2,4) / PRODUTO(2,4);	resposta = (2)*(4)/(2)*(4);

Esta macro continua atendendo às duas primeiras expressões, passa a atender à terceira. Entretanto, para a quarta expressão, não obtivemos o resultado desejado. A princípio, desejaríamos efetuar os dois produtos, que são idênticos, dividir um pelo outro, obtendo como resposta 1`. Ao utilizar a macro, como os operadores * e / possuem a mesma precedência, eles são executados na ordem em que são encontrados. Ou seja, a divisão é feita antes do segundo produto ter sido efetuado, o que fornece o resultado 16 para a expressão. A solução para este problema é ser redundante no uso de perênteses ao se construir macros. A macro PRODUTO é definida então por

```
#define PRODUTO(a,b) ((a)*(b))
```

Ainda assim, colocar os parênteses em macros não atende a todos os casos. Por exemplo, a macro

```
#define DOBRO(a) ((a)+(a))
```

apesar de conter todos os parênteses necessários, se for utilizada na expressão

```
resposta = DOBRO(++x)
```

criará o código pré-compilado

```
resposta = (++x)+(++x)
```

que incrementará a variável x duas vezes, ao invés de uma.

Ou seja, **MACROS NÃO SÃO FUNÇÕES**. Tenha sempre cuidado em sua utilização.

4.6.2 A Diretiva #undef

A diretiva **#undef** tem a seguinte sintaxe:

```
#undef nome
```

Esta diretiva anula uma definição de **#define**, o que torna *nome* indefinido.

4.6.3 A Diretiva #include

A instrução **#include** inclui outro arquivo fonte dentro do arquivo atual. Possui duas formas:

```
#include "nome-do-arquivo"
#include <nome-do-arquivo>
```

A primeira inclui um arquivo que será procurado no diretório corrente de trabalho. A segunda incluirá um arquivo que se encontra em algum lugar predefinido.

Os arquivos incluídos normalmente possuem **#define**s e declarações de funções. Os compiladores fornecem uma série de funções que são definidas em arquivos de cabeçalho, com extensão .h, que posem ser incluídos no código-fonte escrito pelo programador. São alguns exemplos a biblioteca matemática *math.h* e a biblioteca de entrada e saída padrão *stdio.h*.

4.6.4 Outras Diretivas

O preprocessador C possui ainda outras diretivas de pré-compilação. Alguns trechos do código podem ser ignorados pelo compilador, dependendo de alguns parâmetros de pré-compilação.

#if *expr*	testa para verificar se a expressão com constantes inteiras *exp* é diferente de zero
#ifdef *nome*	testa a ocorrência de *nome* com **#define**
#ifndef *nome*	testa se *nome* não está definido (nunca foi definido com **#define** ou foi eliminado com **#undef**)

Se os resultados destes testes forem avaliados como verdadeiros, então as linhas seguintes são compiladas até que um **#else** ou um **#endif** seja encontrado.

#else	sinaliza o começo das linhas a serem compiladas se o teste do **if** for falso
#endif	finaliza os comandos **#if**, **#ifdef** ou **#ifndef**

Exemplo: a seqüência seguinte permite que diferentes códigos sejam compilados, baseado na definição de COMPILADOR.

```
#define COMPILADOR TC
  /* Esta definicao precisa ser trocada para cada compilador*/

#if COMPILADOR == TC
  /* Codigo para o Turbo C
#endif

#if COMPILADOR == GCC
  /* Codigo para o Gnu C
#endif
```

CAPÍTULO 5

VETORES E MATRIZES

5.1 Vetores

Um vetor armazena uma determinada quantidade de dados de mesmo tipo. Vamos supor o problema de encontrar a média de idade de 4 pessoas. O programa poderia ser:

```
main()
{
  int idade0, idade1, idade2, idade3;

  printf("Digite a idade da pessoa 0: ");
  scanf("%d", &idade0);
  printf("Digite a idade da pessoa 1: ");
  scanf("%d", &idade1);
  printf("Digite a idade da pessoa 2: ");
  scanf("%d", &idade2);
  printf("Digite a idade da pessoa 3: ");
  scanf("%d", &idade3);
  printf("Idade media: %d\n", (idade0+idade1+idade2+idade3) / 4);
}
```

Suponhamos agora que desejássemos encontrar a média das idades de 500 pessoas. A tarefa passa a ser bem mais trabalhosa, sendo em diversos casos impraticável se resolver da maneira apresentada acima.

A solução para este problema é a utilização de vetores. Um vetor é uma série de variáveis de mesmo tipo referenciadas por um único nome, onde cada variável é diferenciada através de um índice, que é representado entre colchetes depois do nome da variável.

A declaração

```
int idade[4];
```

aloca memória para armazenar 4 inteiros e declara a variável idade como um vetor de 4 elementos.

O programa da média das idades poderia ser substituído por:

```
main()
{
  int idade[4], i, soma = 0;

  for(i = 0; i < 4; i++)
  {
    printf("Digite a idade da pessoa %d: ", i);
    scanf("%d", &idade[i]);
  }
  for(i = 0; i < 4; i++)
    soma += idade[i];
  printf("Idade media: %d\n", soma / 4);
}
```

Ao escrever o mesmo programa utilizando vetores, a complexidade do código fica independente do tamanho do vetor.

Para acessar um determinado elemento do vetor, deve-se chamar o elemento pelo nome da variável seguido do índice, entre colchetes. Note que o índice começa em 0, e não em 1. Ou seja, o elemento

```
idade[2]
```

não é o segundo elemento e sim o terceiro, pois a numeração começa de 0. Em nosso exemplo utilizamos uma variável inteira, i, como índice do vetor.

Suponhamos o seguinte programa:

```
main()
{
  int idade[5];

  idade[0] = 15;
  idade[1] = 16;
  idade[2] = 17;
  idade[3] = 18;
  idade[4] = 19;
  idade[5] = 20;
}
```

Nota-se que neste programa, definimos a variável *idade* como um vetor de 5 elementos (0 a 4). Definimos os elementos do vetor mas, na última definição, ultrapassamos o limite do vetor. A linguagem C não realiza verificação de limites em vetores. Quando o vetor foi definido, o compilador reservou o espaço de memória equivalente a 5 variáveis inteiras, ou seja, 10 bytes. Quando tentamos acessar um elemento que ultrapasse o limite de um vetor, estamos acessando uma região de memória que não pertence a esse vetor.

5.1.1 Inicialização

A linguagem C permite que vetores sejam inicializados. No caso, será inicializada uma variável contendo o número de dias de cada mês:

```
int numdias[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

O número de elementos do vetor pode ser omitido, ou seja

```
int numdias[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Se nenhum número for fornecido para inicializar o vetor, o compilador contará o número de itens da lista de inicialização e o fixará como dimensão do vetor.

Na falta de inicialização explícita, variáveis **extern** e variáveis **static** são inicializadas com valor zero; variáveis automáticas têm valor indefinido (isto é, lixo).

Se há menos inicializadores que a dimensão especificada, os outros serão zero. Se há mais inicializadores que o especificado, o compilador acusa um erro.

Em C não há como se especificar a repetição de um inicializador, nem de se inicializar um elemento no meio de um vetor sem inicializar todos os elementos anteriores ao mesmo tempo.

5.1.2 Vetores como argumentos de funções

Vetores podem ser passados também como argumentos de funções. Vamos rescrever o programa que calcula a média das idades para um programa contendo uma função que receba um vetor e retorne a sua média.

```
int media(int valor[], int nElementos)
{
  int i, soma = 0;

  for(i = 0; i < nElementos; i++)
    soma += valor[i];
  return soma / nElementos;
}

main()
{
  int idade[5], i;

  for(i = 0; i < 5; i++)
  {
    printf("Digite a idade da pessoa %d: ", i);
    scanf("%d", &idade[i]);
  }
  printf("Idade media: %d\n", media(idade, 5));
}
```

Note que na declaração da função, o argumento que representa o vetor é declarado com colchetes. Além dele, passamos como argumento da função também o tamanho do vetor. Sem ele, a função não tem como saber o tamanho do vetor que foi passado a ela como argumento.

Na função main(), chamamos a função media() passando dois atributos: idade e 5. Para acessarmos um elemento do vetor, escrevemos após o seu nome o índice do elemento desejado entre colchetes (por exemplo, idade[0]). A chamada ao nome de um vetor sem que seja fornecido o índice de um determinado elemento representa o primeiro endereço de memória acessado por esse vetor. Ou seja, ao passarmos um vetor como argumento da função, o compilador não cria uma cópia do vetor para ser utilizada na função. É o próprio vetor quem é passado como argumento e pode ser alterado na função.

5.2 Vetores de caracteres

O vetor de caracteres, também conhecidos como *string*, é uma das formas de dados mais importantes da linguagem C. É usado para manipular texto, como palavras, nomes e frases. Em algumas linguagens, como Pascal e Basic, string é um tipo primitivo. Em C, é tratada como um vetor de elementos do tipo **char**. Cada elemento pode ser acessado individualmente, como em qualquer vetor, através do uso de colchetes.

Sempre que o compilador encontra qualquer coisa entre aspas, ele reconhece que se trata de uma string constante, isto é, os caracteres entre aspas mais o caractere nulo ('\0'). Para declarar uma variável do tipo string, deve-se declará-la como um vetor de caracteres, ou seja:

```
char Nome[50];
```

Esta declaração cria a variável Nome como string, ou vetor de caracteres, permitindo um comprimento de até 50 caracteres. Vale lembrar que o último caractere de uma string deve ser o caractere nulo, ou seja, temos 49 caracteres para trabalhar livremente. A declaração:

```
char Nome[5] = "Pedro";
```

é inválida, por não ter sido reservado espaço suficiente para as 6 variáveis ('P', 'e', 'd', 'r', 'o' e '\0').

Para manipulação de strings, são fornecidas diversas funções na biblioteca C (arquivo string.h). Algumas dessas funções são descritas a seguir. A sintaxe apresentada para cada função não é exatamente a sintaxe fornecida pela biblioteca, mas representa basicamente o que a função executa e já foi explicado até o presente momento.

5.2.1 A função strlen()

A função strlen() retorna o número de caracteres da string, sem contar o caractere nulo.

Sintaxe:

```
int strlen(char[] s);
```

Exemplo:

```
main()
{
    char Nome[6] = "Navio";

    printf("%d\n%d\n", strlen("Barco a vela"), strlen(Nome));
}

SAÍDA
12
5
```

Note que os espaços fazem parte da string, e são simplesmente caracteres, assim como letras e algarismos.

5.2.2 A função strcmp()

A função strcmp() compara duas strings. Retorna

- um valor menor que zero se a primeira string for menor que a segunda;
- zero se as strings forem iguais;
- um valor maior que zero se a primeira string for maior que a segunda.

Entende-se pôr comparação entre strings, sua posição em ordem alfabética. A ordem alfabética é baseada na tabela ASCII (Anexo A). Portanto, cuidado ao comparar maiúsculas com minúsculas, pois na tabela ASCII as letras maiúsculas possuem um valor menor que as letras minúsculas, ou seja, o caractere 'Z' vem antes do caractere 'a'.

Sintaxe:

```
int strcmp(char[] s1, char[] s2);
```

Exemplo:

```
main()
{
    printf("%d\n", strcmp("Ariranha", "Zebra"));
    printf("%d\n", strcmp("Zebra", "Ariranha"));
    printf("%d\n", strcmp("Zebra", "Zebra"));
}
```

```
SAÍDA
(Algum valor menor que 0)
(Algum valor maior que 0)
0
```

Espaços e outros caracteres especiais também são considerados na comparação, algumas vezes não fornecendo o que se espera de uma ordenação alfabética.

5.2.3 A função strcpy()

A função strcpy() copia o conteúdo de uma string para outra. A string de destino já tem que ter espaço reservado na memória antes de ser chamada a função.

Sintaxe:

```
void strcpy(char[] destino, char[] origem);
```

Exemplo:

```
main()
{
    char Nome[10];

    strcpy(Nome, "Teste")
    printf("%s\n", Nome);
}

SAÍDA
Teste
```

5.2.4 A função strcat()

A função strcat() concatena duas strings, ou seja, anexa o conteúdo de uma na outra. Similarmente à função strcpy(), a string de destino tem que ter espaço reservado na memória.

Sintaxe:

```
void strcat(char[] destino, char[] origem);
```

Exemplo:

```
main()
{
    char Nome[12];

    strcpy(Nome, "Teste")
    strcpy(Nome, "geral")
    printf("%s\n", Nome);
}
SAÍDA
Testegeral
```

5.3 Matrizes

A linguagem C permite vetores de qualquer tipo, inclusive vetores de vetores. Por exemplo, uma matriz é um vetor em que seus elementos são vetores. Com dois pares de colchetes, obtemos uma

matriz de duas dimensões e, para cada par de colchetes adicionado, obtemos uma matriz com uma dimensão a mais.

Por exemplo, a declaração

```
int A[5][6];
```

indica que A é uma matriz 5x6 e seus elementos são inteiros.

5.3.1 Inicialização

As matrizes são inicializadas como os vetores, ou seja, os elementos são colocados entre chaves e separados por vírgulas. Como seus elementos são vetores, estes, por sua vez, também são inicializados com seus elementos entre chaves e separados por vírgulas. Por exemplo:

```
int A[5][6] = { { 1,  2,  3,  4,  5,  6},
                { 7,  8,  9, 10, 11, 12},
                {13, 14, 15, 16, 17, 18},
                {19, 20, 21, 22, 23, 24},
                {25, 26, 27, 28, 29, 30} };
```

Caso as matrizes sejam de caracteres, isto é equivalente a termos um vetor de strings. Sua inicialização pode se dar da forma

```
char Nome[5][10] = {"Joao",
                    "Jose",
                    "Maria",
                    "Geraldo",
                    "Lucia"};
```

A matriz será inicializada como

J	o	a	o	\0	\0	\0	\0	\0	\0
J	o	s	e	\0	\0	\0	\0	\0	\0
M	a	r	i	a	\0	\0	\0	\0	\0
G	e	r	a	l	d	o	\0	\0	\0
L	u	c	i	a	\0	\0	\0	\0	\0

5.3.2 Matrizes como argumento de funções

A passagem de uma matriz para uma função é similar à passagem de um vetor. O método de passagem do endereço da matriz para a função é idêntico, não importando quantas dimensões ela possua, já que sempre é passado o endereço da matriz.

Entretanto, na declaração da função, a matriz é um pouco diferente. A função deve receber o tamanho das dimensões a partir da segunda dimensão. Por exemplo:

```
void Determinante(double A[][5]);
```

Note que é fornecido a segunda dimensão da matriz. Isto é necessário para que, ao chamarmos o elemento A[i][j], a função saiba a partir de que elemento ela deve mudar de linha. Com o número de elementos de cada linha, a função pode obter qualquer elemento multiplicando o número da linha pelo tamanho de cada linha e somando ao número da coluna.

Por exemplo, o elemento A[2][3], é o elemento de índice 13, já que 2 * 5 + 3 = 13.

CAPÍTULO 6

PONTEIROS

6.1 Definição

Uma variável declarada como ponteiro armazena um endereço de memória. A declaração

```
int *a;
```

indica que a variável a é um ponteiro que aponta para um inteiro, ou seja, armazena o endereço de uma variável inteira.

Dois operadores são utilizados para se trabalhar com ponteiros. Um é o operador de endereço (**&**), que retorna o endereço de memória da variável. Este operador já foi utilizado em capítulos anteriores. O outro é o operador indireto (*****) que é o complemento de (*****) e retorna o conteúdo da variável existente no endereço (ponteiro). O exemplo a seguir ilustra a utilização desses operadores.

```
#include <stdio.h>

void main(void)
{
  int* a;
  int b = 2;
  int c;

  a = &b;
  c = b;
  printf("%d %d %d\n", *a, b, c);
  b = 3;
  printf("%d %d %d\n", *a, b, c);
}

SAIDA
2 2 2
3 3 2
```

No exemplo acima, as variáveis *b* e *c* são declaradas como inteiras, ou seja, cada uma delas ocupa dois bytes, em endereços de memória diferentes. A variável *a* é declarada como um ponteiro que aponta para um inteiro. O comando `a = &b` indica que a variável *a* armazenará o endereço da variável *b*. Já o comando `c = b` indica que a variável *c*, que fica armazenada em outro endereço de memória, guardará o valor da variável *b*.

A primeira linha de impressão nos fornece o conteúdo do ponteiro *a*, obtido através do operador indireto (*****), e o valor das variáveis *b* e *c*. Note que o conteúdo do ponteiro *a* é o valor da variável *b*, já que ele aponta para o endereço da variável.

Ao alterarmos o valor da variável *b*, o conteúdo de seu endereço é alterado. A variável *c*, por ocupar outro endereço de memória, permanece inalterada. Já o conteúdo do ponteiro *a* foi alterado, já que aponta para o endereço da variável *b*.

6.2 Passagem de Argumentos por Endereço

Até o momento, utilizamos funções e seus argumentos em todos os casos foram passados por valor, isto é, é criada uma cópia da variável dentro da função. Nesta seção veremos a passagem de argumentos por endereço, onde a própria variável é utilizada na função, e não uma cópia dela.

Suponha o seguinte programa:

```
#include <stdio.h>

void troca(int a, int b)
{
  int aux;

  aux = a;
  a = b;
  b = aux;
}

void main(void)
{
  int a = 2;
  int b = 3;

  printf("%d %d\n", a, b);
  troca(a, b);
  printf("%d %d\n", a, b);
}
```

A função troca() tem como objetivo receber duas variáveis inteiras e trocar o seu valor. Escrita da maneira proposta no exemplo acima, ao chamarmos a função, são criadas cópias das variáveis *a* e *b* dentro da função troca(). Estas cópias, que só existem dentro da função, tem o seu valor trocado, mas isso não implica nenhuma alteração nas variáveis *a* e *b* da função main().

Para que possamos alterar o valor de argumentos dentro de uma função, é necessário que estes argumentos sejam passados por endereço. Desta forma, a função trabalhará com a própria variável fornecida como argumento. O programa é reescrito como:

```
#include <stdio.h>

void troca(int* pa, int* pb)
{
  int aux;

  aux = *pa;
  *pa = *pb;
  *pb = aux;
}

void main(void)
{
  int a = 2;
  int b = 3;

  printf("%d %d\n", a, b);
  troca(&a, &b);
  printf("%d %d\n", a, b);
}
```

Percebem-se algumas alterações, descritas a seguir.

- a função troca() passa a receber argumentos do tipo **int***, ao invés de **int**. Isto é necessário para que a função receba o endereço das variáveis fornecidas, e possa alterar-lhes o valor;
- dentro da função, as variáveis *a* e *b* são tratadas com o operador indireto (*****), para que possamos trabalhar com seu conteúdo;
- Na chamada da função troca(), são passados como argumentos os endereços das variáveis, através do operador endereço (**&**).

Neste curso nós já passamos argumentos para funções através de seu endereço. Isto foi utilizado na função scanf(). O motivo para utilizarmos o operador endereço (**&**) na função scanf() e não o utilizarmos na função printf() é que somente a primeira altera o valor das variáveis que foram passadas como argumento.

6.3 Operações com ponteiros

A linguagem C oferece cinco operações básicas que podem ser executadas em ponteiros. O próximo programa mostra estas possibilidades. Para mostrar o resultado de cada operação, o programa imprimirá o valor do ponteiro (que é um endereço), o valor armazenado na variável apontada e o endereço do próprio ponteiro.

```
#include <stdio.h>

void main(void)
{
  int x = 5;
  int y = 6;
  int* px;
  int* py;

  /* Atribuicao de ponteiros */
  px = &x;
  py = &y;
  /* Comparacao de ponteiros */
  if(px < py)
    printf("py-px = %u\n", py - px);
  else
    printf("px-py = %u\n", px - py);
  printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
  printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);
  px++;
  printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
  py = px + 3;
  printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);
  printf("py-px = %u\n", py - px);
}

SAIDA

py-px = 1
px = 65492, *px = 5, &px = 65496
py = 65494, *py = 6, &py = 65498
px = 65494, *px = 6, &px = 65496
py = 65500, *py = -24, &py = 65498
py-px = 3
```

6.3.1 Atribuição

Um endereço pode ser atribuído a um ponteiro através do operador igual (=). No exemplo, atribuímos a *px* o endereço de *x* e a *py* o endereço de *y*.

6.3.2 Conteúdo

O operador indireto (*****) fornece o valor armazenado no endereço apontado.

6.3.3 Endereço

Como todas as variáveis, os ponteiros têm um endereço e um valor. Seu valor é o endereço de memória apontado. O operador (**&**) retorna o endereço de memória utilizado para armazenar o ponteiro. Em resumo:

- o nome do ponteiro fornece o endereço para o qual ele aponta;
- o operador **&** junto ao nome do ponteiro retorna o endereço do ponteiro;
- o operador ***** junto ao nome do ponteiro fornece o conteúdo da variável apontada.

6.3.4 Soma e diferença

Ponteiros permitem os operadores soma (+), diferença (**-**), incremento (++) e decremento (--). Estas operações não são tratadas como soma e diferença comuns entre inteiros. Elas levam em conta o espaço de memória utilizado pela variável apontada.

No exemplo, a variável *px* antes do incremento valia. 65492. Depois do incremento ela passou a valer 65494 e não 65493. O incremento de um ponteiro faz com que ele seja deslocado levando-se em conta o tamanho da variável apontada. No caso de um inteiro, o ponteiro será deslocado 2 bytes, que é o que ocorre no exemplo.

Assim como no incremento, o decremento, a soma e a diferença levam em conta o tamanho da variável apontada. Assim, com *py* = *px* + 3, a variável *py* passou a apontar para um endereço de memória localizado 6 bytes depois de *px*.

6.3.5 Comparações

Os operadores ==, !=, >, <, <=, >= são aceitos entre ponteiros e comparam os endereços de memória.

6.3.6 Ponteiros para void

Como dito anteriormente, ponteiros podem apontar para qualquer tipo de variável. Podemos ter inclusive ponteiros que apontam para o tipo **void**. Estes ponteiros armazenam um endereço de memória, sem que seja definido qual o tipo de variável que ocupa aquele espaço de memória.

Em ponteiros para **void** (**void***), não é possível a utilização do operador indireto (*****), uma vez que o compilador não sabe para qual o tipo de variável ele deva converter o conteúdo do endereço apontado.

Também não são permitidos os operadores de soma e diferença, já que o compilador não sabe quanto espaço ocupa a variável armazenada naquele endereço.

6.4 Ponteiros e vetores

Vetores em C são tratados pelo compilador como ponteiros. Isso significa que o nome do vetor representa o endereço ocupado pelo seu primeiro elemento. O programa a seguir exemplifica a utilização de ponteiros representando vetores:

```
#include <stdio.h>

void main(void)
{
  int  i, *px, x[] = {8, 5, 3, 1, 7};

  px = x;
  for(i = 0; i < 5; i++)
  {
    printf("%d\n", *px);
    px++;
  }
}
```

Nota-se no exemplo que fazemos com que o ponteiro *px* aponte para o vetor *x*. Repare que não é utilizado o operador endereço. Isto ocorre porque o nome do vetor já é tratado pelo compilador como um endereço. No caso, seria possível ao invés de

```
px = x;
```

utilizarmos

```
px = &x[0];
```

O mesmo resultado seria obtido, já que o nome do vetor é tratado como um ponteiro que aponta para o endereço do primeiro elemento do vetor.

A linha que incrementa o valor de *px* faz com que o ponteiro aponte para o endereço de memória do próximo inteiro, ou seja, o próximo elemento do vetor.

Escreveremos agora o mesmo programa de outra maneira:

```
#include <stdio.h>

void main(void)
{
  int  x[] = {8, 5, 3, 1, 7};
  int  i;

  for(i = 0; i < 5; i++)
    printf("%d\n", *(x + i));
}
```

Note que neste exemplo chamamos cada elemento do vetor através de **(x + i)*. Como pode-se notar,

```
*(x + 0) = *x = x[0];
*(x + 1) = x[1];
*(x + 2) = x[2];
...
*(x + indice) = x[indice].
```

Como exemplo, vamos fazer uma função equivalente à função *strcpy*, da biblioteca *string.h*.

```
int strcpy(char* s1, char* s2)
{
  /* Copia a string s2 em s1 */
  while(*s2 != '\0')
  {
    *s1 = *s2;
    s1++;
    s2++;
  }
  *s1 = '\0';
}
```

Nesta função, todos os caracteres de *s2* serão copiados em *s1*, até que seja encontrado o caractere nulo. Para passar para o caractere seguinte, foi utilizado o operador ++.

6.5 Alocação dinâmica de memória

Quando um vetor é declarado, o endereço de memória para o qual ele aponta não teve necessariamente seu espaço reservado (alocado) para o programa. Nos exemplos anteriores, os ponteiros utilizados sempre apontavam para outras variáveis existentes.

Nesta seção veremos o caso de alocação dinâmica, isto é, memória alocada em tempo de execução do programa. Para isto, usaremos as função *malloc* e *free*, presentes nas bibliotecas *stdlib.h* e *alloc.h*.

6.5.1 Função malloc

Sintaxe:

```
void* malloc(unsigned long size);
```

A função malloc aloca um espaço de memória (*size* bytes) e retorna o endereço do primeiro byte alocado. Como pode-se alocar espaço para qualquer tipo de variável, o tipo de retorno é **void***. Torna-se necessário converte-lo para um ponteiro do tipo desejado.

No caso de não haver espaço suficiente para alocar a quantidade de memória desejada, a função retorna um ponteiro para o endereço 0 (ou NULL). Caso a variável *size* fornecida seja 0, a função também retorna NULL.

Ao contrário das variáveis estaticamente alocadas apresentadas até agora, o a memória dinamicamente alocada não é automaticamente liberada. É necessário que o programador libere a memória dinamicamente alocada através da função *free*.

6.5.2 Função free

Sintaxe:

```
void free(void* block);
```

A função free desaloca a memória alocada..

6.5.3 Exemplo das funções malloc e free

A seguir será fornecido um exemplo das funções *malloc* e *free*.

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

void main(void)
{
  char* str = (char*) malloc(10);

  strcpy(str, "Alo mundo");
  printf("Variavel str: %s\n", str);
  free(str);
}
```

O programa apresentado aloca dinamicamente 10 bytes e associa o endereço do bloco alocado ao ponteiro *str*. Como a função *malloc* retorna **void***, é necessário converter o resultado para o tipo desejado, no caso **char***.

A partir daí trabalha-se com o ponteiro normalmente. Ao terminar a utilização do espaço alocado, é necessário liberar a memória, através da função *free*.

6.6 Ponteiros para ponteiros

Como apresentado anteriormente, ponteiros podem apontar para qualquer tipo de variável, inclusive para outros ponteiros. A seguir será fornecido um exemplo de uma matriz alocada dinamicamente.

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

void main(void)
{
  float** Matriz;
  int nLinhas, nColunas, i, j;

  printf("Quantas linhas?\n");
  scanf("%d", &nLinhas);
  printf("Quantas colunas?\n");
  scanf("%d", &nColunas);
  /* Aloca a matriz */
  Matriz = (float**) malloc(nLinhas * sizeof(float*));
  for(i = 0; i < nLinhas; i++)
    Matriz[i] = (float*) malloc(nColunas * sizeof(float));
  /* Define os elementos da matriz */
  for(i = 0; i < nLinhas; i++)
    for(j = 0; j < nColunas; j++)
      Matriz[i][j] = i * j;
  /* Imprime a matriz */
  for(i = 0; i < nLinhas; i++)
  {
    for(j = 0; j < nColunas; j++)
      printf("%f\t", Matriz[i][j]);
    printf("\n");
  }
  /* Desaloca a matriz */
  for(i = 0; i < nLinhas; i++)
    free(Matriz[i]);
  free(Matriz);
}
```

Note que neste exemplo não se sabe inicialmente qual será o tamanho da matriz. Caso desejássemos alocar o espaço estaticamente, seria necessário definir a matriz com número de linhas e colunas grande, de modo que a escolha do usuário não ultrapassasse entes limites. Por exemplo,

```
float Matriz[1000][1000];
```

Há dois inconvenientes imediatos de se alocar vetores e matrizes estaticamente quando não sabemos de antemão o seu tamanho: um espaço de memória é provavelmente desperdiçado; e ficamos limitados ao tamanho definido inicialmente. Alocando memória dinamicamente, somente a quantidade necessária de memória é utilizada, ao passo que nosso limite de tamanho é o limite físico de memória do computador.

Para alocar a matriz, inicialmente alocamos espaço para um vetor de ponteiros. O tamanho deste vetor será o número de linhas da matriz. Cada um destes ponteiros apontará para um vetor de **float** dinamicamente alocado.

O acesso a cada elemento da matriz é feito de forma semelhante à matriz alocada estaticamente. O primeiro índice entre colchetes retorna o vetor que contém a linha da matriz equivalente àquele índice. O segundo índice entre colchetes fornece o elemento contido na coluna.

Note que as linhas não se localizam necessariamente juntas na memória, podendo estar em qualquer lugar. O vetor de ponteiros indica onde cada linha se encontra na memória.

Para desalocar a matriz, é necessário desalocar primeiro cada linha para só então desalocar o vetor de ponteiros.

6.6.1 Passando matrizes alocadas dinamicamente como argumento de funções

No capítulo 5, vimos como passar uma matriz estaticamente alocada como argumento de uma função. Era necessário que a função conhecesse o número de colunas da matriz, para que soubesse onde acabava uma linha e começava outra. Este problema não existe com matrizes dinamicamente alocadas, podendo ser passada uma matriz de qualquer tamanho para a função.

Para que a função saiba o tamanho da matriz, é necessário que se passe como argumento o seu tamanho. O exemplo abaixo ilustra uma função que recebe três matrizes A, B e C. Esta função soma A com B e armazena o resultado em C.

```
void SomaMatrizes(int nLinhas, int nColunas, float** C,
                  const float** A, const float** B)
{
  int i, j;

  for(i = 0; i < nLinhas; i++)
  {
    for(j = 0; j < nColunas; j++)
    {
      C[i][j] = A[i][j] + B[i][j];
    }
  }
}
```

Para chamar esta função, é necessário que se tenha três matrizes de **float** alocadas dinamicamente. Estas matrizes devem ter o mesmo tamanho, já que não é possível somar matrizes de tamanhos

diferentes. Note que, desenvolvendo a função para matrizes alocadas dinamicamente, é possível utilizar matrizes de qualquer tamanho.

Note a utilização do modificador **const** nos argumentos A e B da função. Isto indica que são argumentos constantes, isto é, seus elementos não serão alterados dentro da função. A utilização do modificador **const**, neste caso, apesar de opcional, fornece segurança ao programador.

Sabe-se, antes de escrever a função, que ela não deve mexer nos argumentos A e B. Como matrizes são passadas por endereço, a princípio a função poderia alterar estes argumentos. Como não se deseja que a função altere os argumentos A e B, coloca-se o modificador const à frente dos argumentos constantes.

Caso tentemos modificar dentro da função algum elemento da matriz A ou da matriz B, o compilador acusará erro de compilação. Se não utilizássemos o modificador **const**, só perceberíamos o erro muito depois, com saídas incorretas do programa. O tempo de depuração de erros de compilação é muito menor que o de erros de execução.

6.7 Ponteiros para funções

Um ponteiro para uma função é um caso especial de tipo apontado. Se você definiu um ponteiro para função e inicializou-o para apontar para uma função particular ele terá o valor do endereço onde a função está localizada na memória.

O exemplo seguinte mostra este uso.

```
#include <stdio.h>

void ImprimeOla(void)
{
  printf("Ola");
}

void Imprime(const char* str)
{
  printf(str);
}

void main(void)
{
  void (*f1)(void);
  void (*f2)(const char*);

  f1 = ImprimeOla;
  f2 = Imprime;
  (*f1)();
  (*f2)(" mundo!\n");
}
```

6.7.1 A função qsort()

No capítulo 5 utilizamos o método da bolha para a ordenação de um vetor. Este método, apesar de bastante intuitivo, é lento quando comparado a outros métodos de ordenação. A biblioteca C contém uma função de ordenação que utiliza o algoritmo *quick sort*, consideravelmente mais rápido que o

método da bolha: a função *qsort*. Esta função está presente na biblioteca *stdlib.h*. Sua sintaxe é a seguinte.

```
void qsort(void* base, unsigned long nelem, unsigned long width,
           int(*fcmp)(const void*, const void*));
```

Esta função ordena *nelem* elementos de tamanho *width* localizados em *base*. O usuário deve fornecer uma função de comparação, *fcomp*, que compara dois elementos, *elem1* e *elem2*. Esta função deve retornar

```
< 0, se elem1 < elem2
= 0, se elem1 == elem2
> 0, se elem1 > elem2
```

Note que tanto *base* quanto os argumentos da função, *elem1* e *elem2* são do tipo **void***. Isto é necessário já que deseja-se que a função ordene qualquer tipo de variável.

Como exemplo, vamos criar um programa que ordena um vetor em ordem crescente e decrescente.

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

int OrdenaCrescente(const void* pa, const void* pb)
{
  int* px = (int*) pa;
  int* py = (int*) pb;

  return (*px) - (*py);
}

int OrdenaDecrescente(const void* pa, const void* pb)
{
  int* px = (int*) pa;
  int* py = (int*) pb;

  return (*py) - (*px);
}

void Imprime(int* piValor, int iTamanho)
{
  int i;

  for(i = 0; i < iTamanho; i++)
  {
    printf("%d ", piValor[i]);
  }
  printf("\n");
}

void main(void)
{
  int piValor[] = {1, 5, 3, 7, 4, 5, 9};

  qsort(piValor, 7, sizeof(int), OrdenaCrescente);
  Imprime(piValor, 7);
  qsort(piValor, 7, sizeof(int), OrdenaDecrescente);
  Imprime(piValor, 7);
}
```

Na função main(), o programa declara um vetor *piValor* e o inicializa com números inteiros desordenados.

Em seguida, o programa chama a função qsort() para ordená-lo em ordem crescente. Na chamada, é fornecido o vetor, *piValor* (lembrando: vetores são ponteiros), o número de elementos, 7, o tamanho de cada elemento, obtido através do operador **sizeof**, e a função de comparação, OrdenaCrescente(). Completada a ordenação, o programa imprime o vetor, através da função Imprime().

O programa chama novamente a função qsort() para ordená-lo em ordem decrescente. A única diferença é que agora a função de ordenação fornecida é a função OrdenaDecrescente().

As funções OrdenaCrescente() e OrdenaDecrescente() tem que ter a sintaxe exigida pela função qsort(), ou seja, recebem dois argumentos **void*** e retornam um **int** que indica qual dos dois argumentos vem na frente na ordenação. Como recebem argumentos **void***, é necessário converter para o tipo **int** antes de comparar.

A função OrdenaCrescente() retorna a diferença entre o primeiro e o segundo elementos. Isso faz com que o valor retorno satisfaça às condições impostas pela função qsort(). A função OrdenaDecrescente() retorna o contrário da função OrdenaCrescente(), de modo que a função qsort() interprete um valor maior como vindo à frente na ordenação.

A função qsort() pode ordenar um vetor de qualquer tipo de variável, desde que definidos os critérios de comparação entre dois elementos do vetor.

CAPÍTULO 7

DADOS ORGANIZADOS

7.1 Estruturas

Estrutura é um conjunto de variáveis, provavelmente de tipos diferentes, agrupadas e descritas por um único nome.

Por exemplo, numa folha de pagamento, desejamos guardar diversos registros (ou estruturas) representando funcionários. Cada funcionário, neste caso, possui alguns atributos, entre eles: nome (string), número do departamento (inteiro), salário (float), além de outros. Uma estrutura define um novo tipo composto, contendo diversos tipos básicos (ou compostos).

Para definir uma estrutura, definem-se os elementos dentro dela:

```
struct etiqueta
{
  declaração da variável membro
  declaração da variável membro
  ...
};
```

Po exemplo, a estrutura definida por:

```
struct Data
{
  int  dia;
  char mes[10];
  int  ano;
};
```

cria um novo tipo, chamado Data. Uma variável do tipo Data contém um dia (inteiro), um mês (string) e um ano (inteiro).

Para declarar variáveis do tipo Data, deve-se utilizar a palavra-chave **struct** antes do tipo. Por exemplo:

```
struct Data hoje;
```

declara uma variável de nome *hoje*, do tipo **struct** Data.

7.1.1 Acessando dados membro

Para acessar cada membro da estrutura separadamente, é utilizado o operador seleção (**.**). Por exemplo,

```
hoje.Ano = 2001;
hoje.Dia++;
strcpy(hoje.Mes, "Abril");
int iDia = hoje.Dia;
```

Cada membro da estrutura é equivalente a uma variável simples de seu tipo. O exemplo a seguir ilustra a utilização de estruturas.

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

void main(void)
{
  struct Data
  {
    int Dia;
    char Mes[10];
    int Ano;
  };

  struct Data hoje;

  hoje.Dia = 4;
  strcpy(hoje.Mes, "Outubro");
  hoje.Ano = 2001;
  printf("%d de %s de %d\n", hoje.Dia, hoje.Mes, hoje.Ano);
}
```

7.1.2 Estruturas dentro de estruturas

Estruturas podem conter outras estruturas em seu interior. Por exemplo, em uma agenda telefônica podemos guardar, além do telefone de cada pessoa, outros dados como o endereço e a data de nascimento.

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

void main(void)
{
  struct Data
  {
    int Dia;
    char Mes[10];
    int Ano;
  };

  struct Pessoa
  {
    char Nome[50];
    char Telefone[20];
    char Endereco[50];
    struct Data Nascimento;
  };

  struct Pessoa pessoa1;

  strcpy(pessoa1.Nome, "João");
  strcpy(pessoa1.Telefone, "2222-2222");
  strcpy(pessoa1.Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa1.Nascimento.Dia = 15;
  strcpy(pessoa1.Nascimento.Mes, "Janeiro");
  pessoa1.Nascimento.Ano = 1980;
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    pessoa1.Nome, pessoa1.Telefone, pessoa1.Endereco,
    pessoa1.Nascimento.Dia, pessoa1.Nascimento.Mes, pessoa1.Nascimento.Ano);
}
```

7.1.3 Atribuições entre estruturas

Na versão original do C é impossível atribuir o valor de uma variável estrutura a outra do mesmo tipo, usando uma simples expressão de atribuição. Seus dados membro tinham que ser atribuídos um a um.

Nas versões mais modernas de C, esta forma de atribuição já é possível. Isto é, se pessoa1 e pessoa2 são variáveis estrutura do mesmo tipo, a seguinte expressão pode ser usada:

```
pessoa1 = pessoa2;
```

7.1.4 Passando estruturas para funções

Versões mais recentes de C ANSI permitem que estruturas sejam passadas como argumento de funções. Como exemplo, vamos reescrever o exemplo chamando uma função que imprime os atributos de uma pessoa.

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

struct Data
{
  int Dia;
  char Mes[10];
  int Ano;
};

struct Pessoa
{
  char Nome[50];
  char Telefone[20];
  char Endereco[50];
  struct Data Nascimento;
};

void ImprimePessoa(struct Pessoa p)
{
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    p.Nome, p.Telefone, p.Endereco,
    p.Nascimento.Dia, p.Nascimento.Mes, p.Nascimento.Ano);
}

void main(void)
{
  struct Pessoa pessoa1;

  strcpy(pessoa1.Nome, "João");
  strcpy(pessoa1.Telefone, "2222-2222");
  strcpy(pessoa1.Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa1.Nascimento.Dia = 15;
  strcpy(pessoa1.Nascimento.Mes, "Janeiro");
  pessoa1.Nascimento.Ano = 1980;
  ImprimePessoa(pessoa1);
}
```

Note que, como mais de uma função do programa vai acessar as estruturas, elas são declaradas fora do escopo de qualquer função, o que as torna globais.

7.1.5 Vetores de estruturas

Assim como qualquer tipo básico, podemos ter vetores de estruturas. Estes vetores são definidos da mesma maneira que vetores de tipos básicos.

Vamos modificar o exemplo anterior para que trabalhemos com um vetor de pessoas.

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

struct Data
{
  int Dia;
  char Mes[10];
  int Ano;
};

struct Pessoa
{
  char Nome[50];
  char Telefone[20];
  char Endereco[50];
  struct Data Nascimento;
};

void ImprimePessoa(struct Pessoa p)
{
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    p.Nome, p.Telefone, p.Endereco,
    p.Nascimento.Dia, p.Nascimento.Mes, p.Nascimento.Ano);
}

void main(void)
{
  struct Pessoa pessoa[100];

  strcpy(pessoa[0].Nome, "João");
  strcpy(pessoa[0].Telefone, "2222-2222");
  strcpy(pessoa[0].Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa[0].Nascimento.Dia = 15;
  strcpy(pessoa[0].Nascimento.Mes, "Janeiro");
  pessoa[0].Nascimento.Ano = 1980;
  ImprimePessoa(pessoa[0]);
  strcpy(pessoa[1].Nome, "José");
  strcpy(pessoa[1].Telefone, "2222-2221");
  strcpy(pessoa[1].Endereco, "Av. Pres. Vargas, 10/1002");
  pessoa[1].Nascimento = pessoa[0].Nascimento;
  ImprimePessoa(pessoa[1]);
}
```

No exemplo acima, note que cada elemento do vetor é do tipo da estrutura Pessoa. O vetor é inicialmente dimensionado com 100 elementos, ou seja, podemos utilizar até 100 pessoas diferentes no vetor.

7.1.6 Ponteiros para estruturas

Ponteiros podem apontar para estruturas da mesma maneira que apontam para qualquer variável. Por exemplo, um ponteiro para a estrutura Data seria declarado como:

```
struct Data* pHoje = &hoje;
```

Para acessarmos os dados membro da estrutura através de seu ponteiro, podemos escrever

```
(*pHoje).Dia  /* Os parenteses sao necessarios */
```

ou podemos acessar os dados membro de um ponteiro através do operador (**->**)

```
pHoje->Dia
```

Vamos alterar a função ImprimePessoa, no exemplo anterior, para que receba o endereço da pessoa, ao invés de uma cópia. A vantagem de escrever a função desta maneira é o aumento da velocidade de processamento, já que é necessário copiar apenas o endereço, ao invés de todo o conteúdo da variável.

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

struct Data
{
  int Dia;
  char Mes[10];
  int Ano;
};

struct Pessoa
{
  char Nome[50];
  char Telefone[20];
  char Endereco[50];
  struct Data Nascimento;
};

void ImprimePessoa(struct Pessoa* p)
{
  printf("%s\n%s\n%s\n%d de %s de %d\n",
    p->Nome, p->Telefone, p->Endereco,
    p->Nascimento.Dia, p->Nascimento.Mes, p->Nascimento.Ano);
}

void main(void)
{
  struct Pessoa pessoa[100];

  strcpy(pessoa[0].Nome, "João");
  strcpy(pessoa[0].Telefone, "2222-2222");
  strcpy(pessoa[0].Endereco, "Av. Pres. Vargas, 10/1001");
  pessoa[0].Nascimento.Dia = 15;
  strcpy(pessoa[0].Nascimento.Mes, "Janeiro");
  pessoa[0].Nascimento.Ano = 1980;
  ImprimePessoa(&pessoa[0]);
  strcpy(pessoa[1].Nome, "José");
  strcpy(pessoa[1].Telefone, "2222-2221");
  strcpy(pessoa[1].Endereco, "Av. Pres. Vargas, 10/1002");
```

```
  pessoa[1].Nascimento = pessoa[0].Nascimento;
  ImprimePessoa(&pessoa[1]);
}
```

É recomendável que estruturas muito grandes sejam sempre passadas por endereço. No caso, a estrutura Pessoa ocupa 134 bytes (50 do nome, 20 do telefone, 50 do endereço e mais 14 da estrutura Data - 2 do dia, 10 do mês e mais 2 do ano). Passando o argumento por valor, o programa será obrigado a copiar 134 bytes, ao passo que passando por endereço, apenas os 4 bytes do ponteiro são copiados.

7.2 Uniões

Uma união permite que as mesmas localizações de memória sejam referenciadas de mais de um modo. Sua sintaxe é semelhante à da estrutura.

```
union etiqueta
{
  declaração da variável membro
  declaração da variável membro
  ...
};
```

A diferença é que a estrutura armazena todos os seus dados membro individualmente, enquanto a união utiliza o mesmo espaço de memória para armazenar todos os seus dados membro. Enquanto o espaço ocupado por uma estrutura é a soma do espaço utilizado por seus dados membro, a memória utilizada pela união é a memória ocupada por seu maior dado membro.

Po exemplo, a união definida por:

```
union Mes
{
  int  numero;
  char nome[10];
};
```

ocupa 10 bytes (relativos ao dado *nome*). Podemos acessar seus dados da mesma forma que na estrutura mas, uma vez alterado um dado membro, os outros são alterados também.

Ao trabalhar com uniões, deve-se acessar apenas um dos dados membro, dependendo do tipo de uso. Por exemplo, se definirmos

```
Mes mes;
strcpy(mes.nome, "Janeiro");
```

e tentarmos obter

```
mes.numero
```

o valor retornado é m valor sem sentido, já que o espaço de memória ocupado foi alterado por outra variável.

Não tendo problemas de falta de memória, deve-se optar pela utilização de estruturas ao invés de uniões.

7.3 Enumeração

A linguagem C apresenta um tipo de dado adicional, chamado enumeração. Sua sintaxe é:

```
enum identificacao {enum1, enum2 ...};
```

Por exemplo:

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

especifica que *dias* é uma identificação para uma variável do tipo **enum**, que somente pode ter os valores de segunda a domingo.

Podemos declarar variáveis do tipo do enumerado acima, por exemplo:

```
enum dias dia;
```

os tipos de enumeração são representados internamente por inteiros. O primeiro identificador recebe o valor 0, o próximo 1 e assim sucessivamente. Ou seja, na enumeração *dias*, *segunda* vale 0, *terca* vale 1 e assim sucessivamente, até *domingo*, que vale 6.

Um enumerador pode ser declarado alterando os valores correspondentes a cada termo. Por exemplo, se declararmos

```
enum dias {segunda = 2, terca, quarta, quinta, sexta};
```

nesta enumeração, *segunda* vale 2, *terca* vale 3 e assim sucessivamente, até *sexta*, que vale 6. Se declararmos

```
enum dias {segunda = 2, quarta = 4, quinta = 5};
```

definimos separadamente os termos.

CAPÍTULO 8

ENTRADA E SAÍDA

Neste capítulo, serão apresentados as operações de C para leitura e gravação em disco. Operações em disco são executadas em arquivos. A biblioteca C oferece um pacote de funções para acessar arquivos de quatro maneiras diferentes:

- Os dados são lidos e escritos um caractere por vez. Oferece as funções getc() e putc().
- Os dados são lidos e escritos como strings. Oferece as funções fgets() e fputs().
- Os dados são lidos e escritos de modo formatado. Oferece as funções fscanf() e fprintf().
- Os dados são lidos e escritos num formato chamado registro ou bloco. É usado para armazenar seqüências de dados como vetores e estruturas. Oferece as funções fread() e fwrite().

Outra maneira de classificar operações de acesso a arquivos é conforme a forma como eles são abertos: em modo texto ou em modo binário. Estes conceitos serão apresentados nas seções a seguir.

8.1 Arquivos Texto

Um arquivo aberto em modo texto é interpretado em C como seqüências de caracteres agrupadas em linhas. As linhas são separadas por um único caractere chamado caractere de nova linha ou LF (*line feed*), de código ASCII 10 decimal. É equivalente ao caractere `'\r'`.

Alguns sistemas operacionais, como o DOS, representam a mudança de linha por dois caracteres: O caractere de retorno de carro ou CR (*carriage return*), de código ASCII 13 decimal (caractere `'\n'`) e o caractere LF. Neste caso, o compilador C converte o par CR/LF em um único caractere de nova linha quando um arquivo em modo texto é lido e converte o caractere de nova linha no par CR/LF quando o arquivo é gravado.

O código em C é independente do sistema operacional que estamos utilizando, tratando a mudança de linha sempre da mesma forma.

O exemplo abaixo lê caracteres de um arquivo texto e armazena em outro arquivo texto. Para executar este exemplo, crie um arquivo texto de nome entrada.txt, contendo algumas frases. Este arquivo pode ser criado em qualquer editor de texto não formatado (evite usar o Word ou o WordPad, por serem de texto formatado. Utilize, por exemplo, o próprio compilador ou o Bloco de Notas). Por exemplo, o arquivo entrada.txt pode ter o seguinte conteúdo:

```
Linguagem C
Programa teste
```

O código do programa é fornecido a seguir.

```
#include <stdio.h>

void main(void)
{
  char ch;
  FILE *in, *out;
```

```
  if((in = fopen("entrada.txt", "rt")) == NULL)
  {
    printf("Impossivel abrir arquivo entrada.txt.");
    return;
  }
  if((out = fopen("saida.txt", "wt")) == NULL)
  {
    printf("Impossivel abrir arquivo saida.txt.");
    return;
  }
  while((ch = getc(in)) != EOF)
    putc(ch, out);
  fclose(in);
  fclose(out);
}
```

Este programa aguarda a entrada de uma linha de texto e termina quando a tecla <ENTER> for pressionada. A linha é gravada no arquivo **saida.txt**.

A estrutura FILE está presente na biblioteca stdio.h e armazena informações sobre o arquivo. Esta estrutura não será discutida neste curso.

8.1.1 As funções fopen() e fclose()

Quando abrimos um arquivo, a informação que recebemos (se o arquivo for aberto) é um ponteiro para a estrutura FILE. Cada arquivo que abrimos terá uma estrutura FILE com um ponteiro para ela.

A função fopen() tem a seguinte sintaxe:

```
FILE* fopen(const char* filename, const char* mode);
```

Esta função recebe como argumentos o nome do arquivo a ser aberto (*filename*) e o modo de abertura (*mode*). Retorna um ponteiro para a estrutura FILE, que armazena informações sobre o arquivo aberto.

O nome do arquivo pode ser fornecido com ou sem o diretório onde ele está localizado. Caso se deseje fornecer o caminho completo do arquivo, lembre-se que a contrabarra em C não é um caractere, e sim um meio de fornecer caracteres especiais. O caractere contrabarra é representado por `'\\'`. Ou seja, em DOS o caminho completo de um arquivo pode ser, por exemplo, `"C:\\temp\\saida.txt"`.

A lista de modos de abertura de arquivos é apresentada a seguir.

"r"	Abrir um arquivo para leitura (*read*). O arquivo deve estar presente no disco.
"w"	Abrir um arquivo para gravação (*write*). Se o arquivo estiver presente, ele será destruído e reinicializado. Se não existir, ele será criado.
"a"	Abrir um arquivo para gravação em anexo (*append*). Os dados serão adicionados ao fim do arquivo existente, ou um novo arquivo será criado.

Além destes modos podem ser adicionados os seguintes modificadores.

"+"	A adição deste símbolo permite acesso de leitura e escrita.
"b"	Abrir um arquivo em modo binário.
"t"	Abrir um arquivo em modo texto.

No exemplo é aberto o arquivo saida.txt, em modo texto, para gravação.

Caso o arquivo possa ser aberto, a função retorna um ponteiro para a estrutura FILE contendo as informações sobre o arquivo. Caso contrário, retorna NULL. No exemplo, é verificado se o arquivo foi aberto com êxito. Caso negativo, o programa apresenta uma mensagem de erro e termina a execução.

O erro na abertura de arquivo pode ser causado por diversos fatores: espaço insuficiente em disco (no caso de gravação), arquivo inexistente (no caso de leitura) etc.

Ao ser aberto um arquivo é necessário que ele seja fechado após utilizado. Isto é feito por meio da função fclose(). Sua sintaxe é a seguinte:

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

Para fechar o arquivo, basta chamar a função fclose() e passar como argumento o ponteiro para a estrutura FILE que contém as informações do arquivo que se deseja fechar. Em caso de êxito, a função retorna 0. Caso contrário, retorna EOF (*end-of-file*). EOF é definido na biblioteca stdio.h através da diretiva **#define** e indica fim de arquivo.

8.1.2 As funções getc () e putc()

A função getc() lê um caractere por vez do arquivo. Sua sintaxe é

```
int getc(FILE* f);
```

A função recebe como argumento um ponteiro para FILE e retorna o caractere lido. Em caso de erro, retorna EOF.

A função putc() é o complemento de getc(). Ela escreve um caractere no arquivo. Sua sintaxe é

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

A função recebe como argumentos um caractere e um ponteiro para FILE. O valor retornado é o próprio caractere fornecido. Em caso de erro, a função retorna EOF.

8.1.3 As funções fgets () e fputs()

O exemplo a seguir faz a mesma coisa que o exemplo anterior. A única diferença é que ao invés de acessar os arquivos texto caractere a caractere, eles serão acessados linha a linha.

```
#include <stdio.h>

void main(void)
{
  char ch[1001];
  FILE *in, *out;

  if((in = fopen("entrada.txt", "rt")) == NULL)
  {
    printf("Impossivel abrir arquivo entrada.txt.");
    return;
  }
  if((out = fopen("saida.txt", "wt")) == NULL)
  {
    printf("Impossivel abrir arquivo saida.txt.");
    return;
  }
  while((fgets(ch, 1000, in)) != NULL)
    fputs(ch, out);
```

```
  fclose(in);
  fclose(out);
}
```

A função fgets() tem a seguinte sintaxe.

```
char* gets(char* s, int n, FILE* f);
```

Esta função lê caracteres do arquivo é os coloca na string *s*. A função pára de ler quando lê *n-1* caracteres ou o caractere LF ('\r'), o que vier primeiro. O caractere LF é incluído na string. O caractere nulo é anexado ao final da string para marcar seu final. Em caso de êxito, a função retorna *s*. Em caso de erro ou fim do arquivo, retorna NULL.

A função fputs() é o complemento de fgets(). Ela escreve uma seqüência de caracteres no arquivo. Sua sintaxe é

```
int putc(char* s, FILE* f);
```

A função recebe como argumentos uma string e um ponteiro para FILE. O valor retornado é o último caractere fornecido. Em caso de erro, a função retorna EOF.

Note que a função copia a string fornecida tal como ela é para o arquivo. Se não existir na string, não é inserido o caractere de nova linha (LF) nem o caractere nulo.

8.1.4 As funções fprintf () e fscanf()

Para leitura e gravação de arquivos com formatação, são utilizadas as funções fprintf() e fscanf(). Estas funções são semelhantes a printf() e scanf(), utilizadas ao longo do curso.

O programa a seguir lê dados de um arquivo, executa um processamento com os dados lidos e grava os resultados em outro arquivo, em forma de um relatório de saída.

```
#include <stdio.h>

void main(void)
{
  float distancia, tempo, velocidade;
  FILE *in, *out;

  if((in = fopen("entrada.txt", "rt")) == NULL)
  {
    printf("Impossivel abrir arquivo entrada.txt.");
    return;
  }
  if((out = fopen("saida.txt", "wt")) == NULL)
  {
    printf("Impossivel abrir arquivo saida.txt.");
    return;
  }
  fscanf(in, "%f%f", &distancia, &tempo);
  velocidade = distancia / tempo;
  fprintf(out, "Distancia percorrida: %10.2f km\n", distancia);
  fprintf(out, "Tempo decorrido:      %10.2f h\n", tempo);
  fprintf(out, "Velocidade media:     %10.2f km/h\n", velocidade);
  fclose(in);
  fclose(out);
}
```

Este programa lê do arquivo entrada.txt a distância percorrida e o tempo decorrido. É feito o cálculo da velocidade média e a saída é armazenada num arquivo texto, formatado. As sintaxes de fprintf() e fscanf() são semelhantes às sintaxes de printf() e scanf() exceto pela inclusão do primeiro argumento, que é um ponteiro para a estrutura FILE.

As strings de formatação utilizadas nestas funções são as mesmas utilizadas nas funções printf() e scanf().

8.2 Arquivos Binários

Os dados são armazenados em arquivos binários da mesma forma que são armazenados na memória do computador. Ou seja, ao guardarmos o valor de uma variável **float** num arquivo binário, ela ocupa no arquivo os mesmos 4 bytes que ocupa na memória.

Arquivos guardados em modo binário não são facilmente entendidos por uma pessoa que o esteja lendo. Em compensação, para um programa é muito fácil interpretar este arquivo.

Ao guardarmos valores em modo texto, o compilador converte o valor da variável para uma forma de apresentá-lo através de caracteres. Por exemplo, o número inteiro 20000, apesar de ocupar somente dois bytes na memória, para ser armazenado em um arquivo texto ocupa pelo menos cinco bytes, já que cada caractere ocupa um byte.

Arquivos binários normalmente são menores que arquivos texto, contendo a mesma informação. A velocidade de leitura do arquivo também é consideravelmente reduzida ao utilizarmos arquivos binários.

Em compensação, a visualização do arquivo não traz nenhuma informação ao usuário, pelo fato de serem visualizados os caracteres correspondentes aos bytes armazenados na memória. No caso de variáveis **char**, a visualização é idêntica em arquivos texto e binários.

8.2.1 As funções fread() e fwrite()

O programa abaixo grava um arquivo binário contendo 2 vetores de 20 elemento cada um: o primeiro vetor é de caracteres e o segundo é de variáveis inteiras.

```
#include <stdio.h>

void main(void)
{
  char ch[20] = "Teste geral";
  int valor[20] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  FILE *out;

  out = fopen("binario.bin", "wb");
  if(out == NULL)
  {
    printf("Impossivel abrir arquivo binario.bin.");
    return;
  }
  fwrite(ch, sizeof(char), 20, out);
  fwrite(valor, sizeof(int), 20, out);
  fclose(out);
}
```

Ao executar este programa, ele gera um arquivo binário, `binario.bin`. Se você tentar visualizar este arquivo num editor como o Bloco de Notas, do Windows, conseguirá distinguir os 20 primeiros bytes, que é o vetor de caracteres. A partir daí, os bytes utilizados pelo compilador para armazenar os valores inteiros não fazem mais sentido quando visualizados.

O programa abaixo lê o arquivo `binario.bin` e apresenta os valores lidos na tela.

```
#include <stdio.h>

void main(void)
{
  char ch[20];
  int valor[20];
  FILE *in;

  in = fopen("binario.bin", "rb");
  if(in == NULL)
  {
    printf("Impossivel abrir arquivo binario.bin.");
    return;
  }
  fread(ch, sizeof(char), 20, in);
  fread(valor, sizeof(int), 20, in);
  for(i = 0; i < 20; i++)
    printf("%c", ch[i]);
  for(i = 0; i < 20; i++)
    printf("\n%d", valor[i]);
  fclose(in);
}
```

ANEXO A

TABELA ASCII

As tabelas mostradas neste apêndice representam os 256 códigos usados nos computadores da família IBM. Esta tabela refere-se ao *American Standard Code for Information Interchange* (código padrão americano para troca de informações), que é um conjunto de números representando caracteres ou instruções de controle usados para troca de informações entre computadores entre si, entre periféricos (teclado, monitor, impressora) e outros dispositivos. Estes códigos tem tamanho de 1 byte com valores de `00h` a `FFh` (0 a 255 decimal). Podemos dividir estes códigos em três conjuntos: controle, padrão e estendido.

Os primeiros 32 códigos de `00h` até `1Fh` (0 a 31 decimal), formam o **conjunto de controle** ASCII. Estes códigos são usados para controlar dispositivos, por exemplo uma impressora ou o monitor de vídeo. O código `0Ch` (*form feed*) recebido por ima impressora gera um avanço de uma página. O código `0Dh` (*carriage return*) é enviado pelo teclado quando a tecla ENTER é pressionada. Embora exista um padrão, alguns poucos dispositivos tratam diferentemente estes códigos e é necessário consultar o manual para saber exatamente como o equipamento lida com o código. Em alguns casos o código também pode representar um caracter imprimível. Por exemplo o código 01h representa o caracter ☺ (*happy face*).

Os 96 códigos seguintes de `20h` a `7Fh` (32 a 127 decimal) formam o **conjunto padrão** ASCII. Todos os computadores lidam da mesma forma com estes códigos. Eles representam os caracteres usados na manipulação de textos: códigos-fonte, documentos, mensagens de correio eletrônico, etc. São constituídos das letras do alfabeto latino (minúsculo e maiúsculo) e alguns símbolos usuais.

Os restantes 128 códigos de `80h` até `FFh` (128 a 255 decimal) formam o **conjunto estendido** ASCII. Estes códigos também representam caracteres imprimíveis porem cada fabricante decide como e quais símbolos usar. Nesta parte do código estão definidas os caracteres especiais: é, ç, ã, ü ...

Tabela A.1 - Conjunto de controle ASCII

Dec.	Hex.	Controle
0	00h	NUL (*Null*)
1	01h	SOH (*Start of Heading*)
2	02h	STX (*Start of Text*)
3	03h	ETX (*End of Text*)
4	04h	EOT (*End of Transmision*)
5	05h	ENQ (*Enquiry*)
6	06h	ACK (*Acknowledge*)
7	07h	BEL (*Bell*)
8	08h	BS (*Backspace*)
9	09h	HT (*Horizontal Tab*)
10	0Ah	LF (*Line Feed*)
11	0Bh	VT (*Vertical Tab*)
12	0Ch	FF (*Form Feed*)
13	0Dh	CR (*Carriage Return*)
14	0Eh	SO (*Shift Out*)
15	0Fh	SI (*Shift In*)
16	10h	DLE (*Data Link Escape*)
17	11h	DC1 (*Device control 1*)
18	12h	DC2 (*Device control 2*)
19	13h	DC3 (*Device control 3*)
20	14h	DC4 (*Device control 4*)
21	15h	NAK (*Negative Acknowledge*)
22	16h	SYN (*Synchronous Idle*)
23	17h	ETB (*End Transmission Block*)
24	18h	CAN (*Cancel*)
25	19h	EM (*End of Media*)
26	1Ah	SUB (*Substitute*)
27	1Bh	ESC (*Escape*)
28	1Ch	FS (*File Separator*)
29	1Dh	GS (*Group Separator*)
30	1Eh	RS (*Record Separator*)
31	1Fh	US (*Unit Separator*)

Dec.	Hex.	Caractere
0	00h	<espaço>
1	01h	!
2	02h	"
3	03h	#
4	04h	$
5	05h	%
6	06h	&
7	07h	'
8	08h	(
9	09h	)
10	0Ah	*
11	0Bh	+
12	0Ch	,
13	0Dh	-
14	0Eh	.
15	0Fh	/
16	10h	0
17	11h	1
18	12h	2
19	13h	3
20	14h	4
21	15h	5
22	16h	6
23	17h	7
24	18h	8
25	19h	9
26	1Ah	:
27	1Bh	;
28	1Ch	<
29	1Dh	=
30	1Eh	>
31	1Fh	?

Tabela A.1 - Conjunto de controle ASCII (continuação)

Dec.	Hex.	Caractere
64	40h	@
65	41h	A
66	42h	B
67	43h	C
68	44h	D
69	45h	E
70	46h	F
71	47h	G
72	48h	H
73	49h	I
74	4Ah	J
75	4Bh	K
76	4Ch	L
77	4Dh	M
78	4Eh	N
79	4Fh	O
80	50h	P
81	51h	Q
82	52h	R
83	53h	S
84	54h	T
85	55h	U
86	56h	V
87	57h	W
88	58h	X
89	59h	Y
90	5Ah	Z
91	5Bh	[
92	5Ch	\
93	5Dh	]
94	5Eh	^
95	5Fh	_

Dec.	Hex.	Caractere
96	60h	`
97	61h	a
98	62h	b
99	63h	c
100	64h	d
101	65h	e
102	66h	f
103	67h	g
104	68h	h
105	69h	i
106	6Ah	j
107	6Bh	k
108	6Ch	l
109	6Dh	m
110	6Eh	n
111	6Fh	o
112	70h	p
113	71h	q
114	72h	r
115	73h	s
116	74h	t
117	75h	u
118	76h	v
119	77h	w
120	78h	x
121	79h	y
122	7Ah	z
123	7Bh	{
124	7Ch	|
125	7Dh	}
126	7Eh	~
127	7Fh	<delete>

Tabela A.1 - Conjunto de controle ASCII (continuação)

Dec.	Hex.	Caractere
128	80h	Ç
129	81h	ü
130	82h	é
131	83h	â
132	84h	ä
133	85h	à
134	86h	å
135	87h	ç
136	88h	ê
137	89h	ë
138	8Ah	è
139	8Bh	ï
140	8Ch	î
141	8Dh	ì
142	8Eh	Ä
143	8Fh	Å
144	90h	É
145	91h	æ
146	92h	Æ
147	93h	ô
148	94h	ö
149	95h	ò
150	96h	û
151	97h	ù
152	98h	ÿ
153	99h	Ö
154	9Ah	Ü
155	9Bh	¢
156	9Ch	£
157	9Dh	¥
158	9Eh	
159	9Fh	ƒ

Dec.	Hex.	Caractere
160	A0h	áááá
161	A1h	í
162	A2h	ó
163	A3h	ú
164	A4h	ñ
165	A5h	Ñ
166	A6h	ª
167	A7h	º
168	A8h	¿
169	A9h	
170	AAh	¬
171	ABh	½
172	ACh	¼
173	ADh	¡
174	AEh	«
175	AFh	»
176	B0h	
177	B1h	
178	B2h	
179	B3h	
180	B4h	
181	B5h	
182	B6h	
183	B7h	
184	B8h	
185	B9h	
186	BAh	
187	BBh	
188	BCh	
189	BDh	
190	BEh	
191	BFh	

Tabela A.1 - Conjunto de controle ASCII (continuação)

Dec.	Hex.	Caractere
192	C0h	
193	C1h	
194	C2h	
195	C3h	
196	C4h	
197	C5h	
198	C6h	
199	C7h	
200	C8h	
201	C9h	
202	CAh	
203	CBh	
204	CCh	
205	CDh	
206	CEh	
207	CFh	
208	DOh	
209	D1h	
210	D2h	
211	D3h	
212	D4h	
213	D5h	
214	D6h	
215	D7h	
216	D8h	
217	D9h	
218	DAh	
219	DBh	
220	DCh	
221	DDh	
222	DEh	
223	DFh	

Dec.	Hex.	Caractere
224	E0h	
225	E1h	ß
226	E2h	
227	E3h	
228	E4h	
229	E5h	
230	E6h	µ
231	E7h	
232	E8h	
233	E9h	
234	EAh	
235	EBh	
236	ECh	
237	EDh	
238	EEh	∈
239	EFh	
240	F0h	
241	F1h	±
242	F2h	
243	F3h	
244	F4h	
245	F5h	
246	F6h	÷
247	F7h	
248	F8h	°
249	F9h	.
250	FAh	.
251	FBh	
252	FCh	
253	FDh	²
254	FEh	.
255	FFh	

ANEXO B

EXERCÍCIOS

B.1 Introdução

1. Fazer um programa que imprima na tela o nome de todos os tipos de dados utilizados pelo C e seus respectivos tamanhos ocupados em memória.
2. Quais dos seguintes nomes são válidos para variáveis em C?
   a) 5ij
   b) _abc
   c) a_b_c
   d) 00TEMPO
   e) int
   f) A123
   g) a123
   h) x**x
   i) __A
   j) a-b-c
   k) OOTEMPO
   l) \abc
   m) *abc

B.2 Operadores

1. Que valor têm as seguintes expressões:
   a) 7 / 2
   b) 7 % 2
   c) 7.0 / 2
   d) 7 / 2.0
   e) 7.0 / 2.0
2. Fazer um programa para transformar graus Farenheit em Celsius. A fórmula para conversão é a seguinte:

$$\boxed{\frac{C}{5} = \frac{F - 32}{9}}$$

   onde:
   C – Temperatura em graus Celsius
   F – Temperatura em graus Farenheit
3. Fazer um programa para transformar graus Celsius em Farenheit. A fórmula de conversão é fornecida no exercício anterior.

4. Implementar um programa que calcule a média aritmética de três números reais.

B.3 Controle de Fluxo

1. Fazer um programa que leia dois números inteiros e apresente o maior deles.
2. Fazer um programa que leia três números inteiros e apresente o maior deles.
3. Fazer um programa que mostre ao usuário quatro opções de operação:
   a) adição
   b) subtração
   c) multiplicação
   d) divisão

   Após a seleção da opção desejada, leia dois números e realize a operação, exibindo a resultado na tela.
4. O que será impresso pelo seguinte programa ?

```
main()
{
 int x = 1, y = 1;

  if (y<0)
    if (y>0)
      x = 3;
    else
      x = 5;
  printf("x= %d\n", x);
}
```

5. Supondo que a população de um país, tomado como comparação, seja de 200 milhões de habitantes em 2000 e que sua taxa de crescimento seja de 1,3% ao ano. Fazer um programa para calcular o ano em que um outro país, cuja população e taxa de crescimento sejam fornecidas pelo usuário, iguale ou ultrapasse a população do país base.
6. Implementar um programa que dados:
   a) saldo em conta corrente (negativo),
   b) limite de cheque especial,
   c) taxa de juros cobrada,

   calcule quantos meses a conta poderá ficar sem receber depósito sem exceder o limite
7. Fazer um programa que realize a operação de exponenciação, sendo que
   a) O usuário deverá digitar a base e o expoente;
   b) A base deverá ser um número real e positivo;
   c) O expoente deverá ser um número inteiro.
8. Fazer um programa para calcular o fatorial de um número, sendo que
   $N! = N*(N-1)!$
   $0! = 1$.
9. Fazer um programa para calcular o N-ésimo termo da seqüência de Fibonacci. Sendo que
   $F_n = F_{n-1} + F_{n-2}$
   $F_0 = 0$
   $F_1 = 1$

Logo, temos que a seqüência ficará: 0,1,1,2,3,5,8, ...

10. Implementar um programa que calcule a quantidade de divisores de um número
11. Implementar um programa que escreva um número de 1 a 99, por extenso.
12. Implementar um programa que leia o dia e mês de nascimento de uma pessoa e imprima o seu signo no horóscopo.

Aries	21/03 até 20/04
Touro	21/04 até 20/05
Gêmeos	21/05 até 20/06
Câncer	21/06 até 21/07
Leão	22/07 até 22/08
Virgem	23/08 até 22/09
Libra	23/09 até 22/10
Escorpião	23/10 até 21/11
Sagitário	22/11 até 21/12
Capricórnio	22/12 até 20/01
Aquário	21/01 até 19/02
Peixes	20/02 até 20/03

13. Implemente um programa que transforme números arábicos em romanos, até 999.
    Exemplo:
    1 I
    5 V
    10 X
    50 L
    100 C
    500 D

B.4 Funções

1. Criar uma função para a exponenciação, considerado o expoente inteiro.
2. Criar uma função que calcule o fatorial de um número inteiro
3. Criar uma função recursiva que calcule o fatorial de um número inteiro, sendo que:
   N! = N*(N-1)!
   0! = 1.
4. Fazer uma função para calcular o N-ésimo termo da seqüência de Fibonacci. Sendo que
   $F_n = F_{n-1} + F_{n-2}$
   $F_0 = 0$
   $F_1 = 1$
   Logo, temos que a seqüência ficará: 0,1,1,2,3,5,8, ...
5. Fazer uma função recursiva para calcular o N-ésimo termo da seqüência de Fibonacci.
6. Fazer um programa que calcule os números palíndromos de 0 a 5000. O programa deverá solicitar do usuário uma das seguintes alternativas:
   a) Calcular números palíndromos
   b) Terminar

OBS. Número Palíndromo é aquele que tem igual valor se lido da esquerda para direita ou vice-versa. Ex. 0,1, ... 9, 11, 22, ..., 99, 101, 111, 121, ...

7. Implementar um programa que calcule o MDC de dois números através do seguinte algorítmo:
Função MDC(M,N)
```
Início
  Se (M < N) então
    Resp := MDC(N,M)
  Fim-se
  Se (N = 0) então
    Resp := M;
  Senão
    Resp := MDC(N,Resto(M,N));
  Fim-se
  Retorne Resp;
Fim
```

8. Faca a estrutura de um programa principal que calcule a potência de um número fazendo chamada a seguinte função:

```
potencia(int x, int n)
{
  int p;
  for (p=1;n>0;--n)
    p *= x;
  return(p);
}
```

OBS. Caso exista algum erro na função potência, corrija-o.

9. Considere a seguinte função que calcula a área de um triângulo de lados a, b e c:

```
#include <math.h>   /* arquivo onde eestá definida a funcao sqrt() */

AreaTri( )
{
  x = (a+b+c)/2.0;
  area = x*(x-a)*(x-b)*(x-c);
  area = sqrt(area);
}
```

Faca um programa que calcule a área do triângulo de lados a=3, b=4 e c=5, utilizando a funcao acima.

B.5 Vetores e Matrizes

1. Fazer um programa que leia dez números inteiros e apresente o maior deles.
2. Fazer um programa para ler um nome (máximo de 50 caracteres) e abreviar os nomes do meio.
Exemplo: Joaquim José da Silva Xavier, ficaria: Joaquim J. d. S. Xavier
3. Fazer um programa que faça a reserva de lugares em um teatro sendo que:
a) o teatro tem 10 fileiras de cadeiras (A,B, .. J), cada uma com 50 cadeiras.
b) os lugares são identificados com uma letra (coluna) e um número (fila)
Exemplo: A-04, B-23 etc.
c) O programa deverá solicitar do usuário qual lugar ele deseja ocupar.

d) Caso o lugar indicado não esteja vago, o programa deverá avisar ao usuário para que escolha um novo lugar.
e) Caso o lugar indicado esteja vago, este deverá ser reservado ao usuário.
f) Sempre que uma dada fileira (1a., 2a. 3a. etc.) estiver totalmente ocupada, o programa deverá informar ao usuário antes que ele efetue a escolha.
g) Ao final de cada reserva o programa deverá indicar o total de lugares ocupados e o total de lugares vagos.

4. Analise o código abaixo e responda:
a) para que serve a função dada ?
b) de que maneira deveríamos ter a função main() para chamar corretamente esta função ?
c) indique os possíveis erros de compilação e corrija-os;

```
int  avg(float a[], int size)
{
  int i;
  float sum;

  sum = 0;
  for (i=0;i < size; i++)
    sum += a[i];
  return(sum / size);
}
```

5. Analise o código abaixo e responda:
a) para que serve a função dada ?
b) de que maneira deveríamos ter a função main() para chamar corretamente esta função ?

```
strpos(char s1[],char s2[])
{
  int len1, len2;
  int i, j1, j2;
  len1 = strlen(s1);
  len2 = strlen(s2);
  for (i=0; i+len2 <= len1; i++)
    for (j1 = i, j2 = 0; j2 <= len2 && s1[j1] == s2[j2]; j1++, j2+++
  if (j2 == len2)
    return(i);
  return(-1);
}
```

6. Analise o código abaixo e responda:
a) para que serve a função dada ?
b) de que maneira deveríamos ter a função main() para chamar corretamente esta função ?

```
strcat(char s1[], char s2[])
{
  int i, j;
  for (i=0; s1[i] != '\0'; i++)
    ;
  for (j=0; s2[j] != '\0'; s1[i++] = s2[j++])
    ;
}
```

7. Analise o código abaixo e responda:
   a) para que serve a função dada ?
   b) de que maneira deveríamos ter a função main() para chamar corretamente esta função ?

```
substr(char s1[], int i, int j, char s2[])
{
  int k, m;
  for (k = i, m = 0; m < j; s2[m++] = s1[k++])
    ;
  s2[m] = '\0';
}
```

8. A MODA de um vetor de números é o número m no vetor que é repetido com maior freqüência. Se mais de um número for repetido com freqüência máxima igual, não existirá uma moda. Escreva uma função em C que aceite um vetor de números e retorne a moda ou uma indicação de que a moda não existe.
9. A mediana de um vetor de números é o elemento m do vetor, tal que a metade dos números restantes no vetor é maior ou igual a m e a outra metade é menor ou igual a m, se o número de elementos no vetor for ímpar. Se o número de elementos for par, a mediana será a média dos dois elementos, m1 e m2, tal que metade dos elementos restantes é maior ou igual a m1 e m2, e metade dos elementos é menor ou igual a m1 e m2. Escreva uma função em C que aceite um vetor de números e retorne a mediana dos números do vetor.
10. Faca um programa que lei um vetor de 20 posições de inteiros e imprima o maior e o segundo maior valor do vetor.

B.6 Ponteiros

1. Quais das seguintes instruções são corretas para declarar um ponteiro?
   a) int_ptr x;
   b) int *x;
   c) *int x;
   d) *x;
2. Qual é a maneira correta de referenciar *ch*, assumindo que o endereço de *ch* foi atribuído ao ponteiro *pch*?
   a) *pch
   b) &pch
   c) pch
3. Na expressão float* pf, o que é do tipo float?
   a) a variável pf
   b) o endereço de pf
   c) a variável apontada por pf
   d) nenhuma das anteriores
4. Assumindo que o endereço da variável var foi atribuído a um ponteiro pvar, escreva uma expressão que divida var por 10 sem utilizar a variável var.

B.7 Dados Organizados

5. Considere as seguintes declarações:

```
struct ss
{
 char a[10]; int b;
} v;

union uu
{
 char a[10]; int b;
} v;
```

Em cada declaração, qual é o tamanho (em bytes) do bloco de memória que será alocado para armazenar os valores das variáveis "v"?

B.8 Entrada e Saída

1. Faça um programa em C que receba como argumentos o nome de dois arquivos e a seguir efetue a cópia do primeiro arquivo no segundo arquivo.
2. Faça um programa que leia o mês e ano e imprima a folha do calendário correspondente ao mês. Exemplo: Para mes=9 e ano=1995 teremos a saída:

```
SETEMBRO DE 1995
DOM SEG TER QUA QUI SEX SAB
                        1   2
  3   4   5   6   7   8   9
 10  11  12  13  14  15  16
 17  18  19  20  21  22  23
 24  25  26  27  28  29  30
```

O programa deve levar em conta que fevereiro tem 29 dias se: (resto(ano/4)=0 e resto(ano/100)<>0) ou (resto(ano/400)=0). Para determinar em que dia da semana cai o primeiro dia do mês e ano, utilize o seguinte algoritmo:

```
E1.: se mes>2, então vá para E3
E2.: mes = mes +10, ano = ano-1, vá para E4
E3.: mes = mes - 2
E4.: aux1 = ano / 100, aux2 = resto(ano/100)
E5.: aux3 = 106 + (13*mes-1)/5 + aux2/4 + aux1/4
E6.: dia_semana = resto((aux3+aux2-2aux1+1)/7)
```

Desta forma, se:
dia_semana = 0, então o dia primeiro cai num domingo,
dia_semana = 1, então o dia primeiro cai numa segunda-feira, e assim por diante.
Obs: As divisões nas expressões acima são divisões inteiras.

ANEXO C

EXEMPLOS DE FUNÇÕES

C.1 Função de ordenação - método da bolha

```
void BolhaOrdenaInt(int iTamanho, int* piDados)
{
    int iFlag;
    int iAux;
    int i;

    do
    {
        iFlag = 0;
        for(i = 0; i < iTamanho - 1; i++)
        {
            if(piDados[i] > piDados[i + 1])
            {
                iAux = piDados[i];
                piDados[i] = piDados[i + 1];
                piDados[i + 1] = iAux;
                iFlag = 1;
            }
        }
    } while(iFlag == 1);
}
```

REFERÊNCIAS

DEITEL, H. M., DEITEL, P. J., 1999, *Como programar em C*, LTC - Livros Técnicos e Científicos Editora, Rio de Janeiro, Brasil.

KEINIGHAN, B. e RITCHIE, D., *C - A linguagem de programação*, Editora Campus.

MIZRAHI, V. V., 1990, *Treinamento em linguagem C - Curso completo. Módulos 1 e 2*, McGraw-Hill, São Paulo, Brasil.

PUGH, K., 1990, *Programando em linguagem C*, Trad.: José R. Martins, Roberto C. Mayer, McGraw-Hill, São Paulo, Brasil.

SCHELDT, H. *C Completo e Total.* Makron Books.