Índice

1 – CONCEITOS PRELIMINARES

Este capítulo visa mostrar alguns conceitos básicos, que são de grande importância para quem pretende se tornar um programador em linguagem C e trabalhar sob a plataforma Intel x86. O estudo de uma linguagem de programação não está ligada, na maioria das vezes, ao equipamento com o qual se está trabalhando. Entretanto, existem diferentes implementações para uma mesma linguagem, disponíveis para determinadas plataformas de trabalho. Neste curso, trabalhamos com as estruturas de linguagem C que estão disponíveis nas diversas implementações, mas iremos utilizar a plataforma Intel x86 com sistemas operacionais baseados na família Windows 9x (ou MSDOS).

Este fato alterará os recursos disponíveis para o desenvolvimento de sistemas, não a estrutura da linguagem em si, desta forma, serão mostrados alguns conceitos básicos de Hardware e Software de baixo nível, nessa plataforma.

1.1 – MICROCOMPUTADORES PADRÃO INTEL X86

Arquitetura Interna

Os equipamentos compatíveis com o padrão Intel x86 possuem uma arquitetura interna semelhante, que permite a compatibilidade, se desejado, entre os diversos modelos de processadores desde um PC-XT até os PCs-AT do 386 até os processadores de nova geração.

Internamente, os microprocessadores possuem algumas variáveis, chamadas de registradores, que são utilizadas pelas suas instruções a fim de obterem dados para processa-las ou para armazenarem resultados de instruções processadas. Existem vários tipos de registradores: os de uso geral, os registradores de pilha, os registradores de segmento, etc.

Registradores de uso Geral

São utilizados pelos microprocessadores para realizarem as tarefas básicas, como: implementar um contador em um ciclo de repetição, armazenar um dado que será escrito na memória ou receber um dado lido da memória, armazenar o resultado de uma operação aritmética, etc. Os registadores de uso geral são: AX, BX, CX e DX, todos de 16 bits, podendo ser divididos em dois registradores de 8 bits, chamados de parte menos significativa (L → Low) e parte mais significativa (H → High). Ficando: AL e AH, BL e BH, CL e CH, DL e DH.

O registrador AX (AL e AH) é chamado de acumulador e é o registrador mais utilizado no processamento de instruções.

Registradores de segmento e Segmentação de Memória

A memória dos microcomputadores é do tipo segmentada, ou seja, está dividida em segmentos ou páginas. Esse recurso foi desenvolvido para permitir a utilização de mais de 1Mb de memória, dispondo de registradores de 16 bits.

Existem quatro segmentos, bem definidos, que estarão presentes em qualquer programa, cada um representado por um registrador de segmento:

- **Segmento de código (CS)** – é onde são armazenadas as instruções do programa que serão executadas.
- **Segmento de dados (DS)** – é onde são armazenados os dados globais ou fixos do programa (mensagens, por exemplo).
- **Segmento de pilha (SS)** – é onde são armazenados os dados temporários de um programa. A pilha é um dispositivo auxiliar que permite o armazenamento de dados temporariamente, de forma que, os primeiros dados a serem armazenados são os últimos a saírem.
- **Segmento extra ou auxiliar (ES)** – é utilizado como auxiliar do segmento DS para manipulação de dados na memória.

O registrador de segmento funciona como uma referência, uma marca de início, a partir da qual são alojados dados ou instruções. Um segmento de memória tem um tamanho máximo de 64Kb. Para se poder utilizar ou endereçar cada um desses bytes, o registrador de segmento conta com um registrador auxiliar, que indica a posição o deslocamento ou "offset" a partir do início do segmento. Por exemplo: o segmento de código utiliza o registrador IP, ou ponteiro da pilha, para indicar a posição de memória em que está o "topo da pilha", ou seja, a primeira informação a ser resgatada.

Desta forma, para podermos utilizar ou endereçar a memória temos que trabalhar com um endereço composto, formado por um segmento (que indica a referência ou início) e um "offset" (que indica o deslocamento ou distância em bytes, a partir da referência).

Quando se quer indicar um endereçamento, no IBM-PC, normalmente utiliza-se a notação hexadecimal do segmento e do offset, separados por dois pontos ":". Exemplo: B800:0000, CS:0000, SS:SP, etc.

Interrupções

Um dos grandes recursos criados nos microcomputadores padrão Intel x86 são o de interrupções, que são as responsáveis pela manutenção da compatibilidade entre os diversos equipamentos existentes no mercado. A interrupção se constitui do desvio da execução de um programa para uma outra posição de memória, a fim de ser tratado algum evento ou ser realizada alguma tarefa específica, com o retorno, em seguida, para o local onde o processamento foi interrompido. As interrupções podem ser "geradas" de duas maneiras: por Hardware, quando algum dispositivo tiver sido acionado (teclado, placa de comunicação serial, relógio, drives, etc.); ou por Software, usando-se uma instrução específica do microprocessador. Em um microcomputador podemos ter até 256 interrupções.

Quando uma interrupção é gerada, é feita uma consulta a uma tabela, que fica localizada nos primeiros 1024 bytes da memória RAM (4 bytes para cada interrupção – 2 do segmento e 2 do offset), de onde é lido o endereço de memória para onde o processamento será desviado. Os endereços armazenados na tabela são chamados de "vetores de interrupções". Muitas dessas interrupções tem um uso pré-definido, como, por exemplo, a interrupção *05h*, que executa a cópia do conteúdo do vídeo para a impressora, quando a tecla Print Screen é pressionada. Outras ficam em aberto, para serem utilizadas pelos programas aplicativos.

A grande vantagem de se utilizar interrupções é que, para utilizarmos os recursos da máquina ou do sistema operacional, através de interrupções, será feita uma consulta na tabela de vetores para se saber qual é o endereço de memória para onde o programa deve ser desviado. Desta forma, se um fabricante quiser alterar as rotinas internas do seu microcomputador ou do seu sistema operacional, alterando assim o endereço de memória das mesmas, poderá faze-lo sem problemas; basta alterar também o conteúdo da tabela de vetores de interrupção, para manter a compatibilidade.

Interrupções do BIOS nos microcomputadores

Nos microcomputadores existem uma série de rotinas (BIOS – Basic Input Output System) para manipular os dados (armazenados no CMOS da placa mãe) dos periféricos conectados a MotherBoard. Essas rotinas são acessadas através de interrupções, chamadas de interrupções do BIOS, e vão da interrupção *00h* e *19h*. Uma rotina para "tratar" ou responder a uma interrupção *10h* permite acesso ao vídeo, possuindo procedimentos de mudança de modo, mudança ou leitura da posição do cursor, mudança de cor, etc. Cada um desses procedimentos deve ser selecionado antes da chamada da interrupção e, normalmente, isto é feito usando-se o registrador AH. De acordo com o valor que estiver armazenado em AH, a rotina que responderá à interrupção irá executar um procedimento diferente.

1.2 – MODOS DE MEMÓRIA NOS MICROCOMPUTADORES

Todo programa desenvolvido para microcomputadores utiliza os registradores de segmentos descritos anteriormente. Entretanto, como cada segmento só pode ter até 64Kb, teríamos uma limitação de criar programas como no máximo 64Kb de código e 64Kb de dados fixos. Para solucionar esse problema, criou-se um artifício para se construir programas, alterando a forma de se manusear os registradores de segmento. São os modos de memória destacados abaixo:

- **Modo SMALL**: nesse modo, todos os registradores de segmento apontam para a mesma posição de memória. Desta forma, o tamanho do programa, somando as suas interrupções, os dados fixos e a pilha, não podem exceder 64Kb. É utilizado para a construção de programas pequenos e tem a vantagem que, para se manipular os dados, a pilha e as interrupções só há a necessidade de se alterar o "offset", pois os segmentos são fixos. Isso torna o programa menor e mais rápido.
- **Modo MEDIUM** : nesse modo, o registrador do segmento de dados e o da pilha apontam para a mesma posição da memória, podendo juntos ocupar até 64Kb. O registrador de segmento de código não tem valor fixo, podendo ser alterado de acordo com a necessidade. Normalmente, para cada módulo de um programa, define-se um valor para o segmento de código. Desta forma, o conjunto de instruções de um programa pode ocupar até 1Mb de memória. É utilizado para a construção de programas extensos, mas que não utilizaram uma grande quantidade de dados. Para manipular os dados e a pilha só há necessidade de se alterar o "offset", pois os segmentos estão fixos. Para manipular as instruções há a necessidade de se alterar tanto o segmento como o "offset", deixando o processamento um pouco mais lento.
- **Modo LARGE**: nesse modo, o registrador de segmentos de dados não tem valor fixo, podendo ser alterado de acordo com a necessidade. Normalmente, para cada variável ou grupo de variáveis de um programa, define-se um valor para o segmento de dados. O registrador de segmento de código não tem valor fixo, podendo ser alterado de acordo com a necessidade. Normalmente, para cada módulo de um programa, define-se um valor para o segmento de código. Desta forma, tanto o conjunto de dados como o conjunto de instruções de um programa podem ocupar até 1Mb de memória, cada um. É utilizado para a construção de programas extensos, que utilizaram uma grande quantidade de dados. Para manipular os dados e as instruções há necessidade de se alterar tanto o segmento quanto o "offset", tornando o programa maior e um pouco mais lento.

1.3 – O SISTEMA OPERACIONAL

Interrupções do Sistema Operacional

O sistema operacional também utiliza interrupções para realizar as suas operações de baixo nível, como: abrir, ler, gravar ou fechar um arquivo, ler ou alterar um vetor de interrupção, acessar o teclado ou vídeo, etc. As principais interrupções do sistemas operacional estão entre *20h* e *24h*, sendo que a interrupção *21h* é a que permite manipular arquivos e periféricos, sendo a principal delas (possui mais de 100 funções diferentes).

Dispositivos do Sistema Operacional

O S.O. possui alguns dispositivos padrões. São eles: COM (console), que funciona como entrada (teclado) ou saída (vídeo); COMn, que funciona como entrada e saída através da placa de comunicação serial; PRN, que funciona como a saída para a impressora, sendo a porta padrão de impressão; e LPTn, que funciona como entrada e saída através da placa de comunicação paralela.

Estes dispositivos são utilizados pela maioria das linguagens de programação, a fim de permitirem o acesso aos equipamentos periféricos. Desta forma, cria-se um certa transparência, pois sabe-se que se está utilizando um determinado dispositivo de E/S, mas cabe ao sistemas operacional saber como acionar o equipamento periférico correspondente ao dispositivo.

Além disso, essa transparência ainda permite o redirecionamento dos dispositivos, ou seja, pode-se redirecionar a saída do console para a impressora, ou vice-versa, por exemplo.

Quando um programa é executado sob o Sistema Operacional, este reserva cinco manipuladores (handles) para o acesso aos seus dispositivos. São eles: STDOUT (saída padrão), STDIN (entrada padrão), STDPRN (impressora auxiliar), STDERR (saída padrão das mensagens de erro) e STDAUX (manipulador auxiliar). Qualquer alteração ou configuração feita para esses dispositivos, através de algum comando do S.O. (comando MODE, comando de redirecionamento ">" e "<", etc.), será válida dentro dos programas que os utilizam.

Divisão de memória para o Sistema Operacional

Em microcomputadores padrão Intel x86 a memória pode receber algumas denominações dependendo de seu tamanho em bytes e da forma como é manipulada pelo S.O. Temos então:

- **Memória baixa ou principal (LM)** – possui até 640Kb e é onde o S.O. e os aplicativos 16 bits são armazenados e executados (na maioria das vezes).
- **Memória alta (HMA ou UMB)** – é a parte da memória entre 640Kb e 1Mb, conhecida como bloco de memória alta. Normalmente, é utilizada pelo S.O. para gravar parte de seus arquivos ou dispositivos instalados e programas residentes. É o local onde o ambiente gráfico do Windows é executado. Somente fica disponível se existir a instrução HIMEM.SYS no arquivo de configuração da máquina.
- **Memória expandida (EMS)** – é a memória disponível além dos 640Kb da memória principal. Para ser usada é necessário a utilização de um programa especial chamado de gerenciador de memória expandida. Esta memória é gerenciada sob a forma de páginas ou blocos.
- **Memória estendida (XMS)** – é a memória disponível acima de 1Mb em computadores AT e posteriores. Esta memória é gerenciada de forma contínua e não é acessada pela maioria dos programas do Sistema Operacional, sendo normalmente utilizada pelo gerenciador de memória expandida para aumentar a quantidade de memória EMS.

É importante saber que, num computador com sistemas operacionais 16 bits, os programas só conseguem se alojar e serem executados na memória principal. A memória EMS é utilizada para o armazenamento provisório de dados ou de partes do programa executável. Para isso, o driver de memória EMS deverá estar instalado e o programa deverá ser projetado de uma forma que consiga gerenciar as informações que serão armazenadas na memória EMS.

2 – INTRODUÇÃO A LINGUAGEM C

2.1 – CARACTERÍSTICAS PRINCIPAIS

A linguagem C foi criada por Dennis M. Ritchie e Ken Thompson em 1972, nos Laboratórios Bell, e foi baseada na linguagem B (Thompson), que era uma evolução da linguagem BCPL.

A linguagem começou a ter sucesso após a edição do livro "The C Programming Language", de Brian Kernigham e Dennis M. Ritchie, que descreve todas as características da linguagem, como foi projetada inicialmente e deve ser leitura obrigatória para os programadores de C.

Hoje, a linguagem C sofreu modificações e possui características de linguagem visual com embutimento do conceito de Orientação a Objeto, chamada atualmente de C++, ela é utilizada no desenvolvimento de diversos tipos de sistemas, entre eles, planilhas eletrônicas, editores de texto, gerenciadores de bancos de dados, sistemas de comunicação de dados, etc. Os principais programas existentes no mercado foram desenvolvidos parcialmente ou totalmente em linguagem C.

A linguagem C permite aos programadores desenvolverem programas estruturados e modulares; permite a reutilização de código através da construção de bibliotecas; os programas executáveis construídos em C são rápidos e compactos; além de serem portáveis, ou seja, podem ser transferidos de uma plataforma de trabalho para outra, sem grandes modificações.

Por exemplo: de um computador com sistema operacional padrão Microsoft (Windows 9x) para uma estação RISC com UNIX. Isto é possível porque os compiladores em Linguagem C são padronizados, na sua maioria.

2.2 – ESTRUTURA DE UM PROGRAMA EM C

Um programa em C é constituído de um ou mais módulos ou sub-rotinas chamadas de funções. Uma função simples em C é identificada por um nome, seguido de parêntesis "()". Para marcar o início da função, usa-se uma chave de abertura "{", e para identificar o final da função usa-se uma chave de fechamento "}". Normalmente, utiliza-se apenas letras minúsculas. Exemplo:

```
nomefc ()
{

}
```

Todo programa em C deverá conter pelo menos uma função, chamada main(), que significa principal. É através da função main() que o programa irá iniciar. As instruções devem ser escritas dentro das chaves e finalizadas por um ponto-e-vírgula ";". Espaços em branco são suprimidos pelos compiladores C. Exemplo:

```
main()
{
        instrucao1;     instrucao2;
        instrucao3;
}
```

A função main() tem que ser escrita em letras minúsculas, bem como todas as palavras-chave ou palavras-reservadas da linguagem C. Isto é necessário porque os compiladores C são "Case Sensitive" – Sensíveis ao Caso, isto é, eles fazem distinção entre palavras escritas em letras minúsculas ou maiúsculas. Por exemplo: for ≠≠ For ≠≠ FOR.

2.3 – PALAVRAS RESERVADAS EM C

As palavras reservadas em linguagem C são:

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

Observe que alguns compiladores, que possuem outras implementações para a linguagem C, têm outras palavras reservadas. No caso do Turbo C++ podemos destacar:

asm	_ds	interrupt	_seg
cdecl	far	near	_ss
_cs	huge	pascal	

2.4 – VARIÁVEIS E CONSTANTES

Em C podemos utilizar variáveis e constantes de diversos tipos. Todas devem ser devidamente declaradas antes de sua utilização, para permitir que a memória seja corretamente reservada e para que o compilador as manipule da forma correta.

Tipos de dados básicos

Em C existem quatro tipos básicos de dados:

Tipos	Bytes	Universo
char	1	-128 a 127
int	2	-32768 a 32767
float	4	3.4e-38 a 3.4e+38
double	8	1.7e-308 a 1.7e+308

Os tipos *char* e *int* permitem o armazenamento de números inteiros, positivos e negativos e são utilizados para armazenar caracteres ou resultados lógicos, implementarem um contador, etc. Esses tipos são armazenados na memória em notação binária: o byte menos significativo e, em seguida, o mais significativo (para o caso do *int*); o bit mais significativo é utilizado para indicar o sinal.

Os tipos *float* e *double* permitem o armazenamento de números reais, positivos e negativos e são utilizados em cálculos científicos, financeiros, etc. Esses números são armazenados na memória em notação binária, da seguinte forma: 1 bit indica o sinal, M bits indicam a mantissa e os E bits restantes indicam o expoente, por exemplo, no caso de uma variável *float* M=25 e E=8.

Modificadores

Em C existem ainda os modificadores que alteram a forma dos dados a serem tratados. São eles:

Modificador	Exemplo
short	short int
long	long int ou long double
unsigned	unsigned char ou unsigned int

O modificador *short*, em alguns computadores, tem a metade do tamanho de um inteiro *int*. O modificador *long* dobra o tamanho do número inteiro ou real, permitindo uma maior capacidade de armazenamento. O tipo *long double* só é aceito em alguns compiladores C. Ao invés de se escrever *long int*, pode-se usar apenas *long*.

O modificador *unsigned* faz com que o compilador trate o dado como positivo. Desta forma, a capacidade de armazenamento dobra. Uma variável *char* pode armazenar um número entre 0 e 255, e uma variável *int* de 0 a 65535. Ao invés de se escrever *unsigned int*, pode-se usar apenas *unsigned*.

Declaração de variáveis e constantes

A declaração de variáveis em C é constituída pelo tido de dado, seguido de uma lista de nomes de variáveis de mesmo tipo a serem declaradas, separadas por vírgula, e finalizada com um ponto-e-vírgula. Exemplo:

```
char a;
int b,c;
long int Tipo, TIPO, tipo;
double Soma, resultado, Dt_lanc;
```

A declaração de uma constante é precedida da palavra *cons* e após o nome da constante deve-se inicializá-la, usando o operador de igualdade "=". Exemplo:

```
const x = 10;
const y = 3.14;
const float CONSTANTE = 25.4;
```

Observe que podemos ter constantes sem a definição de tipo. Neste caso o compilador irá utilizar um tipo equivalente ao valor atribuído. No exemplo acima, *x* será do tipo *int* e *y* será do tipo *float*.

O nome de uma variável ou constante pode ter qualquer tamanho, entretanto, apenas alguns caracteres serão reconhecidos, ou seja, são significativos. Normalmente, esse número é de 32 caracteres, mas pode ser configurado na maioria dos compiladores. Nenhuma palavra-chave pode ser utilizada como nome de variável ou constante. Variáveis e constantes escritas com letra maiúsculas ou minúsculas são diferentes.

Inicialização de variáveis

As variáveis podem ser inicializadas durante sua declaração e/ou utilizando-se uma instrução de atribuição. Por exemplo:

```
main()
{
        float dado1 = 1.2, dado2 = 2.5, dado3 = 4.7, dado4;
        float media;

dado4 = 1.9;
media = (dado1 + dado2 + dado3 + dado4) / 4;
}
```

As duas primeiras instruções declaram as variáveis que serão utilizadas e já inicializam três delas. As duas últimas exemplificam instruções de atribuição.

Tipos de dados lógicos ou booleanos

Em C não existe um tipo de dado para armazenar dados lógicos ou booleanos, ou seja, dados que possuem apenas um estado dual (verdadeiro ou falso). Neste caso qualquer valor diferente de zero representa um valor verdadeiro; enquanto que o zero representa um valor falso. Se for necessário, pode-se simular um tipo de dado booleano, para melhor compreensão do programa. Exemplo;

a = 1; ← representa um valor lógico verdadeiro
b = 0; ← representa um valor lógico falso
c = -8 ← representa um valor lógico verdadeiro

2.5 – OPERADORES

A linguagem C é rica em operadores, que são representados por um símbolo (um ou mais caracteres) e são utilizados para "operar" ou "manipular" dados, sob a forma de constantes ou variáveis. Talvez a linguagem C seja a mais rica em operadores; mas deve ser a mais pobre em símbolos, pois existem operadores diferentes com o mesmo símbolo.

Operadores aritméticos

Operador	Significado
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo ou resto da divisão
=	Atribuição
-	Menos unário – sinal negativo

Todos os operandos, com exceção do menos unário, são operadores binários, ou seja, possuem dois operandos. Outro detalhe que deve ser observado é o tipo dos dados (operandos) que estão sendo utilizados.

O operador de atribuição exige que o operando da esquerda seja uma variável (inteira ou real), para que possa armazenar o valor correspondente ao operando da direita. Exemplo:

```
a = 10;
b = 10;
c = -1;
d = a + b + c;

15 = a; ← isto é inválido
```

Além disso, é permitido inicializar mais de uma variável, com o mesmo valor, numa única instrução. Exemplo:

```
a = b = 10;
c = a * b;
```

Recomenda-se, por questão de organização e para evitar erros, que os operandos do operador de atribuição sejam do mesmo tipo. A linguagem C permite que operandos de tipos diferentes sejam utilizados em conjunto e que o resultado seja atribuído a outro operando, do mesmo tipo dos anteriores, ou não. Para isso o compilador utiliza um recurso chamado conversão de tipos de dados, que será visto mais adiante. Exemplo:

```
int a,b = 1;
float c = 2.5;

a = b + c;
```

Neste caso, “a = 3”, pois é uma variável inteira.

O operador unário opera sobre o operando da sua direita, que pode ser uma variável, uma constante ou uma expressão, inteira ou real. Exemplo:

```
a = -1;
b = - 10 + a;
c = - (b + 4);
```

Operadores aritméticos de atribuição

Operador	Exemplo	Significado
+=	op1 += op2	Atribui a op1 o resultado de op1 + op2
-=	op1 -= op2	Atribui a op1 o resultado de op1 - op2
*=	op1 *= op2	Atribui a op1 o resultado de op1 * op2
/=	op1 /= op2	Atribui a op1 o resultado de op1 / op2
%=	op1 %= op2	Atribui a op1 o resultado de op1 % op2

Como pode ser observado, os operadores aritméticos de atribuição simplificam as instruções de atribuição em que o operando da esquerda também aparece do lado direito do operador de atribuição. Por exemplo:

```
a = 10;                 ou          a = 10;
a = a + 10;                         a += 10;
```

Operadores de incremento e decremento

Operador	Significado
++	Incremento de um
--	Decremento de um

Os operadores de incremento e decremento são unários e operam apenas variáveis inteiras (char ou int). Quando a instrução for apenas o incremento de uma variável, podem ser colocados antes do operando ou depois do operando. Exemplo:

```
int a = 1, b = 1;

a++;        ou          ++a;        ← a=2
b--;        ou          --b;        ← b=0
```

Quando forem usados dentro de expressões, a sua posição em relação ao operando não irá mudar o valor final do operando, mas irá mudar o valor final da expressão. Desta forma, deve-se observar que:

- Se o operador é escrito antes do operando, primeiro se realiza a operação de incremento ou decremento para, em seguida, se calcular o valor final da expressão.
- Se o operador é escrito depois do operando, primeiro se calcula o valor final da expressão para, em seguida, realizar a operação de incremento ou decremento.

Exemplos:

```
a = b = 2;
c = ++a;                ← c = 3 e a = 3
d = b--;                ← d = 2 e b = 1
e = c++ - --d;          ← e = 2, c = 4 e d = 1
```

Operadores binários (Bit a bit)

Os operadores binários são aqueles que realizam as operações “e”, “ou”, “não” ou deslocamentos (shift), sobre cada bit de operandos inteiros (char ou int). São, na verdade, operadores aritméticos, só que operam sobre a representação binária dos números inteiros.

Operador	Significado
&	"e" binário
|	"ou" binário
~	"não" binário
>>	Deslocamento para esquerda
<<	Deslocamento para direita

Os operadores & e | possuem dois operandos e o operador ~ possui apenas um. O resultado de sua aplicação pode ser analisado de acordo com a tabela a seguir, só que bit a bit.

Bit1	Bit2	Bit1 & Bit2	Bit1 | Bit2	~Bit1
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Exemplo:

```
char a, b, c, d, e;

a = 0x40;  /* 0x indica notação hexadecimal */
b = 0x2F;

c = a & b;  /* c = 0 */
d = a | b; /* d = 0x6F */
e = ~a; /* e = 0xBF */
```

Os operadores ">>" e "<<" possuem dois operandos: o da esquerda é a variável inteira a ser operada e o da direita é um valor ou variável inteira que indica o número de deslocamentos de bits que devem ser feitos. Exemplo:

a >> 1 ← deslocar 1 bit á esquerda ➔ a = 0x80
a << 3 ← deslocar 3 bits á direita ➔ b = 0x05

Como pode ser notado, deslocar á esquerda é o mesmo que multiplicar por 2; e deslocar á direita é o mesmo que dividir por 2.

Operador de endereço

O operador endereço "&" é utilizado sobre uma variável para representar o seu endereço de memória. Isto é muito utilizado em C, inclusive quando se for passar parâmetros para funções. O valor do endereço é sempre um número inteiro sem sinal, e representa o offset da posição da variável em relação ao segmento de dados ou do segmento da pilha.

```
int a = 10;
unsigned int b;

b = &a;          ← b é igual ao endereço de memória da variável "a"
```

2.6 – EXPRESSÕES

Expressões em C são um conjunto de variáveis e operadores organizados em uma certa ordem, que são resolvidas e retornam resultados aritméticos ou lógicos. As expressões são avaliadas sempre da esquerda para a direita, mas deve-se levar em conta dois fatores: a precedência dos operadores, ou seja, a prioridade que um operador tem sobre o outro; e o uso dos parênteses, que altera a seqüência de avaliação da expressão.

Procedência de operadores

Os operadores de incremento e decremento têm precedência maior que os aritméticos. Os operadores aritméticos de multiplicação e divisão têm precedência maior que os de soma e subração.

Os operadores aritméticos têm precedência sobre os relacionais e os lógicos, que serão vistos depois.

A seguir é mostrada a tabela de precedência de operadores em linguagem C.

OPERADORES	TIPOS
! - ++ --	"não lógico" e menos unário incremento / decremento
* / &	aritméticos
+ -	aritméticos
< > <= >=	relacionais
== !=	relacionais
&& ||	lógicos
= += -= *= / =	aritméticos de atribuição

Expressões aritméticas

Uma expressão aritmética irá resultar sempre em um resultado inteiro ou real, cujo tipo do resultado será definido pelo maior tipo existente na expressão. Exemplo:

```
int  a;
long  b;
float  c;
double  d;

d = a + b;  ← resultado "long" atribuído a um "double"
a = c + b;  ← resultado "float" atribuído a um "int"
```

Conversão de tipos de dados (cast)

As conversões entre os diversos tipos de dados são feitas naturalmente pelo compilador C. Entretanto, muitas vezes, essas conversões causam perdas de precisão ou de informação (algarismos após o ponto decimal). Para evitar que as conversões sejam feitas de acordo com o critério do compilador, podemos utilizar um "cast", que é uma forma de forçar a conversão para um determinado tipo de dado. Esse "cast" é constituído de um tipo de dado, entre parêntesis, colocado antes da expressão, ou da parte da expressão, que deverá ser convertida. Exemplo:

```
d  =  (double)  (a + b);      ←  converte a+b para "double"
a  =  (int)  ((long) c + b);  ←  converte c para "long" e c+b para "int"
```

2.7 – COMENTÁRIOS

Em linguagem C podemos escrever comentários, ou seja, informações de orientação do programador que não devem ser processadas pelo compilador. Um comentário em C pode ter várias linhas e deve ser iniciado pelos caracteres "*/". Exemplo:

```
a = 0;   /*  Inicializando a variável  a */  ou,

/* Arquivo de exemplo : exemplo1.c
   Construído em Linguagem C
*/
```

Em C, não se pode colocar um comentário dentro de outro.

Alguns compiladores, como o TC++, aceitam também um comentário de uma linha, utilizando-se os caracteres "//". Exemplo:

```
a = 0;  // Inicializando a variável  a     ou,

//  Arquivo de exemplo : exemplo1.c
//  Construído em Linguagem C
```

3 – ESTRUTURAS DE CONTROLE DE FLUXO

3.1 – OPERADORES RELACIONAIS E LÓGICOS

Os operadores relacionais e lógicos são utilizados nas expressões de controle das estruturas de decisão e repetição condicional.

Operadores relacionais

Operador	Significado
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a
==	igual a
!=	diferente de

Os operadores relacionais são utilizados em expressões de comparação ou expressões lógicas, onde o resultado esperado não é um número, e sim, um resultado lógico verdadeiro ou falso. Exemplo:

```
a = b = 10;
c = 20;
d = 40;

a > b       "a" é maior que "b"?          Falso
a == b      "a" é igual a  "b"?           Verdadeiro
d >= c      "d" é maior ou igual a "c"?   Verdadeiro
c <= a      "c" é menor ou igual a "a"?   Falso
```

Observe que, no exemplo acima “a, b, c e d” são variáveis, mas poderiam ser expressões aritméticas ou até mesmo relacionais. Exemplo:

a + b == c Verdadeiro
d - 2*c <= 0 Verdadeiro
(a > b) == (d >= c) Falso
(a == b) > (c <= a) Verdadeiro

A última expressão só é possível porque cada expressão relacional, após ser avaliada, gera um resultado 1 ou 0, representando verdadeiro e falso, respectivamente.

Lógicos

OPERADOR	SIGNIFICADO
&&	“e” lógico
22	“ou” lógico
!	“não” lógico

Os operadores lógicos são utilizados em expressões de comparação ou expressões lógicas, e tem como operandos valores lógicos, e não valores aritméticos. Eles reúnem os resultados das expressões lógicas que compõem seus operandos e realizam uma outra operação lógica, de acordo com a álgebra de BOOLE, como mostrado na tabela a seguir.

op1	op2	op1&&op2	op1||op2	!op1
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

As regras básicas para a composição de expressões utilizando-se os operadores “&&” e “22”, são:

- O resultado de uma expressão lógica “e” só é verdadeiro, se todos os operandos forem verdadeiros.
- O resultado de uma expressão lógica “ou” é verdadeiro, se pelo menos um dos seus operandos for verdadeiro.

a = 1; b = 0; c = 1;

a > b && c == 1 → V && V ➔Verdadeiro
a == b && c >= 0 → F && V ➔Falso

a > b 22 c == 1 → V 22 V ➔Verdadeiro
a == b 22 c >= 0 → F 22 V ➔Verdadeiro

O operador “não” é unário e opera sobre o operando que estiver a sua direita, seja ele uma variável, uma constante ou uma expressão lógica ou aritmética. Exemplos:

!(a > b) → !V ➔Falso
!(a > b) && b + 1 → F && V ➔Falso
a == b 22 !b → F 22 V ➔ Verdadeiro

Observe que, nos dois últimos exemplos, foram utilizados uma expressão aritmética “b+1” e uma variável “b”. Nesses casos, para podermos avaliar o resultado lógico, devemos seguir a regra citada anteriormente: se a variável ou a expressão aritmética tiver como resultado um valor diferente de 0 (zero), seu resultado lógico é verdadeiro, caso contrário, é falso.

3.2 – ESTRUTURAS DE DECISÃO

Estrutura Condicional

A estrutura condicional permite uma tomada de decisão durante a execução do programa. Pode ser escrita da seguinte maneira:

```
if (expressão) procedimento;
```

ou

```
if (expressão)
{
        procedimento1;
        procedimento2;
        ....
        procedimento n;
}
```

A expressão pode ser tanto aritmética quanto lógica, pode conter chamadas a funções, etc. A decisão é tomada da seguinte forma: se a "expressão" for verdadeira, a instrução ou o conjunto de instruções será executado.

OBS: Deve ser observado que quanto tivermos mais de uma instrução para ser executada, dentro de uma estrutura de decisão ou repetição, elas devem vir "marcadas" por um bloco, ou seja, entre chaves. Essa regra vale para todas as estruturas, com exceção do SWITCH.

A estrutura condicional possui ainda uma outra variação que é utilizada para implementar procedimentos para a condição verdadeira ou falsa, como pode ser descrita abaixo:

```
if (expressão)
        procedimentoV;
else
        procedimentoF;
```

Pode-se escrever estruturas condicionais aninhadas, isto é, uma dentro da outra para testar diversas condições diferentes. Exemplo:

```
if (expressãoA)
        procedimento1;
else {
        procedimento2;
        if (expressãoB)
                procedimento3;
        else
                procedimento4;
}
```

Operador Ternário

Existe um operador em Linguagem C, chamado de operador ternário, cujo símbolo é uma interrogação ?. Pode ser utilizado para substituir pequenas estruturas condicionais. Sua sintaxe é:

Expressão ? instruçãoV : instruçãoF;

Se a expressão for verdadeira, a instruçãoV será executada; caso contrário, a instruçãoF será executada. Exemplo:

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

Estrutura de casos condicionais

A estrutura de casos condicionais, em C conhecida como SWITCH, é utilizada quando a tomada de decisões envolve várias alternativas, de acordo com o resultado de uma mesma expressão, evitando a utilização de IF aninhados. Sua sintaxe é:

```
switch (expressão) {
        case valor1 :  instrução1;
                break;
        case valor2 : instrução2;
                break;
        .....
        default : instruçãoD;
                break;
}
```

A expressão deve resultar um valor constante e inteiro; valor1, valor2, ..., são os possíveis resultados da expressão que serão comparados e devem ser constantes inteiras; *default* é uma construção opcional, que será executada se nenhum dos valores for igual a expressão.

A instrução BREAK indica o término da lista de instruções para cada caso. Na estrutura SWITCH, não é necessário o uso das chaves, caso deseja-se executar mais de uma instrução. O importante é lembrar que, para finalizar as instruções, deve-se colocar um BREAK; caso contrário, a execução continuará até encontrar algum BREAK ou o fim da estrutura SWITCH.

3.3 – ESTRUTURAS REPETITIVAS

Estrutura repetitiva finita

Esta estrutura permite a repetição de uma ou mais instruções, um número fixo de vezes, utilizando-se para isso de um contador (variável inteira). Nas linguagens de programação mais conhecidas, temos um contador, cujo valor inicial é VI e que vai até o valor final VF, com um incremento INC. Por exemplo: um contador A, que começa com 1 e vai até 10, saltando de 1 em 1. Isto significa que as instruções serão executadas 10 vezes, e em cada repetição o contador A terá um valor diferente.

Em C, a estrutura FOR é composta por três expressões, separadas por ";", dentro de parênteses. A primeira expressão é a de inicialização, a segunda é de comparação e a terceira é de incremento.

for (inicialização; comparação; incremento) instrução; ou

```
for (inicialização; comparação; incremento) {
        procedimentos;
}
```

Exemplo:

```
for (a=1; a<=10; a++) printf("%d", a);
```

A lógica da estrutura FOR é a seguinte:

1. Primeiro, a expressão de inicialização é avaliada;
2. Em seguida, a expressão de comparação é avaliada. Se for verdadeira, as instruções serão executadas; caso contrário, abandona-se a estrutura FOR.
3. Finalmente, após a execução das instruções, a expressão de incremento é avaliada e a execução volta para o passo 2.

O mais interessante é que qualquer expressão C é aceita nos três campos da estrutura FOR e nenhuma delas precisa ter uma variável contadora; é opcional. Além disso, os três campos são opcionais, mas deve-se manter os ";". Exemplo:

```
i = 0;
for ( ; i<10; ) {
        printf ('Mensagem');
        i++;
}
```

A estrutura FOR também permite outra estrutura FOR como instrução. São os FOR aninhados. Por exemplo:

```
for ( i=0; i<10; i++)
        for ( j=0; j<5; j++) {
                printf ("%d - %d", i, j);
        }
```

Estrutura repetitiva condicional

Esta estrutura é utilizada para repetir uma ou mais instruções enquanto uma condição for verdadeira. Essa estrutura pode ser substituída pela estrutura repetitiva finita, mas isto não é feito devido às técnicas de programação estruturada.

Sintaxe:

while (condição) procedimento; ou

```
while (condição) {
        instrução1;
        instrução2;
}
```

Enquanto a condição for verdadeira, as instruções serão executadas; caso contrário, a estrutura WHILE é abandonada. Deve-se observar que, a condição é avaliada antes da execução das instruções e que seus componentes já devem estar previamente inicializados. Exemplos:

```
char i, j;
i = 0;
printf ("Digite qualquer valor para continuar ou zero para sair");
while (i >= 0) scanf (i);
j = 10; /* Equivalente ao FOR */
while (j > 0) {
        printf ("valor = %d", j);
        j--;
}
```

Como uma variação deste tipo de estrutura pode-se citar a estrutura DO-WHILE que difere da estrutura WHILE pela ordem de avaliação da condição. Na estrutura WHILE a condição é avaliada na entrada da estrutura e cada vez que ela completa o loop. Na estrutura DO-WHILE a condição é avaliada no final da estrutura, ou seja, neste caso pelo menos uma vez a estrutura é executada. Sintaxe:

```
do
        procedimento
while (condição);

do{
        procedimento1;
        procedimento2;
        procedimento3;
} while (condição);
```

Pode-se utilizar uma instrução para finalizar o processo das estruturas de repetição antes do limite final da sua execução. Exemplo:

```
printf ("Digite um número ou zero para finalizar");
for ( ; ; ) {
        scanf (a);
        if (a==0) break; /* quando a=0 o loop infinito terminará */
}
```

A instrução CONTINUE pode ser utilizada, em qualquer estrutura de repetição e seu efeito é o de forçar o reinício do processo de repetição. Na estrutura FOR, o processamento passa para a expressão de incremento, enquanto que nas estruturas WHILE e DO-WHILE, o processamento passa para a condição. Exemplo:

```
for (i = 1; i < 1000; i++) {
        if (i % 2)
                continue;
        else
                printf ("%d", i);
}
```

3.4 – COMANDOS BÁSICOS DE ENTRADA E SAÍDA

Para existir uma certa interação com o usuário é necessário trabalhar com as entradas e saídas padrão do sistema operacional que são : o monitor para exibição de mensagens e resultados de processamento e o teclado para entrada de dados solicitados ao usuário.

Na biblioteca padrão de controle de Entrada/Saída (STDIO.H) existem funções que podem ser usadas para esta finalidade. São elas:

Printf( )

Função para saída de dados usando a saída padrão (stdout) do Sistema Operacional (monitor), sua sintaxe é:

```
printf ("string de controle", argumentos);
```

Como *string de controle* entenda-se uma frase com caracteres e constantes especiais usadas para formatação das saídas dos resultados. Os argumentos são valores constantes ou variáveis cujos conteúdos serão substituídos nos caracteres de formatação da string de controle. Existem ainda algumas constantes para controle da apresentação das quais destaca-se:

Códigos Especiais	Significado
\n	nova linha
\t	TAB
\b	retrocesso
\"	aspas
\\	barra
\f	salta página
\o	nulo

Como principais caracteres de formatação pode-se destacar:

Formatador	Descrição
%c	caractere simples
%d	Decimal
%e	notação científica
%f	ponto flutuante
%g	%e ou %f (mais curto)
%o	Octal
%s	cadeia de caracteres
%u	decimal sem sinal
%x	Hexadecimal

scanf( )

Função para solicitação de dados ao usuário usando a entrada padrão (stdin) do sistema operacional (teclado), sua sintaxe é:

scanf("string de controle", argumentos);

A string de controle conta com o uso dos formatadores para limitar a digitação do usuário, como argumentos entenda-se o conjunto de variáveis que irão receber o conteúdo digitado. Neste caso existe uma notação especial que passa o endereço de memória onde a variável foi reservada, para passar este endereço basta utilizar o operador &. Este operador diz que o conteúdo será armazenado na posição de memória da variável que é precedida por ele.

4 – CONSTRUINDO PROGRAMAS EM C

4.1 – COMANDOS DO PRÉ-PROCESSADOR

Todo compilador C possui um pré-processador, que implementa os seguintes recursos: inclusão de arquivos, compilação condicional, definição de constantes, macros, etc. Esse pré-processador é executado antes do compilador e "prepara o caminho" para o mesmo.

Diretiva #INCLUDE

Esta diretiva permite a inclusão de um arquivo, dentro do outro, apenas para a compilação do mesmo, ou seja, o conteúdo do arquivo incluído será compilado, mas não será anexado ao arquivo fonte, que está sendo compilado. Sintaxe:

#include <nomearq>
#include "nomearq"

Esta diretiva é utilizada, normalmente, para a inclusão dos arquivos de cabeçalho (arquivo.H), que são aqueles que contém a definição de variáveis e funções globais. Todo compilador C vem acompanhado por uma biblioteca de funções. Cada grupo de funções possui um arquivo de cabeçalho. Por exemplo, MATH.H é o arquivo de cabeçalho para as funções matemáticas. Exemplo:

#include <math.h>
#include "math.h"

A diferença entre usar < > ou " " está no fato de que no primeiro caso o compilador irá procurar o arquivo em um diretório previamente configurado, no segundo caso o arquivo será procurado no diretório atual.

Diretiva #DEFINE

Essa diretiva permite a criação de macros, que podem ser nomes de constantes ou seqüências de instruções, por exemplo. As macros serão substituídas pelo pré-processador, antes da compilação. Não é feita nenhuma alteração no arquivo fonte. Sintaxe:

#define NOMEMACRO textomacro

É comum, mas não obrigatório, criar macros ou constantes com letras maiúsculas, para melhor diferencia-las das variáveis e instruções da linguagem C, que normalmente são escritas em letra minúscula. Exemplos:

#define COR_TEXTO 15
#define TESTA_FLAG flag>10?fim=1:fim=0

Uma vez definidas essas macros, podemos utiliza-las no programa, da seguinte forma:

if (cor == COR_TEXTO) procedimento;

TESTA_FLAG;
if (fim) printf ("Abandonar o programa");

Diretiva #IF

As diretivas #IF permitem fazer com que o compilador tome uma decisão. Por exemplo: se ele irá compilar uma parte do programa, ou outra. Isso é chamado de compilação condicional e é muito útil quanto se tem um trecho de programa que pode ser utilizado por vários programas, com apenas algumas alterações. As estruturas das diretivas são:

```
#if EXP_CONST        /* se EXP_CONST for verdadeira */
        procedimentosV;
#else                /* senão */
        procedimentosF;
#endif               /* fim da diretiva #if */

#if CTE1             /* se CTE1 for verdadeira */
```

```
#elif CTE2          /* senão, se CTE2 for verdadeira */
#endif              /* fim da diretiva */

#ifdef NOME         /* se NOME estiver definido */
#else               /* senão */
#endif              /* final da diretiva */

#ifndef NOME2       /* se NOME2 não estiver definido */
#else               /* senão */
#endif              /* final da diretiva */
```

4.2 – CONSTRUÇÕES E USO DE FUNÇÕES

Construção

Em linguagem C podemos ter várias funções em um mesmo arquivo. Cada função deve ser construída, uma em baixo da outra. A posição da função no arquivo não importa muito (na maioria das vezes). O programa irá sempre começar pela função *main()*, independentemente da posição em que esta estiver no arquivo. As demais funções só serão executadas se forem chamadas através da função *main()*; ou de outra função, que tenha sido previamente chamada através da função *main()*. Isto significa que: "o simples fato de uma função ter sido construída não implica que ela será executada".

Exemplo de construção de funções:

```
funcao1()
{
        procedimentosA;
}

funcao2()
{
        procedimentosB;
}

main()
{
        procedimento1;
        funcao1();
        procedimento2;
}
```

A construção mais simples de uma função em C é a demonstrada acima: o nome da função seguido de parênteses, mais o corpo da função (procedimentos), entre chaves.

Chamada de funções

No exemplo acima, a instrução na função *main()* apresentada como *funcao1()* constitui uma chamada de função. O programa se inicia em *main()* o procedimento1 é executado e, posteriormente, é chamada a *funcao1()* e seus procedimentos são executados, ao final da execução desta função o controle de processamento volta para a função *main()* que executa o procedimento2 e o programa é finalizado. É importante observar que a *funcao2()* não foi executada, embora esteja presente no programa; pois, em momento algum, foi feita uma chamada para ela.

Protótipos de funções

Nos casos onde existem funções que são construídas após o ponto onde são chamadas existe a necessidade de resolver um problemas estrutural. Neste caso, a linguagem C compila o arquivo de cima para baixo. Desta forma, como poderá saber se a forma como a função foi chamada está correta?

Para resolver este problema, pode-se utilizar dois recursos:

1. Construir as funções numa ordem tal que, toda função que for chamada já esteja construída.
2. O caso acima salientado é de difícil construção, principalmente quando se tem um programa com muitas funções, dispostas em vários arquivos. Para este caso utiliza-se o protótipo de função.

O protótipo nada mais é do que a instrução equivalente à linha de construção de uma função (linha do nome da mesma), seguida pelo terminador padrão de C (;). Esse protótipo deve ser escrito fora do corpo das funções e antes do nome da função a ser referenciada. Exemplo:

```
funcao1();      /* Protótipo de funções a serem chamadas */
funcao2();

main()
{
proc1;
funcao1();
proc2;
}

funcao1()
{
proc3;
}

funcao2()
{
proc4;
proc5;
}
```

Neste caso, o compilador não terá dúvidas com relação ao formato das funções, mesmo elas sendo chamadas antes de serem construídas, pois na parte superior do programa, ou cabeçalho do programa", foram escritos os protótipos das funções.

Arquivos de cabeçalho

É devido, principalmente, ao problema de protótipos de funções que foram introduzidos os arquivos de cabeçalho. Nesses arquivos, que possuem a extensão ".H", de preferência, normalmente vêm definidas várias funções que estão disponíveis para o programador, na biblioteca C. O programador também pode construir os seus próprios arquivos de cabeçalho, a fim de definir suas funções. Isto é muito utilizado quando se constrói algumas funções em um arquivo, mas deseja-se utilizá-las em outros. Para evitar escrever os protótipos das funções toda hora, em cada novo arquivo, basta utilizar a diretiva #INCLUDE para carregar o arquivo de cabeçalho onde estão definidas as funções.

4.3 – CONSTRUÇÃO DE FUNÇÕES DE DIVERSOS TIPOS

Funções e parâmetros

Em linguagem C pode-se ter funções de diversos tipos:

- Funções simples: não possuem parâmetros nem retorno.
- Funções que retornam valores.
- Funções que possuem parâmetros.

Os parâmetros são informações que são passadas para a função, e são necessários para que ela realize suas tarefas. Por exemplo: uma função que calcula a raiz quadrada de um número, precisa de pelo menos um parâmetro, que é o número cuja raiz deve ser calculada.

Os valores de retorno podem ser uma simples informação de que a função foi executada com sucesso, ou não; bem como o resultado de algum cálculo ou processamento. Exemplo: O resultado da raiz quadrada do número que foi passado como parâmetro.

Funções que retornam valores

Para se construir funções que retornem valores, deve-se colocar o tipo de dado que esta irá retornar, antes do seu nome; e, antes do final da função, deve-se utilizar a instrução RETURN, seguida do valor (ou variável) a ser retornado. Por padrão, toda função em C retorna "INT". Exemplo:

Protótipo:

```
float função1();
```

Construção:

```
float função1() {
        procedimentos;
        return valor_float;
}
função2() {
        procedimentos2;
        return valor_int;
}
```

Chamada:

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

        b=função1();
        if ( função2() ) c = função1() + 10;
}
```

Observe que a função2() não possui nenhuma informação do tipo de valor que deve ser retornado. O compilador irá considerá-la um função inteira, ou seja, que retorna um valor "INT".

Para se receber um valor retornado por uma função pode-se atribuir a função uma variável; inserir a função em uma expressão aritmética ou lógica; ou ambos.

Funções com parâmetros

Quase todas as funções da biblioteca C e a grande maioria das funções criadas por programadores, possuem parâmetros. Esses parâmetros podem ser de qualquer tipo de dado válido em C, ou podem ser até uma função. Para se construir funções com parâmetros, deve-se colocá-lo dentro do parêntesis, precedido de seu tipo como se estivesse declarando uma variável. Exemplo:

Construção: *double raizquad(double numero) {*
proc_calc_raiz;
return resultado;
}
Protótipo: *double raizquad (double numero); /* ou */*
double raizquad (double)
Exemplo de chamada:

```
main() {
        double x, y, z;
        x = 10000;
        y = raizquad(x);
        z = raizquad(100.);
}
```

Observe que existem duas formas de se construir o protótipo da função. O primeiro, idêntico à construção; o segundo, apenas com o tipo do parâmetro. Isto é possível porque, para o compilador, num protótipo de função, só interessa saber o tipo de informação.

Outro detalhe importante é com relação ao nome do parâmetro. Esse nome, é o de uma variável que só será usada dentro da função. Não existe nenhuma relação entre ele e o nome da variável cujo valor está sendo passado. No exemplo anterior, "x" é uma variável que é utilizada dentro de main(). O seu valor está sendo passado ou transferido para o parâmetro da função raizquad(), que se chama "numero", ou seja, durante esta chamada à função raizquad(), "numero=10000". Observe na segunda chamada à função raizquad(), que não se está passando o valor de uma variável, e sim, um valor constante. Durante esta chamada à função raizquad(), "numero=100".

Funções com vários parâmetros

As funções em C podem ter vários parâmetros, para isso basta separá-los com vírgula, tanto na hora da construção, como na hora da chamada. É importante observar que os parâmetros devem ser passados na mesma seqüência e com o mesmo tipo que foram construídos; caso contrário, poderão ocorrer erros durante a execução. Normalmente, os compiladores C, não informam ou geram mensagens de erro, quando se passa um valor de tipo diferente do parâmetro.

Passagem de parâmetros por valor ou por referência

O processo de passagem de parâmetros mostrado acima é chamado de "passagem de parâmetros por valor", porque se está apenas passando um número, que pode ser uma variável ou uma constante, para outra função.

Muitas vezes, entretanto, deseja-se enviar o valor de uma variável, mas precisa-se que este seja alterado dentro da função. Existem duas formas de se fazer isso:

1) Usando-se a passagem de parâmetros por valor:

```
int quadrado ( int lado )
{
        returna lado*lado;
}
main()
{
        int l=5;

        l = quadrado ( l );
}
```

Neste caso, o valor da variável "l" é passado para o parâmetro "lado", mas a própria variável "l" recebe o resultado da função quadrado().

2) Usando-se passagem de parâmetros por referência:

Neste caso, será utilizado outra variável para receber o resultado, só que, não através da instrução RETURN. Passa-se o "endereço de memória" desta variável, para que a função quadrado() possa alterar o seu valor diretamente.

```
quadrado ( int lado, int *resultado)
{
        *resultado = lado * lado;
}

main()
{
        int l=5, r;

        quadrado ( l, &r);
}
```

Neste caso, o valor de "l" é passado para "lado" (passagem por valor), e o endereço de memória da variável "r" é passado para "resultado" utilizando-se o operador "&" (endereço). Na função quadrado(), existe um operador "*" antes da palavra "resultado" que indica que ela não contém um valor inteiro, e sim o endereço de memória de uma variável inteira (resultado é um ponteiro). Observe que este operador tem que acompanhar a variável, quando ela for utilizada. Na linha "*resultado =", lê-se, "o conteúdo de resultado é igual a". Quando essa instrução for executada, mesmo o processamento ainda estando dentro da função quadrado(), o valor da variável "r", na função main(), já estará alterado.

4.4 – CLASSE DE ARMAZENAMENTO DE FUNÇÕES E VARIÁVEIS

As classe de armazenamento definem em que partes de um programa que uma variável ou função podem ser referenciadas, se elas podem ser utilizadas por outros arquivos, etc. Isto é chamado de escopo de variável ou da função.

Classes de armazenamento de variáveis

Basicamente, uma variável possui dois escopos: LOCAL ou GLOBAL.

Uma variável local é aquela que só pode ser utilizada dentro da função em que foi declarada, porque a sua existência está vinculada ao fato da função estar sendo executada, ou seja, uma variável local só existe enquanto a função na qual ela foi declarada estiver executando. As variáveis locais são alojadas no segmento da pilha (SS), e têm a vantagem de não ocuparem espaço fixo na memória; pois, a pilha é incrementada todas vez que uma função é chamada e é removida toda vez que a função retorna. As variáveis locais são aquelas declaradas dentro do corpo das funções.

Uma variável global é aquela que pode ser utilizada por qualquer função de um programa e o seu valor pode ser lido ou alterado por qualquer uma dessas funções. Ela está sempre presente na memória, pois é alojada no segmento de dados (DS). Tem a vantagem de reduzir o número de variáveis a serem declaradas e a de reduzir o número de parâmetros que devem ser passados para as funções. Em contrapartida, aumentam a necessidade de memória do programa. As variáveis globais são aquelas declaradas fora do corpo das funções e antes de serem referenciadas.

No próximo exemplo teremos duas variáveis globais “a” e “b”, que podem ser utilizadas pelas funções “funcao1()” e “main()”. A funcao1() tem uma variável local chamada de “c”, e que só existirá dentro da mesma, quando essa for executada. A função main() tem duas variáveis locais “a” e “c”, que só existirão enquanto e quando esta função entrar em execução.

Pontos importantes

- Podem existir variáveis locais com o mesmo nome só que em funções diferentes. O importante é saber que são variáveis diferentes, não tem vínculo entre Sistemas de Informações. O fato de terem o mesmo nome é uma coincidência. É o caso da variável “c”.
- Podem existir variáveis locais com o mesmo nome de variáveis globais. Da mesma forma, não têm vínculo entre Sistemas de Informações. Numa função, se existir uma variável local declarada com o mesmo nome da global, o compilador utilizará a variável local. É o caso da variável “a”.
- Qualquer outra variável que não tenha sido declarada internamente em uma função (local), deverá ter sido declarada como global, senão, o compilador irá gerar uma mensagem de erro: “Variável não declarada”.

Exemplo:

```
int a = 5, b;

funcao1() {
int c=1;          /* variável local */

a = 10;           /* modificando o valor da variável global */
procedimentos;
}

main() {
float a;
int c;

a = 15;           /* variável local */
b = 7;            /* variável global */

procedimentos;
funcao1();
}
```

Modificador STATIC

O modificador STATIC, que deve ser escrito antes do tipo, durante a declaração de uma variável, permite que o valor de uma variável local, se mantenha, mesmo quando a função terminar; de forma que, quando a função for chamada novamente, o valor da variável será aquele que ela tinha, quando do término da primeira chamada àfunção. Exemplo:

```
funcao1() {
        static int c = 1;
        procedimentos;
        c = 2;
}
```

Na primeira chamada à funcao1(), "c" iniciará com 1 e depois terá seu valor alterado para 2. Na segunda chamada à funcao1(), "c" iniciará com 2, que era o seu valor anterior, ou seja, seu valor permanecerá inalterado (estático). Variáveis estáticas são alojadas no segmento de dados (DS).

Para uma variável global, o efeito do modificador STATIC é diferente e só tem sentido em programas formados por mais de um arquivo fonte. Uma variável global estática só estará disponível para as funções que estiverem no mesmo arquivo de sua declaração.

Classes de armazenamento de funções

O escopo de uma função também só tem sentido quando se trabalha com programas formados por mais de um arquivo fonte. Neste caso usa-se o modificador STATIC, antes da construção da função, para indicar que aquela função só deve ser utilizada dentro do arquivo em que foi construída. Caso contrário, ela poderá ser referenciada ou utilizada em todos os arquivos que formam o programa.

Funções que serão utilizadas por outros programas, por exemplo, funções de biblioteca, têm que ter escopo global.

Pode-se ter, então, funções diferentes com o mesmo nome, dentro de um programa, desde que pelo menos uma delas tenha escopo local (STATIC).

4.5 – PROGRAMAS COM MÚLTIPLOS ARQUIVOS

Projeto de programas

Normalmente, programas com mais de 1000 linhas não se tornam agradáveis para serem manipulados em apenas um arquivo. Além do mais, a organização das funções fica prejudicada. Por isso, cria-se um projeto de programa em que cada grupo de funções é colocado em um arquivo. Esses arquivos deverão ser compilados separadamente e depois "link-editados" juntamente com a biblioteca da linguagem C.

Uso de múltiplas funções e variáveis globais

Como um programa normalmente possui muitas funções e variáveis globais, e nunca será possível separá-lo totalmente, de forma que, as funções e as variáveis globais em um arquivo não sejam utilizadas pelos demais, será necessário indicar ao compilador que aquelas funções ou variáveis já estão declaradas, só que em outro arquivo. Além disso, é importante indicar ao compilador o tipo de variável e o protótipo da função, para que o programa código gerado seja correto. Isto é feito utilizando-se o modificador EXTERN, que indica que a variável ou função é externa. Exemplo:

Arquivo1:

```
extern int a;

float funcao1() {
        imprime_a;
}
```

Arquivo2:

```
extern float funcao1();

main() {
        funcao1();
}
```

É importante observar que não podem existir duas variáveis globais ou funções, não estáticas, com o mesmo nome.

Compilação e Link-Edição

O problema do uso de vários arquivos está no fato da forma como o compilador C e o Link-Editor trabalham.

O compilador C é um compilador de arquivos, ou seja, ele compila cada arquivo de uma vez, independentemente um do outro. Para cada arquivo compilado, ele gera um arquivo objeto (.OBJ). Desta forma, ele tem que ser capaz de compilaras funções do arquivo fonte apenas com as informações contidas no mesmo (daí a necessidade de se usar a diretiva EXTERN e os arquivos de cabeçalho, para definir tudo o que não foi criado dentro do arquivo).

Nos arquivos objeto, além do código compilado, são colocadas todas as informações referentes as funções e variáveis globais existentes naquele arquivo, bem como as funções e variáveis globais que estão faltando naquele arquivo.

O Link-Editor reúne um ou mais arquivos objeto e cruza as referências, ou seja, faz a ligação entre as variáveis e funções que estão faltando em alguns arquivos, com aquelas que foram criadas nos outros. As informações que ficarem faltando serão buscadas na biblioteca da linguagem C que estiver sendo utilizada. Se todas as ligações forem feitas, é criado um arquivo executável, caso contrário, serão geradas mensagens de erro.

Dicas finais

- Use, sempre que possível, variáveis locais;
- Coloque todas as informações de que uma função precisa sob a forma de parâmetros, em lugar de utilizar variáveis globais, para que esta função, por si só, possa ser utilizada em vários pontos do programa ou em vários programas;
- Procure fazer funções o mais genéricas possível, de modo que, uma única função possa ser várias vezes utilizada em um programa, reduzindo o tamanho do programa e facilitando a sua manutenção.
- Utilize variáveis globais estáticas para um conjunto de funções que realizam algum trabalho em comum e que possuam informações em comum, dentro de um mesmo arquivo.

5 – TIPOS AVANÇADOS DE DADOS EM C

5.1 – VETORES E STRINGS

Um vetor é formado por um conjunto de dados de um mesmo tipo determinado, referenciada pelo mesmo nome, que são alojadas seqüencialmente na memória. Pode-se entender um vetor como uma variável que tem sua área subdividida em áreas menores para guardar diversos dados diferentes. Para acessar cada item do vetor utiliza-se um número que indica o índice de sua posição. Sintaxe:

tipo nomevar1 [n.elem1], nomevar2 [n. elem2];

Exemplos:

int notas[4], cód[10];
char nome[10];
float media[5];

A diferença entre a declaração de uma variável simples para um vetor é que, após o nome da variável, escreve-se o número de elementos entre colchetes. Nos casos acima, a variável *notas* possui 4 elementos do tipo int, enquanto que a variável *cód* possui 10 elementos int. Para que se possa acessar cada elemento do vetor, escreve-se o nome da variável seguido do índice, entre colchetes. Exemplo:

```
notas[1] = 5;
notas[2] = 8;
notas[3] = notas[1] * 1.2;
```

A numeração dos índices, ou indexação, de um vetor, inicia-se sempre de 0 (zero), ou seja, o primeiro elemento é o de índice zero (0) e o último é o de índice "tamanho-1". Assim, para o vetor notas o primeiro elemento seria notas[0] e o último seria notas[3].

STRING

Uma string nada mais é do que um vetor do tipo char (vetor de caracteres), que armazena, normalmente uma mensagem ou frase. Além dessa particularidade, o último elemento da mensagem deve ser sempre um barra zero(\0), não confundir com o caractere zero (0) cujo código ASCII é 48. É importante salientar que o último elemento da mensagem não precisa ser, necessariamente, o último elemento do vetor char. Exemplos:

```
char nome[11];
char título[21] = {"Este é o título!"};
```

Acima pode-se destacar que a string "nome" poderá armazenar até 10 caracteres, pois um deve ser reservado para o terminador \0. A string "titulo" além de ser declarada, está sendo inicializada. Observe que o tamanho da mensagem é de 16 caracteres e a variável suporta até 20 (mais o terminador). Neste caso, o terminado é colocado após o caractere "!", posição [15], e não no final da string", índice [20].

Outra particularidade, muito importante, no uso de strings é que pode-se utiliza-las como uma única variável, sem precisar acessar todos os elementos.

5.2 – MATRIZES

As matrizes são estruturas de armazenamento de dados em memória semelhantes aos vetores, só que possuem mais do que uma dimensão. Sua declaração e utilização é semelhante ao caso dos vetores, com a diferença que deve-se utilizar dois ou mais índices, um para cada dimensão. Sintaxe:

tipo nomevar [dimens1] [dimens2] .... ;

Exemplos:

```
int A[10][10];  /* declarada uma matriz A com 10 linhas e 10 colunas */
double coord [100] [100] [100]  /* matriz coord para armazenar coordenadas 3D */
char nomes [10] [21]; /* vetor de strings */

x[0][0][0] = 1;
x[99] [99] [99] = 100;
nomes[1] [20] = 'a';
```

Para acessarmos algum elemento de uma matriz devemos indicar todos os índices do elemento desejado. Pode-se ter matrizes com várias dimensões; entretanto, não é comum o seu uso, num aconselhável, exceder 3 dimensões. No caso de uma matriz de elementos "char" ser tratada como um vetor de "strings", utiliza-se apenas um índice para referenciar a "string", e dois índices para referenciar um elemento da "string". Por exemplo, nomes[1][20] é um elemento e nome[1] é toda a "string".

5.3 – ESTRUTURAS

Estruturas são conjunto de variáveis diferentes, que podem ser do mesmo tipo ou não, referenciadas por um único nome. Seus elementos podem ser variáveis simples, vetores, matrizes ou até mesmo outras estruturas. No caso da estruturas, pode-se defini-las sem estar declarando nenhuma variável. É comum usar nomes com letras maiúsculas para identificar o nome da estrutura. O tamanho de um estrutura será a soma do tamanho de cada um dos seus elementos.

As estruturas podem ser utilizadas para trabalhos com bancos de dados em forma de registros. Os quais podem ser armazenados em memórias auxiliares (unidades de discos). Cada registro é constituído por uma gama de informações pertinentes à natureza do dado representado (campos).

Sintaxe da definição de uma estrutura:

```
struct NOMESTRU {
        tipo nomevar1;
        tipo nomevar2;
        tipo nomevar3;
};
```

Usando uma estrutura definida para declarar uma variável :

```
struct NOMESTRU nomevar;
```

obs: Não se esqueça do ponto e vírgula após a chave final da estrutura.

Exemplo:

```
struct COORDENADAS {
        float x;
        float y;
        float z;
};

struct COORDENADAS a, c[100];
```

No exemplo acima, *x, y* e *z* são os elementos de uma estrutura chamada COORDENADAS. Declarou-se, em seguida, duas variáveis *a* e *c*, com tipo *struct* COORDENADAS, sendo que *c* é um vetor.

Para acessar um elemento de uma estrutura, escreve-se o nome da variável seguida pelo nome do elemento, separados por um ponto (.). Exemplo:

```
a.x = 0;
a.y = 1;
a.z = 2;
c[6].x = 4;
c[14].z = c[14].x + c[14].y;
```

O exemplo acima da variável *c*, mostra que fica mais fácil e organizado trabalhar com um vetor de estruturas que possa representar as coordenadas, do que ter uma matriz tridimensional (x,y,z), como um ponto só pode ser definido com as três coordenadas é mais simples armazenar em conjunto. Outro exemplo típico é o de se criar um registro com os dados pessoais de clientes, por exemplo. Ao invés de se declarar uma variável para cada dado, deve-se criar uma estrutura com todas as informações pertinentes ao cliente. Isso facilitará a estruturação e a implementação do programa. Exemplo:

```
struct CLIENTES {
        char nome[31];
        char ender[41];
        char telef[10];
];

struct CLIENTES c[300]; /* limitação em até 300 clientes */
```

A declaração da variável *c* pode ser feita diretamente após a definição da estrutura CLIENTES, exemplo:

```
struct CLIENTES { ..... } c[300];
```

Apesar disto, esta forma não é recomendada, pois é costume colocar as definições das estruturas em arquivos de cabeçalho, da mesma forma que as declarações das variáveis devem ser feitas nos respectivos arquivos fonte.

5.4 – UNIÕES

As uniões são semelhantes as estruturas, mas os seus elementos, ao invés de ocuparem a sua posição de memória, um ao lado do outro, ficam sobrepostos entre si, ou seja, existem várias variáveis utilizando o mesmo espaço de memória. A utilização de uniões deve ser cuidadosa, pois tem-se que ter um controle sobre quais informações que estão sendo utilizadas no momento. O acesso aos elementos de uma união é feito de forma análoga ao de uma estrutura. O tamanho de uma união será o tamanho do seu maior elemento. Sintaxe:

```
union NOMEUNIAO {
        tipo nomevar1;
        tipo nomevar2;
};
union NOMEUNIAO nomevar;
```

Exemplo:

```
union REGISTER {
        int ax;
        cha al, ah;
};
union REGISTER reg;

reg.ax = 0x1020;  /* ou */

reg.al = 0x20;
reg.ah = 0x10;
```

No caso demonstrado acima, declarou-se uma união para representar o conteúdo do registrador AX, do microprocessador. Como já foi citado, o registrador pode ser dividido em duas partes de 8 bits cada, chamadas AL e AH. Como a alteração das partes, AL e AH, implica na alteração do todo, AX, eles devem estar sobrepostos na memória. Consegue-se isso através de uma união.

5.5 – ENUMERAÇÕES

Enumerações são conjuntos de constantes do tipo *int* e são utilizadas para facilitar declarações de várias constantes referentes àum mesmo assunto. Sintaxe:

enum NOMEENUM = { const1 = v1; const2 = v2; ..... };

Cada constante declarada terá um valor, que poderá ser definido usando-se o operador de atribuição (=), ou não; neste caso, o próprio compilador irá dar um valor para a constante, que será o valor da constante anterior, mais um.

Exemplo:

enum VIDEO = { MONO=0, CGA, EGA, VGA=5, SVGA};

Neste caso, os valores das constantes serão MONO=1, CGA=2, EGA=3, VGA=5, SVGA=6.

6 – TRABALHANDO COM PONTEIROS (APONTADORES)

Ponteiros são variáveis que contém o endereço de memória de uma outra variável (ou de uma função), que pode ser uma variável simples, um vetor, uma matriz ou uma estrutura. São utilizados quando se precisa fazer passagem de parâmetros por referência, quando deseja-se usar a técnica de alocação dinâmica de memória, criação das estruturas de dados de listas, pilhas, filas e árvores, entre outras aplicações.

A principal vantagem do seu uso está na agilidade imposta ao programa e as strings e matrizes são declaradas de formas mais simples.

A declaração de um ponteiro é feita colocando-se um asterisco (*) antes do seu nome, durante a declaração. Sintaxe:

*tipo *nomeponteiro;*

Exemplo:

```
int *a;
float c, *d;
```

Nos casos acima tem-se: *a* como um ponteiro para uma variável *int*, enquanto que *d* é um ponteiro para uma variável *float*.

6.1 – VETORES E MATRIZES COMO PONTEIROS

Os nomes dos vetores e das matrizes jão são tratados como ponteiros, pelo compilador C. Na verdade, eles contém o endereço da posição de memória do primeiro elemento do vetor ou da matriz, que se confunde com o endereço inicial do próprio vetor ou matriz.

Desta forma, em uma função que possui um vetor como parâmetro, mas com um tamanho desconhecido durante a programação, pode-se declarar um ponteiro ao invés do vetor. Exemplo:

```
/* função que calcula a somatória dos elementos do vetor */
float soma (int numero, float *dados) {
        int i;
        float s= 0;
```

```
        for (i=0; i<numero; i++)
                s+= dados[ i ];
        return s;
}
```

A função acima deixa claro que mesmo tendo declarado dados como um ponteiro, o acesso aos elementos pode ser feitos como em um vetor.

6.2 – PONTEIROS DE ESTRUTURAS

Os ponteiros para estrutura possuem uma particularidade. Para acessarmos os seus elementos, devemos trocar o ponto (.) pelo sinal traço sinal de maior (->). Exemplo:

```
int entra_cliente (struct CLIENTE *c) {
        ler_dado (c-> nome);
        ler_dado (c->ender);
}
```

6.3 – ARITMÉTICA DE PONTEIROS

Os ponteiros, por conterem endereços de memória, podem ser manipulados aritmeticamente, ou seja, podem ser incrementados, decrementados, multiplicados, etc. Os operadores "++" e "--" podem ser utilizados em ponteiros. É importante observar que, somar um ponteiro de 1, não significa acrescentar um 1 byte, mas sim, passar para a próxima posição ou para o próximo elemento. Desta forma, quando incrementa-se um ponteiro inteiro, ele será somado de 2 bytes, que é o tamanho de um inteiro. No caso de um ponteiro de dupla precisão (double), ele será somada de 8 bytes. Exemplo:

```
int *a;
double *b;

a--;  /* -1 "int" corresponde 2 bytes */
b = b + 10;  /* 10 "doubles" corresponde a 8 * 10 bytes */
```

7 – TRABALHANDO COM ARQUIVOS:

Um arquivo armazena informações em algum tipo de memória auxiliar, representada nos computadores pelas unidades de discos (Rígidos ou Winchester e Disquetes). Existem duas formas de manipulação de arquivos:

- Seqüenciais (Texto) – possui uma marca EOF (final de arquivo);
- Binários – não possuem marca EOF.

Para se trabalhar com um arquivo é necessário abri-lo criando um canal de comunicação entre o programa sendo executado na memória e o arquivo em disco. A função de biblioteca *fopen* desempenha o papel de abertura e retorna um ponteiro do canal de comunicação utilizado para o tráfego memória-disco.

Sintaxe de declaração de ponteiro de arquivo:

*FILE *nomept;*

Para a abertura do arquivo utiliza-se a sintaxe:

nomept = fopen("nomearq.ext", "tpabert");

O elemento *tpabert* representa o tipo de abertura que será definido no arquivo, conforme a tabela abaixo:

"r" - Abrir o arquivo texto para leitura. O arquivo deve estar presente no disco.
"w" - Abrir o arquivo texto para gravação. Se existir será substituído, senão será criado.
"a" - Abrir o arquivo texto para acréscimo. Se existir, os dados serão acrescentados no fim.
"r+" - Abrir o arquivo texto para gravação e leitura. Se existir, os dados serão atualizados.
"w+" - Abrir o arquivo texto para gravação e leitura. Se existir, será reinicializado.
"a+" - Abrir o arquivo texto para atualizações e inserções no fim do arquivo.

"rb" - Abrir arquivo binário para leitura. O arquivo deve estar presente no disco.
"wb" - Abrir arquivo binário para gravação. Se existir será substituído, senão será criado.
"ab" - Abrir arquivo binário para acréscimo. Se existir, os dados serão acrescentados no fim.
"rb+" - Abrir o arquivo binário para gravação e leitura. Se existir, os dados serão atualizados.
"wb+" - Abrir o arquivo binário para gravação e leitura. Se existir, será reinicializado.
"ab+" - Abrir o arquivo binário para atualizações e inserções no fim do arquivo.

Caso o arquivo tente ser aberto mas retorne um valor NULL para o ponteiro de arquivo significa que ocorreu algum erro, como por exemplo, arquivo inexistente. Para tratar este tipo de comportamento utiliza-se:

```
/* Exemplo de abertura, teste de êxito e fechamento de arquivo */
void main() {
        FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */

        if ((arquivo = fopen("arqteste.txt", "w")) == NULL) {
                printf ("\nErro na abertura do arquivo, verifique!");
                exit(0);
        }
        fclose(arquivo); /* fechamento do arquivo */
}
```

7.1 – EXEMPLO DE LEITURA DE CARACTERE DE ARQUIVO:

```
void main() {
        FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */
        char letra;

        if ((arquivo = fopen("arqteste.txt", "r")) == NULL) {
                printf ("\nErro na abertura do arquivo, verifique!");
                exit(0);
        }
        while (letra = getc(arquivo)) != 'EOF') /* enquanto não fim-de-arquivo lê e imprime na tela */
                printf ("\nO caracter lido é %c",letra);
        fclose(arquivo); /* fechamento do arquivo */
}
```

7.2 – EXEMPLO DE GRAVAÇÃO DE CARACTERE NO ARQUIVO:

```
void main() {
	FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */
	char letra;

	if ((arquivo = fopen("arqteste.txt", "w")) == NULL) {
		printf ("\nErro na abertura do arquivo, verifique!");
		exit(0);
	}
	while (letra = getche()) != '\r') /* enquanto não for pressionado ENTER imprime no arquivo */
		putc(letra, arquivo);
	fclose(arquivo); /* fechamento do arquivo */
}
```

7.3 – EXEMPLO DE LEITURA DE STRING DE CARACTERES DO ARQUIVO:

```
void main() {
	FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */
	char frase[81];

	if ((arquivo = fopen("arqteste.txt", "r")) == NULL) {
		printf ("\nErro na abertura do arquivo, verifique!");
		exit(0);
	}
	while (fgets(frase, 80, arquivo)) != NULL) /* enquanto existir frase no arquivo imprime na tela */
		puts(frase);
	fclose(arquivo); /* fechamento do arquivo */
}
```

7.4 – EXEMPLO DE GRAVAÇÃO DE STRING EM ARQUIVO:

```
void main() {
	FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */
	char frase[81];

	if ((arquivo = fopen("arqteste.txt", "w")) == NULL) {
		printf ("\nErro na abertura do arquivo, verifique!");
		exit(0);
	}
	while (strlen(gets(frase))) > 0) {  /* enquanto string maior do que comprimento zero */
		fputs(frase, arquivo);
		fputs("\n", arquivo);
	}
	fclose(arquivo); /* fechamento do arquivo */
}
```

7.5 – EXEMPLO DE LEITURA DE ARQUIVO FORMATADA:

```
void main() {
        FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */
        char nome[51];
        int codigo;
        double salário;

        if ((arquivo = fopen("arqteste.txt", "r")) == NULL) {
                printf ("\nErro na abertura do arquivo, verifique!");
                exit(0);
        }
        while (fscanf(arquivo, "%s %d %f", &nome, &codigo, &salario) != 'EOF')
                printf ("\n %s %d %f", nome, codigo, salario); /* imprime na tela o que foi lido */
        fclose(arquivo); /* fechamento do arquivo */
}
```

7.6 – EXEMPLO DE GRAVAÇÃO FORMATADA EM ARQUIVO:

```
void main() {
        FILE *arquivo;  /* arquivo é um ponteiro para um arquivo */
        char nome[51];
        int codigo;
        double salário;

        if ((arquivo = fopen("arqteste.txt", "w")) == NULL) {
                printf ("\nErro na abertura do arquivo, verifique!");
                exit(0);
        }
        do {
                printf ("\n Digite o codigo, o nome e o salário do cliente: ");
                scanf ("%s %s %f", &nome, &codigo, &salário);
                fprintf (arquivo, "%d %s %f", codigo, nome, salário);
        } while (strlen(nome) >1);
        fclose(arquivo); /* fechamento do arquivo */
}
```

8 – APÊNDICE (BIBLIOTECA DE FUNÇÕES)

8.1 – FUNÇÕES DE ENTRADA E SAÍDA PADRÃO (STDIO.H):

clearerr (arqpt)

Esta função coloca em zero (desligado) o indicador de erro para o arquivo do ponteiro definido. O indicador de EOF também é restituído.

fclose (arqpt)

Fecha o arquivo definido pelo ponteiro e esvazia seu buffer. Após o uso desta função o ponteiro não está mais associado ao arquivo e quaisquer buffers automaticamente alocados são liberados. Se a função for bem sucedida devolve zero, caso contrário, devolve um número diferente de zero.

feof (arqpt)

Esta função verifica o indicador de posição de arquivo para determinar se foi atingido o final do arquivo associado ao ponteiro definido. Um valor diferente de zero é devolvido se o indicador de posição de arquivo está no final do mesmo, caso contrário devolve zero.

ferror (arqpt)

Esta função verifica a ocorrência de erros em um dado arquivo definido pelo ponteiro. Caso retorne zero indica que nenhum erro ocorreu, caso contrário algum erro ocorreu.

fflush (arqpt)

Se o ponteiro está associado a um arquivo aberto para gravação, a função promove a descarga do buffer de saída para o arquivo em questão. Caso o ponteiro esteja associado a um arquivo de leitura, o conteúdo do buffer de entrada é esvaziado. Nos dois casos, o arquivo continua aberto. Um valor de retorno indica sucesso ou é retornado EOF se ocorrer um erro.

fgetc (arqpt)

Esta função devolve o próximo caractere do arquivo de entrada associado ao ponteiro e incrementa o indicador de posição de arquivo. O caractere lido é um unsigned char que é convertido em inteiro. Se o final do arquivo for alcançado a função retorna EOF, mas deve-se utilizar a função **feof** para verificar o final do arquivo quando trabalhar com arquivos binários, pois EOF é um valor inteiro válido. Se a função encontrar um erro também retorna EOF.

fgetpos (arqpt, *fpost_t* *posicao)

Esta função armazena o valor atual do indicador de posição do arquivo definido pelo ponteiro no ponteiro posicao. O objeto apontado para posicao deve ser do tipo ***fpost_t***, que é um tipo definido na biblioteca ***stdio.h***. O valor armazenado é útil apenas em uma chamada subseqüente da função **fsetpos()**. Caso ocorra algum erro a função retorna um valor diferente de zero, senão retorna zero.

fgets (linha, num, arqpt)

Esta função lê (**num**-1) caracteres do arquivo definido pelo ponteiro e coloca-os no vetor de caracteres linha. Os caracteres são lidos até que uma nova linha, ou um EOF, seja encontrado ou até que o limite especificado seja atingido. Após os caracteres serem lidos, um nulo é colocado na matriz imediatamente após o último caractere. O caractere de nova linha é mantido e é parte da linha. Caso seja bem sucedida a função devolve a linha e um ponteiro nulo é devolvido quando ocorre alguma falha. Se ocorrer um erro de leitura, o conteúdo da matriz apontada por linha é indeterminado.

fopen (nomearq, modo)

Esta função abre um arquivo no modo especificado e devolve um ponteiro associado a ele. Os tipos de operações permitidas nos arquivos são definidos pelo modo. Os valores legais para o modo, como especificado pelo padrão ANSI, são mostrados no capítulo 7. O nome do arquivo deve ser uma string de caracteres que constitua um nome válido de arquivo, como definido pelo sistema operacional, e pode incluir uma especificação de percurso (caminho ou path), se o ambiente suporta.

Se a função consegue abrir corretamente o arquivo ela retorna um ponteiro, caso contrário é devolvido um ponteiro nulo (NULL).

fprintf (arqpt, formato)

Esta função escreve no arquivo especificado utilizando a formatação de conteúdo de formato. O valor de retorno é o número de caracteres realmente escritos. Se ocorre um erro, um número negativo é devolvido. Pode haver de zero a vários argumentos – o número máximo depende do compilador e do sistema operacional.

fputc (letra, arqpt)

Esta função escreve o caractere letra no arquivo especificado pelo ponteiro na posição atual do arquivo e avança o indicador de posição de arquivo. Embora letra seja declarada como inteiro por razões históricas ele é convertido para unsigned char. Como todos os argumentos de caracteres são elevados a inteiros no momento da chamada, variáveis tipo caractere geralmente são usadas como argumentos. Se um inteiro fosse usado, o byte mais significativo seria simplesmente ignorado. O valor devolvido pela função é o do caractere escrito, se ocorrer um erro ela retorna EOF.

fputs (string, arqpt)

Esta função escreve no arquivo definido pelo ponteiro o conteúdo da string. O terminador nulo não é escrito. Ela devolve um valor não negativo em caso de sucesso e EOF em caso de falha.

fread (buf, tam, num, arqpt)

Esta função lê ***num*** objetos, cada um com tamanho ***tam*** bytes de comprimento do arquivo definido pelo ponteiro e coloca-os na matriz apontada por ***buf***. O indicador de posição de arquivo é avançado pelo número de caracteres lidos. Ela devolve o número de itens realmente lido. Caso sejam lidos menos itens do que o solicitado na chamada, isso significa que ocorreu um erro ou que o final de arquivo foi atingido.

fscanf (arqpt, formato)

Esta função opera exatamente como o **scanf**, mas lê a informação do arquivo definido pelo ponteiro ao invés da entrada padrão (stdin). Ela devolve o número de argumentos que realmente receberam valores. Esse número não inclui os campos ignorados. Um valor de retorno EOF significa que ocorreu uma falha antes que a primeira atribuição tenha sido feita.

fseek (arqpt, passo, inicio)

Esta função coloca o indicador de posição de arquivo associado ao ponteiro de acordo com os valores de inicio e passo. Ela suporta operações de Entrada e Saída aleatórias. O passo é o número de bytes a partir do inicio até chegar a nova posição.

Um valor de retorno zero indica que a função completou com sucesso, senão ocorreu falha.

fsetpos (arqpt, fpost_t posicao)

Esta função move o indicador de posição de arquivo vinculado ao ponteiro definido para o ponto especificado pelo objeto apontado por posicao. Esse valor deve ser previamente obtido através de uma chamada a **fgetpos**. O tipo **fpost_t** é definido pela biblioteca **stdio.h**. Após o uso desta função o indicador de fim de arquivo é desligado e qualquer chamada à função **ungetc()** se torna nula.

ftell (arqpt)

Esta função devolve o valor atual do indicador de posição de arquivo do ponteiro definido. Para arquivos binários o valor é o número de bytes cujo indicador está a partir do início do arquivo. Para arquivos texto o valor de retorno pode não ser significativo, exceto como um argumento de fseek, devido às possíveis traduções de caracteres.

fwrite (buf, tam, num, arqpt)

Esta função escreve **num** objetos no arquivo apontado pelo ponteiro **arqpt** que estão armazenados na matriz de caracteres apontada por **buf**. O indicador de posição de arquivo é avançado pelo número de caracteres escritos. Ela devolve o número de itens realmente escritos, que será igual ao número solicitado, caso a função seja bem sucedida. Se forem escritos menos itens que os solicitados, ocorreu algum erro. Para arquivos texto, diversas tabulações de caracteres podem ocorrer, mas não afetarão o valor devolvido.

getc (arqpt)

Esta função devolve o próximo caractere do arquivo definido pelo ponteiro **arqpt** de entrada a partir da posição atual e incrementa o indicador de posição de arquivo. O caractere é lido como um unsigned char e é convertido para inteiro.

Se o final do arquivo for encontrado ela retorna EOF. Porém como EOF é um valor inteiro válido, deve-se usar a função **feof()** para verificar o final de arquivo quando trabalhar com arquivos binários. Se a função encontra um erro também devolve EOF. Quando se trabalha com arquivos binários, deve-se usar a função **ferror()** para verificar erros em arquivos.

printf ("string de controle", argumentos)

Esta função imprime na tela (saída padrão – stdout) a string de controle substituindo os formatadores pelos argumentos definidos. Ela retorna o número de caracteres realmente escritos. Um valor negativo indica erro.

putc (letra, arqpt)

Esta função escreve a letra no arquivo definido pelo ponteiro. Ela devolve EOF se ocorrer algum erro. Se o arquivo foi aberto no modo binário, EOF é um valor válido para um caractere, desta forma, é importante utilizar a função **ferror()** para determinar se ocorreu algum erro.

putch (letra)

Esta função escreve na saída padrão o caractere contido em letra. Em muitas implementações, a sua saída não pode ser redirecionada. Além disso, em alguns ambientes, ela pode operar relativamente a uma janela em lugar da tela.

puts (string)

Esta função escreve ao vetor de caracteres apontada por string no dispositivo de saída padrão. O terminador nulo é traduzido para uma nova linha (\n). Ela devolve um valor negativo, se bem sucedida e EOF caso contrário.

rewind (arqpt)

Esta função move o indicador de posição do arquivo para o início do mesmo. Ela também limpa os indicadores de erro e de final de arquivo associados ao ponteiro. Não há valor de retorno.

scanf ("string de controle", argumento)

Esta função é uma rotina de entrada de dados de uso geral que lê os argumentos conforme a formatação imposta na string de controle. É o complemento do **printf()**.

8.2 – FUNÇÕES DE TESTES DE TIPOS (CTYPE.H)

isalnum (caractere)

Esta função retorna um valor diferente de zero se o caractere for uma letra ou um dígito. Se não for alfanumérico ela retorna zero.

isalpha (caractere)

Esta função devolve um valor diferente de zero se o caractere for uma letra do alfabeto, caso contrário, devolve zero. O que constitui uma letra pode varia de idioma para idioma.

iscntrl (caractere)

Esta função devolve um valor diferente de zero se o caractere está entre 0 e 0x1F ou é igual a 0x7F (tecla DELETE); caso contrário, devolve zero.

isdigit (caractere)

Esta função devolve um valor diferente de zero se o argumento for um dígito, ou seja, algarismos de 0 a 9. Caso contrário, retorna zero.

isgraph (caractere)

Esta função devolve um valor diferente de zero se o caractere pode ser impresso, com exceção do espaço. Caso contrário devolve zero.

islower (caractere)

Esta função retorna um valor diferente de zero se o caractere é uma letra minúscula, caso contrário, retorna zero.

isprint (caractere)

Esta função retorna um valor diferente de zero se o caractere pode ser impresso, incluindo o espaço; caso contrário retorna zero.

ispuct (caractere)

Esta função devolve um valor diferente de zero se o caractere é um símbolo especial, ou seja, ele inclui todos os caracteres que podem ser impressos e não sejam alfanuméricos num espaço. Caso contrário retorna zero.

isspace (caractere)

Retorna um valor diferente de zero ser o caractere é um espaço, tabulação horizontal, tabulação vertical, alimentação de formulário, retorno de carro ou caractere de nova linha; caso contrário retorna zero.

isupper (caractere)

Esta função devolve um valor diferente de zero se o caractere for uma letra maiúscula, caso contrário retorna zero.

isxdigit (caractere)

Esta função devolve um valor diferente de zero se o caractere é um dígito hexadecimal; caso contrário devolve zero. Um dígito hexadecimal está na faixa de “a” a “f”, de “A” a “F” ou de 0 a 9.

tolower (caractere)

Converte o caractere para minúscula.

toupper (caractere)

Converte o caractere para maiúscula.

8.3 – FUNÇÕES DE CADEIAS DE CARACTERES (STRING.H)

strcat (frase1, frase2)
Esta função concatena uma cópia de frase2 no final de frase1 e termina a frase1 com um nulo (\0). O terminador nulo, que originalmente finalizava a frase1, é sobreposto pelo primeiro caractere de frase2. A frase2 permanece inalterada na operação. Se os vetores de caracteres se sobrepõem, o comportamento de strcat() é indefinido. Ela retorna a frase1 final.

strchr (frase, letra)
Esta função retorna um ponteiro para a posição da primeira ocorrência de letra em frase. Se não for encontrada nenhuma coincidência, será devolvido um ponteiro nulo (NULL).

strcmp (frase1, frase2)
Esta função compara lexicograficamente duas strings e devolve um inteiro baseado no resultado, conforme definido abaixo:

Valor	Significado
Menor do que zero	frase1 é menor que frase2.
Zero	frase1 é igual a frase2.
Maior do que zero	frase1 é maior a frase2.

strcpy (frase1, frase2)
Esta função copia o conteúdo de frase2 em frase1. frase2 deve ser um ponteiro para uma string terminada com um nulo. A função devolve um ponteiro para a frase1. Se frase1 e frase2 se sobrepõem, o comprimento de strcpy() é indefinido.

strerror (num)
Esta função devolve um ponteiro para uma string definida pela implementação, que é associada ao valor de num. Sob nenhuma circunstância deve-se modificar a string.

strlen (frase)
Devolve o comprimento da string frase terminada por um nulo. O nulo não é contado.

8.4 – FUNÇÕES MATEMÁTICAS (MATH.H)

acos (numero)
Esta função retorna o arco-cosseno do número (entre 0 e 1) em radianos.

asin (numero)
Esta função retorna o arco-seno do número (entre 0 e 1) em radianos.

atan (numero)
Esta função retorna o arco-tangente do número em radianos.

atan2 (num1, num2)
Esta função retorna o arco-tangente de num1/num2 em radianos.

ceil (num)
Esta função retorna o menor número inteiro, não menor que num, como doublé.

cos (ângulo)
Esta função retorna o valor do cosseno do ângulo. O ângulo é definido em radianos.

cosh (ângulo)
Esta função retorna o cosseno hiperbólico do ângulo. O ângulo é definido em radianos.

exp (numero)
Esta função retorna o valor do número de Neper elevado ao numero (e^{num}).

fabs (numero)
Esta função retorna o valor absoluto do número.

floor (num)
Esta função retorna o maior número inteiro, não maior que num, como doublé.

fmod (num1, num2)
Esta função retorna resto da divisão de num1 por num2 (Resto de num1/num2).

ldesp (num1, num2)
Esta função retorna o valor da expressão num1*2^{num2}.

log (numero)
Esta função devolve o logaritmo natural (ln) do número. Ocorrerá um erro de domínio se o número for negativo e um erro de escala se ele for zero.

log10 (numero)
Esta função retorna o logaritmo na base 10 do numero. Tem os mesmos erros da função log.

modf (numero, retorno)
Esta função decompõe o numero nas suas partes inteira e fracionária. Ela devolve a parte fracionária e coloca a parte inteira na variável apontada por retorno.

pow (base, exp)
Esta função retorna o resultado da exponenciação da base pelo exp. Ocorrerá um erro de domínio se a base for zero e se o exp for menor ou igual a zero. Também ocorre um erro de domínio se a base for negativa e exp não for um número inteiro. Um estouro produzirá um erro de escala.

sqrt (numero)
Esta função retorna a raiz quadrada de um número. Ocorrerá um erro de domínio se o argumento for negativo.

sin (ângulo)
Esta função retorna o seno do ângulo. O ângulo é definido em radianos.

sinh (ângulo)
Esta função retorna o seno hiperbólico do ângulo. O ângulo é definido em radianos.

tan (ângulo)
Esta função retorna a tangente do ângulo. O ângulo é definido em radianos.

tanh (ângulo)
Esta função retorna a tangente hiperbólica do ângulo. O ângulo é definido em radianos.

8.5 – FUNÇÕES UTILITÁRIAS (STDLIB.H)

atof (string)
Converte a string para double.

atoi (string)
Converte a string para integer.

atol (string)
Converte a string para long integer.

rand (void)
Retorna um número inteiro pseudo-randômico na faixa de 0 a RAND_MAX, que é pelo menos 32767.

free (ptr)
Esta função retorna ao Heap a memória apontada por ptr, tornando a memória disponível para alocação futura.

calloc (numobj, tam)
Retorna um apontador para o espaço de um vetor de *numobj* objetos, cada um com tamanho indicado por *tam*, ou NULL se o pedido não puder ser satisfeito. O espaço é inicializado para zero bytes.

malloc (tam)
Esta função retorna um ponteiro para o primeiro byte de uma região de memória de tamanho *tam* que foi reservada no Heap. Caso não haja memória suficiente para satisfazer a solicitação, a função devolve um ponteiro nulo. Deve-se sempre verificar se o valor devolvido não é um ponteiro nulo antes de utiliza-lo. A tentativa de usar um ponteiro nulo resultará geralmente numa quebra do sistema.

realloc (ptr, tam)
Esta função modifica o tamanho da memória previamente alocada apontada pelo ponteiro ptr para aquele tamanho especificado em tam. O valor tam pode ser maior ou menor que o original. Um ponteiro para o bloco de memória é devolvido porque a função pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorre, o conteúdo do bloco antigo é copiado no novo bloco; nenhuma informação é perdida.
Se o ponteiro ptr é um nulo, a função simplesmente aloca tam bytes de memória e devolve um ponteiro para a memória alocada. Se o tam é zero, a memória apontada por ptr é liberada.

abort ()
Esta função provoca a conclusão imediata do programa. Geralmente, nenhum arquivo é fechado. Em ambientes que a suportam, esta função devolve um valor definido pela implementação ao processo chamador, indicando a falha.

exit (num)
Esta função provoca a terminação normal imediata de um programa. O valor de num é passado ao processo chamador, normalmente o sistema operacional, se o ambiente o suporta. Por convenção, se o valor de num é zero, uma terminação normal do programa é assumida. Um valor diferente de zero pode indicar um erro definido pela implementação.

system (string)

Esta função passa a string como um comando para o processador de comandos do sistema operacional. Quando a função é chamada com um ponteiro para uma string nula, ela devolve um valor diferente de zero se um processador de comandos está presente; caso contrário devolve zero. O valor definido pela função, quando chamada com um ponteiro para a string é definido pela implementação. No entanto, geralmente devolve zero se o comando foi executado com sucesso; e um valor diferente de zero caso contrário.

8.6 – FUNÇÕES DE DATA E HORA (TIME.H)

clock()

Retorna a hora do processador usada pelo programa desde o início da execução, ou –1 se não estiver disponível.

time ()

Retorna a hora do calendário corrente ou –1 se a hora não estiver disponível.

difftime (hora2, hora1)

Retorna um valor double para a diferença entre hora2 e hora1 (hora2 – hora1) em segundos.

asctime (ptr)

Esta função converte a hora na estrutura ptr para uma cadeia no formato:

"Sun Jan 3 15:14:13 2001\n\0"

strftime (retorno, tamax, formato, ptr)

Esta função retorna em *retorno* a hora armazenada na estrutura *ptr* de acordo com o *formato* definido (semelhante ao printf). Os caracteres comuns (incluindo o \0 terminal) são copiados para *retorno*. Os formatadores para o *formato* são definidos na tabela abaixo. Não mais de *tamax* caracteres são colocados em *retorno*. A função pode retornar o número de caracteres, excluindo o \0, ou zero se forem produzidos mais do que *tamax* caracteres.

Formatador	Sigfnicado
%a	dia da semana abreviado
%A	dia da semana completo
%b	nome do mês abreviado
%B	nome do mês completo
%c	representação local da data e hora
%d	dia do mês (0 a 31)
%H	hora (relógio de 24 horas)(00-23)
%l	hora (relógio de 12 horas)(01-12)
%j	dia do ano (001-366)
%m	mês (01-12)
%M	minuto (00-59)
%p	equivalente local de AM e PM
%s	segundo (00-61)
%U	número da semana no ano (Dom. como 1º dia da semana)(00-53)
%w	dia da semana (0-6, domingo é 0)
%W	número da semana no ano (Seg. como 1º dia da semana)(00-53)
%x	representação local da data
%X	representação local da hora
%y	ano sem século (00-99)
%Y	ano com século.
%Z	nome do fuso horário, se houver.

8.7 – FUNÇÕES DE ENTRADA E SAÍDA ACESSÓRIAS (CONIO.H)

clrscr()
Esta função executa a limpeza da tela.

getch () ou getche ()
Estas funções não são definidas pelo padrão C ANSI. Porém, elas geralmente são incluídas em compiladores baseados no DOS. A função **getch()** devolve o primeiro caractere lido no console (teclado), mas ***não*** o mostra na tela. Já a função **getche()** faz a mesma coisa, mas mostra o caractere na tela.

getpass("Mensagem")
Esta função retorna uma string solicitada a partir da mensagem e que não fica visível na tela, como entrada de senhas.

kbhit ()
Esta função não é definida no padrão C ANSI. Porém, ela é encontrada sob diversas nomes, em virtualmente todos os compiladores C. Ela devolve um valor diferente de zero se uma tecla foi pressionada no console, caso contrário devolve zero.

cgets (buffer)
Lê o conteúdo do buffer de teclado.

cprintf ("String de controle", argumentos)
Escreve na janela de modo texto a string e argumentos formatados.

cputs (string)
Escreve a string na janela de modo texto.

cscanf ("formato", argumentos)
Lê os argumentos conforme o formato estabelecido.

delline()
Elimina a linha onde se encontra o cursor.

gettext (esquerda, cima, direita, baixo, retorno)
Copia o texto da janela de texto delimitada para a posição de memória indica em retorno.

gettextinfo(&nomevar)
Retorna uma estrutura com as informações de configuração de tela existente.

gotoxy(px, py)
Posiciona o cursor na posição específica px, py da janela de texto.

highvideo()
Selecione o modo de alta resolução de caracteres.

insline()
Insere uma nova linha em branco na posição do cursor da janela de texto.

lowvideo()
Seleciona o modo de baixa resolução de caracteres.

movetext(esquerda, cima, direita, baixo, destesq, destcima)
Copia o texto contido no retangulo definido por esquerda, cima, direita, baixo para o novo retângulo iniciado em destesq e destcima com as mesmas dimensões.

normvideo()
Retorna ao modo normal de resolução de caracteres.

_setcursortype (tipo)
Define o padrão de cursor a ser utilizado. O tipo pode variar conforme as opções abaixo:
_NOCURSOR – omite o cursor.
_SOLIDCURSOR – cursor sólido.
_NORMALCURSOR – cursor padrão.

textcolor (cor)
Define as características de cores para as letras de texto. Pode ser configurado conforme a tabela abaixo:

Constante simbólica	Valor	Constante simbólica	Valor
BLACK	0	DARKGRAY	8
BLUE	1	LIGHTBLUE	9
GREEN	2	LIGHTGREEN	10
CYAN	3	LIGHTCYAN	11
RED	4	LIGHTRED	12
MAGENTA	5	LIGHTMAGENTA	13
BROWN	6	YELLOW	14
LIGHTGRAY	7	WHITE	15
		BLINK	128

textbackground (cor)
Altera a cor de fundo da tela, as seguem a mesma definição da tabela acima até o valor 7.

textmode (modo)
Define o modo de apresentação da tela conforme a tabela abaixo:

Constante simbólica	Valor numérico	Modo de texto
LASTMODE	-1	Modo de texto anterior
BW40	0	Preto e branco com 40 colunas
C40	1	Colorido com 40 colunas
BW80	2	Preto e branco com 80 colunas
C80	3	Colorido com 80 colunas
MONO	7	Monocromático com 80 colunas
C4350	64	Padrão EGA com 43 linhas e padrão VGA com 50 linhas

wherex()
Retorna a posição horizontal atual do cursor.

wherey()
Retorna a posição vertical atual do cursor.

window(esquerda, cima, direita, baixo)
Define a janela de texto ativa com as dimensões definidas.