Edwar Saliba Júnior

Guia de Consulta Rápida da Linguagem C

Versão 1.0

Belo Horizonte - MG
Setembro de 2006

1. Nota:

Este trabalho é um resumo, com pequenas modificações, da bibliografia aqui referenciada.

2. Características da Linguagem C

Importante: A linguagem C é *case-sensitive* (sensível a caixa), ou seja, uma palavra escrita em maiúscula é diferente de uma palavra escrita em minúscula.

Exemplo:

```
Valor ? valor ? VALOR ? valoR ? VaLoR ? vALOr ? vaLor e assim por
diante...
```

2.1. Caracteres Válidos

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 0
+ - * / \ = | & ! ? # % ( ) { } [ ] _ ‘ “ . , : < >
```

2.2. Caracteres Especiais

Controle/Caracter	Seqüência de escape	Valor ASCII
nulo (null)	\0	00
campainha (bell)	\a	07
retrocesso (backspace)	\b	08
tabulação horizontal	\t	09
nova linha (new line)	\n	10
tabulação vertical	\v	11
alimentação de folha (form feed)	\f	12
retorno de carro (carriage return)	\r	13
aspas (")	\"	34
apostrofo (')	\'	39
interrogação (?)	\?	63
barra invertida (\)	\\	92

2.3. Operadores Aritméticos

Existem cinco operadores aritméticos em C. Além dos operadores aritméticos, existe ainda o operador **%** chamado operador de **módulo** cujo significado é o resto da divisão inteira. Os símbolos dos operadores aritméticos são:

Operador	Operação
+	adição.
-	subtração.
*	multiplicação
/	divisão
%	módulo (resto da divisão inteira)

2.4. Operadores de Atribuição Aritmética

Os símbolos usado são (`+=, -=, *=, /= , %=`).

A sintaxe da atribuição aritmética é a seguinte onde **var** é a variável e **exp** é uma expressão válida.

```
var += exp;
var -= exp;
var *= exp;
var /= exp;
var %= exp;
```

Estas instruções são equivalentes as seguintes:

```
var = var + exp;
var = var - exp;
var = var * exp;
var = var / exp;
var = var % exp;
```

Exemplo: Algumas atribuições aritméticas VÁLIDAS e suas respectivas formas sem o uso do operador de atribuição aritmético:

Expressão COM operador aritmético	Expressão SEM operador aritmético

`var += 5;`	`var = var + 5;`
`var -= 5;`	`var = var - 5;`
`var *= 5;`	`var = var * 5;`
`var /= 5;`	`var = var / 5;`
`var %= 5;`	`var = var % 5;`

2.5. Operadores Incrementais

Na linguagem C existem instruções muito comuns chamadas de operadores de **incremento** `(++)` e **decremento** `(--)`.

A sintaxe dos operadores incrementais é a seguinte:

`var++;`	`var = var + 1;`
`++var;`	`var = var + 1;`
`var--;`	`var = var - 1;`
`--var;`	`var = var - 1;`

Observe que existem duas sintaxes possíveis para os operadores. Pode-se colocar o operador **à esquerda** ou **á direita** da variável. Nos dois casos o valor da variável será incrementado (ou decrementado) de uma unidade. Porém se o operador for colocado **á esquerda** da variável, o valor da variável será incrementado (ou decrementado) **antes** que a variável seja usada em alguma outra operação. Caso o operador seja colocado **à direita** da variável, o valor da variável será incrementado (ou decrementado) **depois** que a variável for usada em alguma outra operação.

2.6. Operadores Relacionais e Lógicos

2.6.1. Operadores Relacionais:

Operadores relacionais verificam a relação de magnitude e igualdade entre dois valores. São seis os operadores relacionais em linguagem C:

Operador	**Significado**
`>`	`maior que`
`<`	`menor que`

```
>=      maior ou igual a (não menor que)
<=      menor ou igual a (não maior que)
==      igual a
!=      não igual a (diferente de)
```

2.6.2. Operadores Lógicos

São três os operadores lógicos na linguagem C:

Operadores em C	Equivalentes em Pascal
&&	AND
||	OR
!	NOT

A sintaxe de uso dos operadores lógicos:

```
expr_1 && expr_2
expr_1 || expr_2
!expr
```

onde `expr_1`, `expr_2` e *expr* são expressões válidas.

2.7. Operador Condicional

O operador condicional **? :** é usado para simplificar expressões condicionais.

A sintaxe de uma expressão condicional é:

```
condição ? expressão_1 : expressão_2;
```

A expressão acima poderia ser escrita também da seguinte forma:

```
if (condição) {
  expressao_1
}
else {
  expressao_2
}
```

2.8. Tipos de Dados

2.8.1. Comuns:

Tipo	Tamanho	Intervalo	Uso
char	1 byte	-128 a 127	Número muito pequeno e caracter ASCII
int	2 bytes	-32768 a 32767	Contador, controle de laço
float	4 bytes	3.4e-38 a 3.4e38	Real (precisão de 7 dígitos)
double	8 bytes	1.7e-308 a 1.7e308	Científico (precisão de 15 dígitos)

2.8.2. Modificados:

Tipo	Tamanho (bytes)	Intervalo
unsigned char	1	0 a 255
unsigned int	2	0 a 65 535
long int	4	-2.147.483.648 a 2.147.483.647
unsigned long int	4	0 a 4.294.967.295
long double	10	3.4e-4932 a 1.1e4932

2.9. Palavras Reservadas

asm	auto	break	case	cdecl	char
class	const	continue	_cs	default	delete
do	double	_ds	else	enum	_es
extern	_export	far	_fastcall	float	for
friend	goto	huge	if	inline	int
interrupt	_loadds	long	near	new	operator
pascal	private	protected	public	register	return
_saveregs	_seg	short	signed	sizeof	_ss
static	struct	switch	template	this	typedef
union	unsigned	virtual	void	volatile	while

2.10. Comentários

Em linguagem C, comentários podem ser escritos em qualquer lugar do texto para facilitar a interpretação do algoritmo. Para que o comentário seja identificado como tal, ele deve ter um **/*** antes e um ***/** depois.

Exemplo:

```
/* esta é uma linha de comentário em C */
```

Observação: O C++ permite que comentários sejam escritos de outra forma: colocando um *//* em uma linha, o compilador entenderá que tudo que estiver à direita do símbolo é um comentário.

Exemplo:

```
// este é um comentário valido apenas em C++
```

2.11. Constantes

A linguagem C possui quatro tipos básicos de constantes: **inteiras**, **ponto flutuante**, **caracteres** e ***strings***. As constantes inteiras e de ponto flutuante representam números, enquanto as de caracteres e strings representam letras e palavras.

2.11.1. Constantes inteiras

Uma constante inteira é um número de valor inteiro. Números inteiros podem ser escritos nos formatos:

1. **Decimal** (base 10):

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Exemplo: Algumas constantes inteiras decimais VÁLIDAS:

```
0   3   -45   26338   -7575   1010
```

Exemplo: Algumas constantes inteiras decimais INVÁLIDAS.

```
1.          (ponto)
1,2         (vírgula)
045         (primeiro dígito é 0: não é constante decimal)
```

```
212-22-33  (caracter ilegal: -)
```

2. **Hexadecimal** (base 16):

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (a, b, c, d, e)
```

Exemplo: A seguir são mostrados algumas constantes inteiras hexadecimais VÁLIDAS.

```
0x0   0x3   0x4f5a   0x2FFE   OxABCD   0xAaFf
```

Exemplo: Algumas constantes inteiras hexadecimais INVÁLIDAS

.

```
0x3.       (ponto)
0x1,e      (vírgula)
0x ff      (espaço)
FFEE       (não começa com 0x: não é constante hexadecimal)
0Xfg34     (caracter ilegal: g)
```

3. **Octal** (base 8):

```
0, 1, 2, 3, 4, 5, 6, 7
```

```
Observação: Na representação de números octais, o primeiro dígito
deve ser o número 0 (zero).
```

Exemplo: Algumas constantes inteiras decimais VÁLIDAS:

```
031   045   071   07575   01010
```

Exemplo: Algumas constantes inteiras decimais INVÁLIDAS.

```
01.        (ponto)
01,2       (vírgula)
0212-22-33 (caracter ilegal: -)
```

2.11.2. Constantes de ponto flutuante

Um número ponto flutuante DEVE ter um ponto decimal que não pode ser substituído por uma vírgula. Pode ser escrito em notação científica, neste caso o "x10" é substituído por "e" ou "E". Assim sendo, o número 1.23e4 representa 1.23 x 10^4 ou 12300.

Exemplo: Números de ponto flutuante VÁLIDOS.

```
0.234   125.65   .93   1.23e-9   -1.e2   10.6e18   -.853E+67
```

A forma de representação de um número real em C é bastante flexível. Por exemplo, o número 314 pode ser representado por qualquer uma das seguintes formas abaixo:

```
314.  3.14e2     +3.14e+2   31.4e1     .314E+3    314e0
```

2.11.3. Constantes caracteres

Uma constante caracter é uma letra ou símbolo colocado entre ASPAS SIMPLES.

Exemplo: Algumas constantes caracter VÁLIDAS.

```
'a'   'b'   'X'   '&'   '{'   ' '
```

2.11.4. Constantes string

Uma constante string consiste de um conjunto de caracteres colocados entre ASPAS DUPLAS. Embora as instruções do C usem apenas os caracteres do conjunto padrão ASCII, as constantes caracter e string podem conter caracteres do conjunto estendido ASCII: é, ã, ç, ü, ...

Exemplo: Abaixo seguem algumas constantes strings VÁLIDAS.

```
"Oba!"
"Caxias do Sul"
"A resposta é: "
"João Carlos da Silveira"
"a"
"isto é uma string"
```

2.11.5. Constantes Definidas pelo Programador

O programador pode definir constantes em qualquer programa.
A sintaxe da instrução de definição de uma constante:

```
#define nome_da_constante valor_da_constante
```

Onde **#define** é uma diretiva de compilação que diz ao compilador para trocar as ocorrências do texto **nome_da_constante** pelo **valor_da_constante**.

Exemplo: Algumas constantes.

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

2.12. Variáveis

A sintaxe para declaração de variáveis é a seguinte:

```
tipo var_1 [, var_2, ...];
```

Onde ***tipo*** é o tipo de dado e ***var_1*** é o nome da variável a ser declarada. Se houver mais de uma variável, seus nomes são separados por vírgulas.

Exemplo:

```
int i;
int x,y,z;
char letra;
float nota_1,nota_2,media;
double num;
```

Exemplo: Declaração de variáveis em um programa:

```
main(){
  float raio, area;     /* declaração de variáveis */
  raio = 2.5;
  área = 3.14 * raio * raio;
}
```

3. Bibliotecas Disponíveis

alloc.h	assert.h	bcd.h	bios.h	complex.h
conio.h	ctype.h	dir.h	dirent.h	dos.h
errno.h	fcntl.h	float.h	fstream.h	generic.h
graphics.h	io.h	iomanip.h	iostream.h	limits.h
locale.h	malloc.h	math.h	mem.h	process.h
setjmp.h	share.h	signal.h	stdarg.h	stddef.h
stdio.h	stdiostr.h	stdlib.h	stream.h	string.h
strstrea.h	sys\stat.h	sys\timeb.h	sys\types.h	time.h
values.h				

4. Comandos Mais Utilizados

Mostraremos, nas duas seções iniciais as mais importantes funções de entrada e saída de dados em C: as funções `printf()` e `scanf()`. A partir do estudo destas funções é possível escrever um programa propriamente dito com entrada, processamento e saída de dados.

4.1.1. printf()

Biblioteca: `stdio.h`

Declaração: `int printf (const char*` *`st_contr`* `[,` *`lista_arg`*`]);`

A função `printf()` (***print formated***) imprime dados da lista de argumentos `lista_arg` formatados de acordo com a string de controle `st_contr`. Esta função retorna um valor inteiro representando o número de caracteres impressos.

A *string de controle*, `st_contr`, é uma máscara que especifica (formata) o que será impresso e de que maneira será impresso.

Exemplo: Abaixo as instruções de saída formatadas e os respectivos resultados.

Instrução	**Saída**
`printf("Ola', Mundo!");`	`Ola', Mundo!`
`printf("linha 1 \nlinha 2 ");`	`linha 1` `linha 2`
`printf("Tenho %d anos.",idade);`	`Tenho 29 anos.`

Instrução:

```
printf("Total: %f.2 \nDinheiro: %f.2 \nTroco: %f.2",tot,din,din-tot);
```

Saída:

```
Total: 12.30
Dinheiro: 15.00
Troco: 2.70
```

Depois do sinal %, seguem-se alguns modificadores, cuja sintaxe é a seguinte:

```
% [flag] [tamanho] [.precisão] tipo
```

[flag] **justificação de saída: (Opcional)**

- `-` justificação à esquerda.
- `+` conversão de sinal (saída sempre com sinal: + ou -)
- `<espaço>` conversão de sinal (saídas negativas com sinal, positivas sem sinal)

[tamanho] **especificação de tamanho (Opcional)**

- `n` pelo menos n dígitos serão impressos (dígitos faltantes serão completados por brancos).

0n	pelo menos n dígitos serão impressos (dígitos faltantes serão completados por zeros).

[.precisão]	**especificador de precisão, dígitos a direita do ponto decimal. (Opcional)**
(nada)	padrão: 6 dígitos para reais.
.0	nenhum digito decimal.
.n	são impressos n dígitos decimais.

Tipo	**caracter de conversão de tipo (Requerido)**
d	inteiro decimal
o	inteiro octal
x	inteiro hexadecimal
f	ponto flutuante: [-]dddd.dddd.
e	ponto flutuante com expoente: [-]d.ddde[+/-]ddd
c	caracter simples
s	string

4.1.2. scanf()

Biblioteca: `stdio.h`

Declaração: `int scanf(const char* st_contr [, end_var, ...]);`

A função `scanf()` (*scan formated*) permite a entrada de dados numéricos, caracteres e 'strings' e sua respectiva atribuição a variáveis cujos endereços são `end_var`. Esta função é dita de entrada formatada pois os dados de entrada são formatados pela *string de controle* `st_contr`.

O uso da função `scanf()` é semelhante ao da função `printf()`. A função lê da entrada padrão (em geral, teclado) uma lista de valores que serão formatados pela string de controle e armazenados nos endereços das variáveis da lista. A string de controle é formada por um conjunto de especificadores de formato, cuja sintaxe é a seguinte:

```
% [*] [tamanho] tipo
```

*	**indicador de supressão (Opcional)**
<presente>	Se o indicador de supressão estiver presente o campo não é lido. Este supressor é útil quando não queremos ler um campo de dado armazenado em arquivo.
<ausente>	O campo é lido normalmente.

Tamanho	**especificador de tamanho(Opcional)**
n	Especifica n como o numero máximo de caracteres para leitura do campo.
<ausente>	Campo de qualquer tamanho.

Tipo	**define o tipo de dado a ser lido (Requerido)**
d	inteiro decimal (int)
f	ponto flutuante (float)
o	inteiro octal (int)
x	inteiro hexadecimal (int)
i	inteiro decimal de qualquer formato(int)
u	inteiro decimal sem sinal (unsigned int)
s	string (char*)
c	caracter (char)

A lista de variáveis é o conjunto de endereços de variáveis para os quais serão passados os dados lidos. Variáveis simples devem ser precedidas pelo caracter &. Vetores não são precedidos pelo caracter &.

4.1.3. getchar()

Biblioteca: `stdio.h`

Declaração: `int getchar(void);`

A função `getchar()` (*get character*) lê um caracter individual da entrada padrão (em geral, o teclado). Se ocorrer um erro ou uma condição de '*fim-de-arquivo*' durante a leitura, a função retorna o valor da constante simbólica `EOF` (*end of file*) definida na biblioteca stdio.h. Esta função permite uma forma eficiente de detecção de finais de arquivos.

4.1.4. putchar()

Biblioteca: `stdio.h`

Declaração: `int putchar(int c);`

Esta função `putchar()` (*put character*) imprime um caracter individual *c* na saída padrão (em geral o monitor de vídeo).

4.1.5. getch(), getche()

Biblioteca: `conio.h`

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

Estas funções fazem a leitura dos códigos de teclado. Estes códigos podem representar tecla s de caracteres (`A`, `y`, `*`, `8`, etc.), teclas de comandos ([`enter`], [`delete`], [`Page Up`], [`F1`], etc.) ou combinação de teclas ([`Alt`] + [`A`], [`Shift`] + [`F1`], [`Ctrl`] + [`Page Down`], etc.).

Ao ser executada, a função `getch()` (*get character*) aguarda que uma tecla (ou combinação de teclas) seja pressionada, recebe do teclado o código correspondente e retorna este valor. A função `getche()` (*get character and echoe*) também escreve na tela, quando possível, o caracter correspondente.

4.1.6. cprintf()

Biblioteca: `conio.h`

Declaração: `int cprintf (const char* st_contr [, lista_arg]);`

Esta função `cprintf()` (*color print formated*) permite a saída de dados usando cores. O uso da função `cprintf()` é semelhante à `printf()`. Para que a saída seja colorida é necessário definir as cores de fundo e de letra para a impressão antes do uso da função.

Cores (Modo Texto)

Cor	Constante	Valor	Fundo	Letra
Preto	BLACK	0	ok	ok
Azul	BLUE	1	ok	ok

Verde	GREEN	2	ok	ok
Cian	CYAN	3	ok	ok
Vermelho	RED	4	ok	ok
Magenta	MAGENTA	5	ok	ok
Marrom	BROWN	6	ok	ok
Cinza Claro	LIGHTGRAY	7	ok	ok
Cinza Escuro	DARKGRAY	8	--	ok
Azul Claro	LIGHTBLUE	9	--	ok
Verde Claro	LIGHTGREEN	10	--	ok
Cian Claro	LIGHTCYAN	11	--	ok
Vermelho Claro	LIGHTRED	12	--	ok
Magenta Claro	LIGHTMAGENTA	13	--	ok
Amarelo	YELLOW	14	--	ok
Branco	WHITE	15	--	ok
Piscante	BLINK	128	--	ok

Estas definições são feitas pelas funções `texcolor()` e `textbackground()` cuja sintaxe é:

```
textcolor(cor_de_letra);
textbackground(cor_de_fundo);
```

onde `cor_de_letra` e `cor_de_fundo` são números inteiros referentes as cores da palheta padrão (16 cores, modo texto). Estes valores de cor são representadas por constantes simbólicas definidas na biblioteca conio.h. Para se usar uma letra piscante deve-se adicionar o valor 128 ao valor da cor de letra.

Exemplo: O trecho de programa abaixo imprime uma mensagem de alerta em amarelo piscante sobre fundo vermelho.

```
#include <conio.h>
...
textbackground(RED);
textcolor(YELLOW + BLINK);
cprintif(" Alerta: Vírus Detectado! ");
...
```

4.1.7. sound(), delay(), nosound()

Biblioteca: `dos.h`

Declarações:
```
void sound(unsigned freq);
void delay(unsigned tempo);
void nosound(void);
```

A função `sound()` ativa o alto-falante do PC com uma freqüência *freq* (Hz). A função `delay()` realiza uma pausa (aguarda intervalo de tempo), a duração é determinada em milisegundos. A função `nosound()` desativa o alto-falante.

4.1.8. clrscr(), clreol()

Biblioteca: `conio.h`

Declarações:
```
void clrscr(void);
void clreol(void);
```

A função `clrscr()` (*clear screen*) limpa a janela de tela e posiciona o cursor na primeira linha e primeira coluna da janela (canto superior esquerdo da janela). A função `clreol()` (*clear to end of line*) limpa uma linha desde a posição do cursor até o final da linha mas não modifica a posição do cursor. Ambas funções preenchem a tela com a cor de fundo definida pela função `textbacground()`.

5. Estruturas de Controle

5.1. Condição de controle

Em todas as estruturas, existe pelo menos uma expressão que faz o controle de **qual** bloco de instruções será executado ou **quantas vezes** ele será executado: é o que chamamos de **condição de controle**. Uma condição de controle é uma expressão lógica ou aritmética cujo resultado pode ser considerado verdadeiro ou falso.

Exemplo: Observe nas condições abaixo, considere as variáveis:

```
int i = 0, j = 3;
```

condição	valor numérico	significado lógico
(i == 0)	1	verdadeiro
(i > j)	0	falso

```
(i)             0           falso
(j)             3           verdadeiro
```

5.2. do...while

Esta é uma estrutura básica de repetição condicional. Permite a execução de um bloco de instruções repetidamente. Sua sintaxe é a seguinte:

Sintaxe:

```
do{
  bloco de instruções
}while(condição);
```

Esta estrutura faz com que o bloco de instruções seja executado pelo menos uma vez. Após a execução do bloco, a condição é testada. Se a condição é **verdadeira** o bloco é executado outra vez, caso contrário a repetição é terminada.

Exemplo: No trecho abaixo, a leitura de um número é feita dentro de um laço de repetição condicional. A leitura é repetida caso o número lido seja negativo.

```
do{
  puts("Digite um número positivo:");
  scanf("%f",&num);
}while(num <= 0.0);
```

5.3. while

A estrutura de repetição condicional while é semelhante a estrutura do...while. Sua sintaxe é a seguinte:

Sintaxe:

```
while(condição){
  bloco de instruções
}
```

Esta estrutura faz com que a condição seja testada antes de executar o bloco de instruções. Se a condição é **verdadeira** o bloco é executado uma vez e a condição é avaliada novamente. Caso a condição seja **falsa** a repetição é terminada sem a execução do bloco. Observe que nesta estrutura, ao contrário da estrutura do...while, o bloco de instruções pode não ser executado nenhuma vez, basta que a condição seja inicialmente falsa.

Exemplo: No trecho abaixo, calcula-se a precisão (ε) do processador aritmético do PC. A variável eps tem seu valor dividido por 2 enquanto o processador conseguir distinguir entre 1 e 1+ε. Após a execução do laço, o valor de eps contém a precisão da máquina.

```
eps = 1.0;
while(1.0 + eps > 1.0){
  eps /= 2.0;
}
```

Exemplo2: No trecho abaixo o programa imprimira na tela os algarismos de 1 a 10.

```
cont = 1;
while(cont < 11){
  printf ("Número: %d", cont);
  cont++;
}
```

5.4. *for*

A estrutura for é muito semelhante as estruturas de repetição vistas anteriormente, entretanto costuma ser utilizada quando se quer um número determinado de ciclos. A contagem dos ciclos é feita por uma variável chamada de **contador**.

Sintaxe:

```
for(inicialização; condição; incremento){
  bloco de instrução
}
```

Esta estrutura executa um número determinado de repetições usando um contador de iterações. O contador é inicializado na expressão de `inicialização` **antes** da primeira iteração. Por exemplo: `i = 0;` ou `cont = 20;`. Então o bloco é executado e **depois** de cada iteração, o contador é incrementado de acordo com a expressão de `incremento`. Por exemplo: `i++` ou `cont -= 2`. Então a expressão de `condição` é avaliada: se a `condição` for verdadeira, o `bloco de instrução` é executado novamente e o ciclo recomeça, se a

condição é falsa termina-se o laço. Esta condição é, em geral, uma expressão lógica que determina o último valor do contador. Por exemplo: `i <= 100` ou `cont > 0`.

Exemplo: No trecho abaixo, o contador `i` é inicializado com o valor 1. O bloco é repetido enquanto a condição `i <= 10` for verdadeira. O contador é incrementado com a instrução `i++`. Esta estrutura, deste modo, imprime os números 1, 2, ..., 9, 10.

```
for(i=1; i<=10; i++){
  printf("Número: %d", i);
}
```

5.5. *if...else*

A estrutura `if...else` é a mais simples estrutura de controle da linguagem C. Esta estrutura permite executar um entre vários blocos de instruções. O controle de qual bloco será executado será dado por uma *condição* (expressão lógica ou numérica).

5.5.1. Decisão com um bloco:

```
if(condição){
  bloco de intruções
}
```

Se a condição **verdadeira**, o *bloco* é executado. Caso contrário, o bloco não é executado.

Exemplo: No trecho abaixo, se o valor lido for maior que 10, então o seu valor é redefinido como 10.

```
printf("Digite o número de repetições: (máximo 10)";
scanf("%d",&iter);
if(iter > 10){
  iter = 10;
}
```

5.5.2. Decisão entre dois blocos:

```
if(condição){
  bloco de instruções 1;
```

```
}else{
  bloco de instruções 2;
}
```

Se a condição for **verdadeira** o `bloco de instruções 1` é executado. Caso contrário, o `bloco de instruções 2` é executado.

Exemplo: No trecho abaixo, se o valor de raiz*raiz for maior que num o valor de raiz será atribuído a max, caso contrario, será atribuído a min.

```
if(raiz*raiz > num){
  max = raiz;
}
else{
  min = raiz;
}
```

5.5.3. Decisão de múltiplos blocos:

```
if(condição 1){
  bloco de instruções 1;
}
else {
   if(condição N){
      bloco de instruções N;
  }
  else {
    bloco de instruções P
  }
}
```

Se a condição 1 for **verdadeira** o `bloco de instruções 1` é executado. Caso contrário, a condição 2 é testada. Se a condição 2 for **verdadeira** o `bloco de instruções 2` é executado. Caso contrário, a condição 3 é testada e assim sucessivamente. Se nenhuma condição é **verdadeira**, o `bloco de instruções P` é executado. Observe que apenas um dos blocos é executado.

Exemplo: No trecho abaixo, uma determinada ação é executada se o valor de num for positivo, negativo ou nulo.

```
if(num > 0){
  a = b;
}
else {
  if(num < 0){
    a = b + 1;
  }
  else {
    a = b - 1;
  }
}
```

5.6. switch...case

A estrutura `switch..case` é uma estrutura de decisão que permite a execução de um conjunto de instruções a partir pontos diferentes conforme o resultado de uma expressão inteira de controle. O resultado desta expressão é comparado ao valor de cada um dos rótulos, e as instruções são executadas a partir desde rótulo.

Sintaxe: Esta estrutura possui a seguinte sintaxe:

```
switch(expressão){
     case rótulo_1:
       conjunto de instruções 1
     case rótulo_2:
       conjunto de instruções 2
     ...
     case rótulo_n:
       conjunto de instruções 1
     [default:
       conjunto de instruções d]
}
```

O valor de `expressão` é avaliado e o fluxo lógico será desviado para o conjunto cujo `rótulo` é igual ao resultado da expressão e todas as instruções **abaixo** deste rótulo serão executadas. Caso o resultado da expressão for diferente de todos os valores dos rótulos então `conjunto de instruções d` é executado. Os rótulos devem ser expressões constantes inteiras **diferentes** entre si. O rótulo `default` é opcional.

Esta estrutura é particularmente útil quando se tem um conjunto de instruções que se deve executar em ordem, porém se pode começar em pontos diferentes.

Exemplo: O trecho abaixo ilustra o uso da instrução switch em um menu de seleção. Neste exemplo, o programa iniciará o processo de usinagem de uma peça em um ponto qualquer dependendo do valor lido.

```
int selecao;
printf("Digite estágio de usinagem:");
scanf("%d",&selecao);
switch(selecao){
  case 1:
    // desbaste grosso...
    break;
  case 2:
    // desbaste fino...
    break;
  case 3:
    // acabamento...
    break;
  case 4:
    // polimento...
}
```

5.7. Comandos: break, continue, goto e a função exit()

As instruções vistas anteriormente podem sofrer **desvios** e **interrupções** em sua seqüência lógica normal através do uso certas instruções. As instruções que veremos a seguir devem ser usadas com muita parcimônia, pois fogem da lógica estruturada têm a tendência de tornar um programa incompreensível.

5.7.1. break

Esta instrução serve para terminar a execução das instruções de um laço de repetição (`for, do...while, while`) ou para terminar um conjunto `switch...case`.

Quando em um laço de repetição, esta instrução força a interrupção do laço independentemente da condição de controle.

Exemplo: No trecho abaixo um laço de repetição lê valores para o cálculo de uma média. O laço possui uma condição de controle sempre verdadeira o que, em princípio, é um erro: laço infinito. Porém, a saída do laço se dá pela instrução `break` que é executada quando um valor negativo é lido.

```
printf("digite valores:");
do{
  printf("valor:");
  scanf("%f", &val);
  if(val < 0){
    break;  // saída do laço
  }
  num++;
  soma += val;
}while(1);  // sempre verdadeiro
printf("média: %f",soma/num);
```

5.7.2. continue

Esta instrução opera de modo semelhante a instrução break dentro de um laço de repetição. Quando executada, ela pula as instruções de um laço de repetição sem sair do laço. Isto é, a instrução força a avaliação da condição de controle do laço.

Exemplo: No trecho abaixo revemos um laço de repetição lê valores para o cálculo de uma média. Se (val < 0.0) então o programa salta diretamente para a condição de controle, sem executar o resto das instruções.

```
printf("digite valores:");
```

```
do{
  printf ("valor:");
  scanf ("%f", &val);
  if(val < 0){      // se val é negativo...
    continue;       // ...salta para...
  }
  num++;            // se (val < 0) estas instruções
  soma += val;      // não são executadas!
}while(val >= 0);   // ...fim do laço
printf("média: %f",soma/num);
```

5.7.3. goto

Esta instrução é chamada de desvio de fluxo. A instrução desvia o programa para um rótulo (posição identificada) no programa. São raros os casos onde a instrução `goto` é necessária, no entanto, há certas circunstâncias, onde usada com prudência, ela pode ser útil.

Sintaxe:

```
goto rótulo;
...
rótulo:
...
```

Exemplo: No trecho abaixo revemos um laço de repetição que lê valores para o cálculo de uma média. Foram usadas duas instruções goto.

```
printf ("digite valores:");
inicio:               // rótulo
  printf ("valor:");
  scanf("%f",&val);
  if(val < 0){        // se val é negativo...
    goto fim;         // ...salta para fim
  }
  num++;              // se (val < 0.0) estas instruções
  soma += val;        // não são executadas!
```

```
goto inicio;          // salta para inicio
fim:                  // rótulo
printf("média: %f",soma/num);
```

5.7.4. exit()

Esta função (não instrução) `exit()`, da biblioteca `stdlib.h`, é uma função que termina a execução de um programa. Normalmente um programa é terminado quando se executa a última sua instrução, porém pode-se terminar a execução do programa a qualquer momento com o uso desta função.

A função `exit()` tem a seguinte declaração: `void exit(int status)`. Onde o argumento da função é um valor inteiro que será passado para o Sistema Operacional: (variável de sistema errorlevel no DOS).

Exemplo: No trecho abaixo revemos um laço de repetição lê valores para o cálculo de uma média. Foi usado a função exit para terminar a execução do programa.

```
printf("digite valores:");
do{
  printf ("valor:");
  scanf("%f",&val);
  if(val < 0.0){                  // se val é negativo...
    printf("média: %f",soma/num); // imprime resultado
    exit(0);                      // termina programa
  }
  num++;  soma += val;
}while(1);
```

6. Vetor

Em muitas aplicações queremos trabalhar com conjuntos de dados que são **semelhantes em tipo**. Por exemplo o conjunto das alturas dos alunos de uma turma, ou um conjunto de seus nomes. Nestes casos, seria conveniente poder colocar estas informações sob um mesmo conjunto, e poder referenciar cada dado individual deste conjunto por um número índice. Em programação, este tipo de estrutura de dados é chamada de **vetor** (ou *array*, em inglês) ou, de maneira mais formal **estruturas de dados homogêneas**.

Exemplo: A maneira mais simples de entender um vetor é através da visualização de um **lista**, de elementos com um nome coletivo e um índice de referência aos valores da lista.

```
n   nota
0   8.4
1   6.9
2   4.5
3   4.6
4   7.2
```

Nesta lista, `n` representa um número de referência e `nota` é o nome do conjunto. Assim podemos dizer que a 2ª nota é 6.9 ou representar `nota[1] = 6.9`

Declaração e inicialização de vetores

Sintaxe:

```
tipo nome[tam];
```

onde:

`tipo` é o **tipo** dos elementos do vetor: `int, float, double`...

`nome` é o **nome** identificador do vetor.

`tam` é o tamanho do vetor, isto é, o número de elementos que o vetor pode armazenar.

Exemplo: Veja as declarações seguintes:

```
int idade[100]; // declara um vetor chamado 'idade' do tipo
                // 'int' que recebe 100 elementos.
float nota[25]; // declara um vetor chamado 'nota' do tipo
                // 'float' que pode armazenar 25 números.
char nome[80];  // declara um vetor chamado 'nome' do tipo
                // 'char' que pode armazenar 80 caracteres.
```

Cada elemento do vetor é referenciado pelo **nome** do vetor seguido de um **índice** inteiro. O **primeiro** elemento do vetor tem índice `0` e o **último** tem índice `tam-1`.

Assim como podemos inicializar variáveis (por exemplo: `int j = 3;`), podemos inicializar vetores.

Sintaxe:

```
tipo nome[tam] = {lista de valores};
```

onde:

`lista de valores` é uma lista, separada por vírgulas, dos valores de cada elemento do vetor.

Exemplo: Veja as inicializações seguintes. Observe que a inicialização de nota gera o vetor do exemplo do início desta seção.

```
int dia[7] = {12,30,14,7,13,15,6};
float nota[5] = {8.4,6.9,4.5,4.6,7.2};
char vogal[5] = {'a', 'e', 'i', 'o', 'u'};
```

Opcionalmente, podemos inicializar os elementos do vetor enumerando-os um a um. Exemplo:

```
int cor_menu[4] = {BLUE,YELLOW,GREEN,GRAY};
```

ou

```
int cor_menu[4];
cor_menu[0] = BLUE;
cor_menu[1] = YELLOW;
cor_menu[2] = GREEN;
cor_menu[3] = GRAY;
```

Importante: Na linguagem C, devemos ter cuidado com os limites de um vetor. Embora na sua declaração, tenhamos definido o tamanho de um vetor, o C não faz nenhum teste de verificação de acesso a um elemento dentro do vetor ou não.
Por exemplo: Se declaramos um vetor como int valor[5], teoricamente só tem sentido usarmos os elementos valor[0], ..., valor[4]. Porém, o C não acusa **erro** se usarmos valor[12] em algum lugar do programa. Estes testes de limite **devem** ser feitos **logicamente** dentro do programa.

7. Matriz

Vetores podem ter mais de uma dimensão, isto é, mais de um índice de referência. Podemos ter vetores de duas, três, ou mais dimensões. Podemos entender um vetor de duas dimensões (por exemplo) associando-o aos dados de um tabela.

Exemplo: Um vetor bidimensional pode ser visualizado através de uma **tabela**.

```
nota      0      1      2
0        8.4    7.4    5.7
```

```
1       6.9   2.7   4.9
2       4.5   6.4   8.6
3       4.6   8.9   6.3
4       7.2   3.6   7.7
```

Nesta tabela representamos as notas de 5 alunos em 3 provas diferentes (matemática, física e química). O nome `nota` é o nome do conjunto, assim podemos dizer que a nota do 3º aluno na 2ª prova é 6.4 ou representar `nota[2,1] = 6.4`

Declaração e inicialização de Matrizes

A declaração e inicialização de vetores de mais de uma dimensão ou matriz é feita de modo semelhante aos vetores unidimensionais.

Sintaxe:

```
tipo nome[tam_1][tam_2]...[tam_N]={{lista},{lista},...{lista}};
```

onde:

`tipo` é o tipo dos elementos do vetor.

`nome` é o nome do vetor.

`[tam_1][tam_2]...[tam_N]` é o tamanho de cada dimensão do vetor.

`{{lista},{lista},...{lista}}` são as listas de elementos.

Exemplo: veja algumas declarações e inicializações de vetores de mais de uma dimensão. Observe que a inicialização de nota gera a tabela do exemplo do início desta seção.

```
float nota[5][3] = {{8.4, 7.4, 5.7},
                    {6.9, 2.7, 4.9},
                    {4.5, 6.4, 8.6},
                    {4.6, 8.9, 6.3},
                    {7.2, 3.6, 7.7}};
int tabela[2][3][2] = {{{10, 15}, {20, 25}, {30, 35}},
                       {{40, 45}, {50, 55}, {60, 65}};
```

Neste exemplo, nota é um vetor **duas** dimensões `([][])`. Este vetor é composto de 5 vetores de 3 elementos cada. Já `tabela` é vetor de três dimensões `([][][])`. Este vetor é composto de 2 vetores de 3 sub-vetores de 2 elementos cada.

8. Ponteiros

Toda informação (dado armazenado em variável simples ou vetor) que manipulamos em um programa está armazenado na memória do computador. Cada informação é representada por um certo conjunto de *bytes* (Ver capítulo 2). Por exemplo: caracter (char): 1 *byte*, inteiro (int): 2 *bytes*, etc.

Cada um destes conjuntos de *bytes*, que chamaremos de **bloco**, tem um nome e um **endereço** de localização especifica na memória.

Exemplo: Observe a instrução abaixo:

```
int num = 17;
```

Ao interpretarmos esta instrução, o processador pode especificar:

Nome da informação: `num`

Tipo de informação: `int`

Tamanho do bloco (número de *bytes* ocupados pela informação): `2`

Valor da informação: `17`

Endereço da informação (localização do primeiro ***byte***): `8F6F:FFF2` (hexadecimal)

Em geral, interessa ao **programador** apenas os nomes simbólicos que representam as informações, pois é com estes nomes que são realizadas as operações do seu algoritmo. Porém, ao **processador** interessa os endereços dos blocos de informação pois é com estes endereços que vai trabalhar.

Ponteiros são variáveis que contêm endereços. Neste sentido, estas variáveis *apontam* para algum determinado endereço da memória.

8.1.1. Declaração de ponteiros.

Quando declaramos um ponteiro, devemos declará-lo com o mesmo tipo (int, char, etc.) do bloco a ser apontado. Por exemplo, se queremos que um ponteiro aponte para uma variável `int` (bloco de 2 bytes) devemos declará-lo como `int` também.

A sintaxe da declaração de um ponteiro é a seguinte:

```
tipo_ptr *nome_ptr_1;
```

ou

```
tipo_ptr* nome_ptr_1, nome_ptr_2, ...;
```

onde:

`tipo_ptr` : é o tipo de bloco para o qual o ponteiro apontará.

`*` : é um operador que indica que `nome_ptr` é um ponteiro.

`nome_ptr_1, nome_ptr_2,...`: são os nomes dos ponteiros.

Exemplo: Veja as seguintes instruções:

```
int *p;
float* s_1, s_2;
```

A primeira instrução declara um ponteiro chamado `p` que aponta para um inteiro. Este ponteiro aponta para o **primeiro** endereço de um bloco de **dois** *bytes*. Sempre é necessário declarar o tipo do ponteiro. Neste caso dizemos que declaramos um ponteiro tipo `int`.

A segunda instrução declara dois ponteiros (`s_1` e `s_2`) do tipo `float`. Observe que o * está justaposto ao tipo: assim todos os elementos da lista serão declarados ponteiros.

8.1.2. Operadores `&` e *

Quando trabalhamos com ponteiros, queremos fazer duas coisas basicamente:

- conhecer **endereço** de uma variável;
- conhecer o **conteúdo** de um endereço.

Para realizar estas tarefas a linguagem C nos proporciona dois operadores especiais:

- o operador de endereço: **&**
- o operador de conteúdo: *

O operador de **endereço (&)** determina o endereço de uma variável (o primeiro *byte* do bloco ocupado pela variável). Por exemplo, `&val` determina o endereço do bloco ocupado pela variável `val`. Esta informação não é totalmente nova pois já a usamos antes: na função `scanf()`.

Exemplo: Quando escrevemos a instrução:

```
scanf("%d", &num);
```

Estamos nos referindo ao **endereço** do bloco ocupado pela variável `num`. A instrução significa: *"leia o buffer do teclado, transforme o valor lido em um valor inteiro (2 bytes) e o armazene no bloco localizado no endereço da variável* `num`*"*.

Exemplo: Para se atribuir a um ponteiro o endereço de uma variável escreve-se:

```
int *p, val=5;  // declaração de ponteiro e variável
p = &val;       // atribuição
```

Observe que o ponteiro p deve ser declarado anteriormente com o mesmo tipo da variável para a qual ele deve apontar.

O operador **conteúdo** (*) determina o conteúdo (valor) do dado armazenado no endereço de um bloco apontado por um ponteiro. Por exemplo, *p determina conteúdo do bloco apontado pelo ponteiro p. De forma resumida: o operador (*) determina o conteúdo de um endereço.

Exemplo: Para se atribuir a uma variável o conteúdo de um endereço escreve-se:

```
int *p = 0x3f8, val;  // declaração de ponteiro e variável
val = *p;             // atribuição
```

8.1.3. Operações elementares com ponteiros

Ponteiros são variáveis especiais e obedecem a regras especiais. Deste modo, existem uma série de operações (aritméticas, lógicas, etc.) envolvendo ponteiros que são permitidas e outras não. A seguir são destacadas algumas operações que podem ser executadas com ponteiros.

- A um ponteiro pode ser atribuído o endereço de uma variável comum.

Exemplo: Observe o trecho abaixo:

```
...
int *p;
int s;
p = &s;  // p recebe o endereço de s
...
```

- Um ponteiro pode receber o valor de outro ponteiro, isto é, pode receber o endereço apontado por outro ponteiro, desde que os ponteiros sejam de mesmo tipo.

Exemplo: Observe o trecho abaixo:

```
...
float *p1, *p2, val;
p1 = &val; // p1 recebe o endereço de val...
p2 = p1;   // ...e p2 recebe o conteúdo de p2 (endereço de val)
```

- Um ponteiro pode receber um endereço de memória diretamente. Um endereço é um número inteiro. Em geral, na forma hexadecimal (0x....). Nesta atribuição devemos, em geral, forçar uma conversão de tipo usando *typecast*[1] para o tipo de ponteiro declarado.

Exemplo: Observe o trecho abaixo:

```
...
int *p1;
float p2;
p1 = 0x03F8;  // endereço da porta serial COM1
p2 = (float)0x0FF6; // casting
...
```

- A um ponteiro pode ser atribuído o valor **nulo** usando a constante simbólica `NULL` (declarada na biblioteca `stdlib.h`). Um ponteiro com valor `NULL` não aponta para lugar nenhum! Algumas funções a utilizam para registrar uma atribuição ilegal ou sem sucesso (ver função `malloc()` adiante).

Exemplo:

```
#include <stdlib.h>
...
char *p;
p = NULL;
...
```

- Uma quantidade inteira pode ser adicionada ou subtraída de um ponteiro. A adição de um inteiro `n` a um ponteiro `p` fará com que ele aponte para o endereço do n-*ésimo* bloco seguinte.

[1] **type conversion** ou **typecasting** refere-se a mudança do tipo de uma entidade em outra. Isto é feito para obter vantagens e flexibilidade nas características de algumas hierarquias. (Disponível em: < http://en.wikipedia.org/wiki/Typecast_(programming)> Acesso em: 15 set. 2006)

Exemplo: Observe o trecho abaixo:

```
...
double *p, *q, var;
p = &var
q = ++p;  // q aponta para o bloco seguinte ao ocupado por var
p = q - 5;// p aponta para o quinto bloco anterior a q
...
```

- Dois ponteiros podem ser comparados (usando-se operadores lógicos) desde que sejam de mesmo tipo.

Exemplo: Observe o trecho abaixo:

```
...
if(px == py){ // se px aponta para o mesmo bloco que py ...
if(px > py){  // se px aponta para um bloco posterior a py ...
if(px != py){ // se px aponta para um bloco diferente de py ...
if(px == NULL) // se px é nulo...
...
```

8.1.4. Alocação Dinâmica de Memória

Os elementos de um vetor são armazenados **seqüencialmente** na memória do computador. Na declaração de um vetor, (por exemplo: `int vet[10]`) é dito ao processador reservar (**alocar**) um certo número de blocos de memória para armazenamento dos elementos do vetor. Porém, neste modo de declaração, não se pode alocar um número variável de elementos.

A linguagem C permite alocar dinamicamente (em tempo de execução), blocos de memória usando ponteiros. Dada a íntima relação entre ponteiros e vetores, isto significa que podemos declarar dinamicamente vetores de tamanho variável. Isto é desejável caso queiramos poupar memória.

Para a alocação de memória usamos a função `malloc()` (*memory allocation*) da biblioteca `alloc.h`. A função `malloc()` reserva, seqüencialmente, um certo número de blocos de memória e retorna, para um ponteiro, o endereço do primeiro bloco reservado.

Sintaxe:

```
pont = (tipo *)malloc(tam);
```

onde:

`pont` é o nome do ponteiro que recebe o endereço do espaço de memória alocado.

`tipo` é o tipo do endereço apontado (tipo do ponteiro).

`tam` é o tamanho do espaço alocado: numero de *bytes*.

A sintaxe seguinte, porem, é mais clara:

```
pont = (tipo*)malloc(num*sizeof(tipo));
```

onde:

`num` É o número de elementos que queremos poder armazenar no espaço alocado.

Exemplo: Se queremos declarar um vetor chamado vet, tipo int, com num elementos podemos usar o trecho abaixo:

```
...
int *vet;  // declaração do ponteiro
vet = (int*)malloc(num*2); // alocação de num blocos de 2 bytes
...
```

ou ainda

```
...
int *vet; // declaração do ponteiro
vet = (int*) malloc(num * sizeof(int));
...
```

Caso não seja possível alocar o espaço requisitado a função `malloc()` retorna a constante simbólica `NULL`.

Para liberar (desalocar) o espaço de memória se usa a função `free()`, cuja sintaxe é a seguinte:

```
free(pont);
```

onde:

`pont` É o nome do ponteiro que contém o endereço inicial do espaço de memória reservado.

8.1.5. Ponteiros para Funções

Até agora usamos ponteiros para apontar para endereços de memória onde se encontravam as variáveis (dados). Algumas vezes é necessário apontar para funções, isto é, apontar para o endereço de memória que contem o início das instruções de uma função. Quando assim procedemos, dizemos que usaremos *ponteiros para funções*.

Um uso de ponteiros para funções é passar uma função como **argumento** de outra função. Mas também pode-se usar ponteiros para funções ao invés de funções nas chamadas normais de funções.

Sintaxe:

```
tipo_r (*nome_p)(lista);
```

onde:

`tipo_r` é o tipo de retorno da função apontada.

`nome_p` é o nome do ponteiro que apontara para a função.

`lista` é a lista de argumentos da função.

Exemplo: Suponha que temos uma função é declarada como:

```
float fun(int a, int b){
   ...
}
```

O ponteiro correspondente será:

```
float (*pt)(int, int);
```

Observe que o ponteiro para função **deve ser** declarado entre parênteses. Observe também que o ponteiro e a função retornam o mesmo tipo de dado e que tem os mesmos argumentos.

Sintaxe:

```
pont = &funcao;
```

onde:

`pont` é o nome do ponteiro.

`funcao` é o nome da função.

Se um ponteiro contém o endereço de uma função, ele pode ser usado no lugar da chamada da função.

Exemplo: O trecho de programa abaixo usa um ponteiro para chamar uma função:

```
float fun(int a,int b){
  ...
}
void main(void){
  float temp;
  float (*pt)(int,int);
  pt = &fun;
  temp = (*pt)(10,20); // eqüivale a: temp = fun(10,20);
  ...
}
```

8.1.6. Passando uma Função como Argumento de Outra Função

Outra utilização de ponteiros para funções é na passagem de uma **função** como **argumento** para outra função. Para que isso ocorra necessitamos:

- Na declaração da **função a ser passada**:

i) Nada de especial, apenas a definição normal:

```
tipo nome_p(lista){
  ...
}
```

Exemplo:

```
float soma(float a,float b){
  ...
}
```

- Na **função receptora**:

i) Declarar o **ponteiro** que recebe a **função passada** na lista de argumentos:

```
tipo nome_r(..., tipo (*pt)(lista), ...){
```

Exemplo:

```
float grad(float x, float y, float (*p)(float,float)){
```

ii) Usar o ponteiro para funções nas chamadas da função passada:

```
var = (*pt)(lista);
```

Exemplo:

```
valor = (*p)(x,y);
```

- Na função principal:

i) Passar o nome da função chamada para a função receptora:

```
var = nome_g(... , nome_p , ...);
```

Exemplo:

```
g = grad(x,y,soma);
```

9. Tabela ASCII

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

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

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

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

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

Caracter	Dec.	Hex.
<espaço>	32	20h
!	33	21h
"	34	22h
#	35	23h
$	36	24h
%	37	25h
&	38	26h
'	39	27h
(	40	28h
)	41	29h
*	42	2Ah
+	43	2Bh
,	44	2Ch
-	45	2Dh
.	46	2Eh
/	47	2Fh
0	48	30h
1	49	31h
2	50	32h
3	51	33h
4	52	34h
5	53	35h
6	54	36h
7	55	37h
8	56	38h
9	57	39h
:	58	3Ah
;	59	3Bh
<	60	3Ch
=	61	3Dh
>	62	3Eh
?	63	3Fh

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

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

Caracter	Dec.	Hex.
Ç	128	80h
ü	129	81h
é	130	82h
â	131	83h
ä	132	84h
à	133	85h
å	134	86h
ç	135	87h
ê	136	88h
ë	137	89h
è	138	8Ah
ï	139	8Bh
Caracter	Dec.	Hex.
î	140	8Ch
ì	141	8Dh
Ä	142	8Eh
Å	143	8Fh
É	144	90h
æ	145	91h
Æ	146	92h
ô	147	93h
ö	148	94h
ò	149	95h
û	150	96h
ù	151	97h
ÿ	152	98h
Ö	153	99h
Ü	154	9Ah
¢	155	9Bh
£	156	9Ch
¥	157	9Dh
P	158	9Eh
ƒ	159	9Fh
ááááá	160	A0h

Caracter	Dec.	Hex.
í	161	A1h
ó	162	A2h
ú	163	A3h
ñ	164	A4h
Ñ	165	A5h
ª	166	A6h
º	167	A7h
¿	168	A8h
¬	169	A9h
¬	170	AAh
½	171	ABh
¼	172	ACh
¡	173	ADh
«	174	AEh
»	175	AFh
¦	176	B0h
Caracter	Dec.	Hex.
¦	177	B1h
¦	178	B2h
¦	179	B3h
¦	180	B4h
¦	181	B5h
¦	182	B6h
+	183	B7h
+	184	B8h
¦	185	B9h
¦	186	BAh
+	187	BBh
+	188	BCh
+	189	BDh
+	190	BEh
+	191	BFh
+	192	C0h
-	193	C1h

Caracter	Dec.	Hex.
-	194	C2h
+	195	C3h
-	196	C4h
+	197	C5h
¦	198	C6h
¦	199	C7h
+	200	C8h
+	201	C9h
-	202	CAh
-	203	CBh
¦	204	CCh
-	205	CDh
+	206	CEh
-	207	CFh
-	208	DOh
-	209	D1h
-	210	D2h
+	211	D3h
+	212	D4h
+	213	D5h
Caracter	Dec.	Hex.
+	214	D6h
+	215	D7h
+	216	D8h
+	217	D9h
+	218	DAh
¦	219	DBh
_	220	DCh
¦	221	DDh
¦	222	DEh
¯	223	DFh
a	224	E0h
ß	225	E1h
G	226	E2h

p	227	E3h
S	228	E4h
s	229	E5h
µ	230	E6h
t	231	E7h
F	232	E8h
T	233	E9h
O	234	EAh
d	235	EBh
8	236	ECh
f	237	EDh
∈	238	EEh
n	239	EFh
=	240	F0h
±	241	F1h
=	242	F2h
=	243	F3h
(	244	F4h
)	245	F5h
÷	246	F6h
~	247	F7h
°	248	F8h
·	249	F9h
·	250	FAh
Caracter	Dec.	Hex.
v	251	FBh
n	252	FCh
²	253	FDh
·	254	FEh
	255	FFh

Entre os caracteres da tabela ASCII estendidos os mais úteis estão, talvez, os caracteres de desenho de quadro em linhas simples e duplas: os caracteres de B3h até DAh (179 a 218 decimal). Como a visualização deste conjunto é difícil, o desenho abaixo pode auxiliar nesta tarefa:

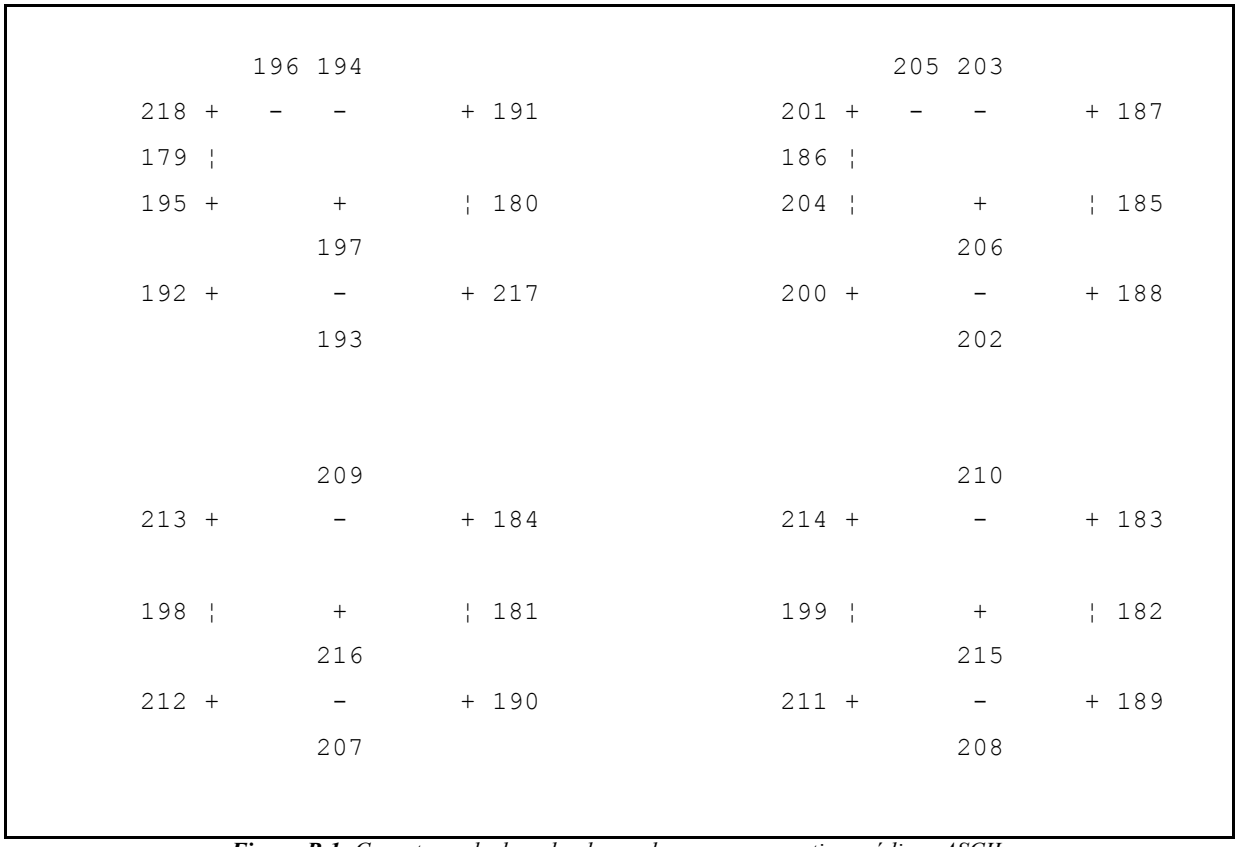

Figura B.1: *Caracteres de desenho de quadro e seus respectivos códigos ASCII.*

10. Bibliografia

FUNDAMENTOS DE LINGUAGEM C: Tudo que você precisa saber sobre C para não passar vergonha!, Centro Tecnológico de Mecatrônica. Caxias do Sul, RS, 1997.