A Precedence Table

Description	Operator	Associativity
Function expression	( )	Left to Right
Array Expression	[ ]	Left to Right
Structure operator	->	Left to Right
Structure operator	.	Left to Right
Unary minus	-	Right to left
Increment/Decrement	++	Right to Left
One's compliment	~	Right to left
Negation	!	Right to Left
Address of	&	Right to left
Value of address	*	Right to left
Type cast	(type)	Right to left
Size in bytes	sizeof	Right to left
Multiplication	*	Left to right
Division	/	Left to right
Modulus	%	Left to right
Addition	+	Left to right
Subtraction	-	Left to right
Left shift	<<	Left to right
Right shift	>>	Left to right
Less than	<	Left to right
Less than or equal to	<=	Left to right
Greater than	>	Left to right
Greater than or equal to	>=	Left to right
Equal to	==	Left to right
Not equal to	!=	Left to right

Continued...

Continued...

Description	Operator	Associativity
Bitwise AND	&	Left to right
Bitwise exclusive OR	^	Left to right
Bitwise inclusive OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	? :	Right to left
Assignment	=	Right to left
	*= /= %=	Right to left
	+= -= &=	Right to left
	^= |=	Right to left
	<<= >>=	Right to left
Comma	,	Right to left

Figure A1.1

B Standard Library Functions

- Standard Library Functions
- Arithmetic Functions
- Data Conversion Functions
- Character Classification Functions
- String Manipulation Functions
- Searching and Sorting Functions
- I/O Functions
- File Handling Functions
- Directcry Control Functions
- Buffer Manipulation Functions
- Disk I/O Functions
- Memory Allocation Functions
- Process Control Functions
- Graphics Functions
- Time Related Functions
- Miscellaneous Functions
- DOS Interface Functions

Let alone discussing each standard library function in detail, even a complete list of these functions would occupy scores of pages. However, this book would be incomplete if it has nothing to say about standard library functions. I have tried to reach a compromise and have given a list of standard library functions that are more popularly used so that you know what to search for in the manual. An excellent book dedicated totally to standard library functions is Waite group's, Turbo C Bible, written by Nabjyoti Barkakti.

Following is the list of selected standard library functions. The functions have been classified into broad categories.

Arithmetic Functions

Function	Use
abs	Returns the absolute value of an integer
cos	Calculates cosine
cosh	Calculates hyperbolic cosine
exp	Raises the exponential e to the x^{th} power
fabs	Finds absolute value
floor	Finds largest integer less than or equal to argument
fmod	Finds floating-point remainder
hypot	Calculates hypotenuse of right triangle
log	Calculates natural logarithm
log10	Calculates base 10 logarithm
modf	Breaks down argument into integer and fractional parts
pow	Calculates a value raised to a power
sin	Calculates sine
sinh	Calculates hyperbolic sine
sqrt	Finds square root
tan	Calculates tangent
tanh	Calculates hyperbolic tangent

Data Conversion Functions

Function	Use
atof	Converts string to float
atoi	Converts string to int
atol	Converts string to long
ecvt	Converts double to string
fcvt	Converts double to string
gcvt	Converts double to string
itoa	Converts int to string
ltoa	Converts long to string
strtod	Converts string to double
strtol	Converts string to long integer
strtoul	Converts string to an unsigned long integer
ultoa	Converts unsigned long to string

Character classification Functions

Function	Use
isalnum	Tests for alphanumeric character
isalpha	Tests for alphabetic character
isdigit	Tests for decimal digit
islower	Tests for lowercase character
isspace	Tests for white space character
isupper	Tests for uppercase character
isxdigit	Tests for hexadecimal digit
tolower	Tests character and converts to lowercase if uppercase
toupper	Tests character and converts to uppercase if lowercase

String Manipulation Functions

Function	Use
strcat	Appends one string to another
strchr	Finds first occurrence of a given character in a string
strcmp	Compares two strings
strcmpi	Compares two strings without regard to case
strcpy	Copies one string to another
strdup	Duplicates a string
stricmp	Compares two strings without regard to case (identical to strcmpi)
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strncat	Appends a portion of one string to another
strncmp	Compares a portion of one string with portion of another string
strncpy	Copies a given number of characters of one string to another
strnicmp	Compares a portion of one string with a portion of another without regard to case
strrchr	Finds last occurrence of a given character in a string
strrev	Reverses a string
strset	Sets all characters in a string to a given character
strstr	Finds first occurrence of a given string in another string
strupr	Converts a string to uppercase

Searching and Sorting Functions

Function	Use
bsearch	Performs binary search
lfind	Performs linear search for a given value
qsort	Performs quick sort

I/O Functions

Function	Use
Close	Closes a file
fclose	Closes a file
feof	Detects end-of-file
fgetc	Reads a character from a file
fgetchar	Reads a character from keyboard (function version)
fgets	Reads a string from a file
fopen	Opens a file
fprintf	Writes formatted data to a file
fputc	Writes a character to a file
fputchar	Writes a character to screen (function version)
fputs	Writes a string to a file
fscanf	Reads formatted data from a file
fseek	Repositions file pointer to given location
ftell	Gets current file pointer position
getc	Reads a character from a file (macro version)
getch	Reads a character from the keyboard
getche	Reads a character from keyboard and echoes it
getchar	Reads a character from keyboard (macro version)
gets	Reads a line from keyboard
inport	Reads a two-byte word from the specified I/O port
inportb	Reads one byte from the specified I/O port
kbhit	Checks for a keystroke at the keyboard
lseek	Repositions file pointer to a given location
open	Opens a file
outport	Writes a two-byte word to the specified I/O port
outportb	Writes one byte to the specified I/O port
printf	Writes formatted data to screen
putc	Writes a character to a file (macro version)
putch	Writes a character to the screen
putchar	Writes a character to screen (macro version)
puts	Writes a line to file
read	Reads data from a file

rewind	Repositions file pointer to beginning of a file
scanf	Reads formatted data from keyboard
sscanf	Reads formatted input from a string
sprintf	Writes formatted output to a string
tell	Gets current file pointer position
write	Writes data to a file

File Handling Functions

Function	Use
remove	Deletes file
rename	Renames file
unlink	Deletes file

Directory Control Functions

Function	Use
chdir	Changes current working directory
getcwd	Gets current working directory
fnsplit	Splits a full path name into its components
findfirst	Searches a disk directory
findnext	Continues *findfirst* search
mkdir	Makes a new directory
rmdir	Removes a directory

Buffer Manipulation Functions

Function	Use
memchr	Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer
memcmp	Compares a specified number of characters from two buffers

memcpy	Copies a specified number of characters from one buffer to another
memicmp	Compares a specified number of characters from two buffers without regard to the case of the characters
memmove	Copies a specified number of characters from one buffer to another
memset	Uses a given character to initialize a specified number of bytes in the buffer

Disk I/O Functions

Function	Use
absread	Reads absolute disk sectors
abswrite	Writes absolute disk sectors
biosdisk	Performs BIOS disk services
getdisk	Gets current drive number
setdisk	Sets current disk drive

Memory Allocation Functions

Function	Use
calloc	Allocates a block of memory
farmalloc	Allocates memory from far heap
farfree	Frees a block from far heap
free	Frees a block allocated with *malloc*
malloc	Allocates a block of memory
realloc	Reallocates a block of memory

Process Control Functions

Function	Use
abort	Aborts a process
atexit	Executes function at program termination

execl	Executes child process with argument list
exit	Terminates the process
spawnl	Executes child process with argument list
spawnlp	Executes child process using PATH variable and argument list
system	Executes an MS-DOS command

Graphics Functions

Function	Use
arc	Draws an arc
ellipse	Draws an ellipse
floodfill	Fills an area of the screen with the current color
getimage	Stores a screen image in memory
getlinestyle	Obtains the current line style
getpixel	Obtains the pixel's value
lineto	Draws a line from the current graphic output position to the specified point
moveto	Moves the current graphic output position to a specified point
pieslice	Draws a pie-slice-shaped figure
putimage	Retrieves an image from memory and displays it
rectangle	Draws a rectangle
setcolor	Sets the current color
setlinestyle	Sets the current line style
putpixel	Plots a pixel at a specified point
setviewport	Limits graphic output and positions the logical origin within the limited area

Time Related Functions

Function	Use
clock	Returns the elapsed CPU time for a process
difftime	Computes the difference between two times

ftime	Gets current system time as structure
strdate	Returns the current system date as a string
strtime	Returns the current system time as a string
time	Gets current system time as long integer
setdate	Sets DOS date
getdate	Gets system date

Miscellaneous Functions

Function	Use
delay	Suspends execution for an interval (milliseconds)
getenv	Gets value of environment variable
getpsp	Gets the Program Segment Prefix
perror	Prints error message
putenv	Adds or modifies value of environment variable
random	Generates random numbers
randomize	Initializes random number generation with a random value based on time
sound	Turns PC speaker on at specified frequency
nosound	Turns PC speaker off

DOS Interface Functions

Function	Use
FP_OFF	Returns offset portion of a far pointer
FP_SEG	Returns segment portion of a far pointer
getvect	Gets the current value of the specified interrupt vector
keep	Installs terminate-and-stay-resident (TSR) programs
int86	Issues interrupts
int86x	Issues interrupts with segment register values
intdos	Issues interrupt 21h using registers other than DX and AL
intdosx	Issues interrupt 21h using segment register values
MK_FP	Makes a far pointer

segread	Returns current values of segment registers
setvect	Sets the current value of the specified interrupt vector

C Chasing The Bugs

C programmers are great innovators of our times. Unhappily, among their most enduring accomplishments are several new techniques for wasting time. There is no shortage of horror stories about programs that took twenty times to 'debug' as they did to 'write'. And one hears again and again about programs that had to be rewritten all over again because the bugs present in it could not be located. A typical C programmer's 'morning after' is red eyes, blue face and a pile of crumpled printouts and dozens of reference books all over the floor. Bugs are C programmer's birthright. But how do we chase them away. No sure-shot way for that. I thought if I make a list of more common programming mistakes it might be of help. They are not arranged in any particular order. But as you would realize surely a great help!

[1] Omitting the ampersand before the variables used in **scanf( )**.

For example,

```
int choice ;
scanf ( "%d", choice ) ;
```

Here, the **&** before the variable choice is missing. Another common mistake with **scanf( )** is to give blanks either just before the format string or immediately after the format string as in,

```
int choice ;
scanf ( " %d ", choice ) ;
```

Note that this is not a mistake, but till you don't understand **scanf( )** thoroughly, this is going to cause trouble. Safety is in eliminating the blanks. Thus, the correct form would be,

```
int choice ;
scanf ( "%d", &choice ) ;
```

[2] Using the operator = instead of the operator = =.

What do you think will be the output of the following program:

```
main( )
{
    int i = 10 ;

    while ( i = 10 )
    {
        printf ( "got to get out" ) ;
        i++ ;
    }
}
```

At first glance it appears the message will be printed once and the control will come out of the loop since **i** becomes 11. But, actually we have fallen in an indefinite loop. This is because the = used in the condition always assigns the value 10 to **i**, and since **i** is non-zero the condition is satisfied and the body of the loop is executed over and over again.

[3] Ending a loop with a semicolon.

Observe the following program.

```
main( )
{
    int j = 1 ;

    while ( j <= 100 ) ;
    {
        printf ( "\nCompguard" ) ;
        j++ ;
    }
}
```

Inadvertently, we have fallen in an indefinite loop. Cause is the semicolon after **while**. This in effect makes the compiler feel that you wanted the loop to work in the following manner:

```
while ( j <= 100 ) ;
```

This is an indefinite loop since **j** never gets incremented and hence eternally remains less that 100.

[4] Omitting the **break** statement at the end of a **case** in a **switch** statement.

Remember that if a **break** is not included at the end of a **case**, then execution will continue into the next **case**.

```
main( )
{
    int ch = 1 ;

    switch ( ch )
    {
        case 1 :
            printf ( "\nGoodbye" ) ;
        case 2 :
            printf ( "\nLieutenant" ) ;
    }
}
```

Here, since the **break** has not been given after the **printf( )** in **case 1**, the control runs into **case 2** and executes the second **printf( )** as well.

However, this sometimes turns out to be a blessing in disguise. Especially, in cases when we are checking whether the value of a variable equals a capital letter or a small case letter. This example has been succinctly explained in Chapter 4.

[5] Using **continue** in a **switch**.

It is a common error to believe that the way the keyword **break** is used with loops and a **switch**; similarly the keyword **continue** can also be used with them. Remember that **continue** works only with loops, never with a **switch**.

[6] A mismatch in the number, type and order of actual and formal arguments.

```
yr = romanise ( year, 1000, 'm' ) ;
```

Here, three arguments in the order **int**, **int** and **char** are being passed to **romanise( )**. When **romanise( )** receives these arguments into formal arguments they must be received in the same order. A careless mismatch might give strange results.

[7] Omitting provisions for returning a non-integer value from a function.

If we make the following function call,

```
area = area_circle ( 1.5 ) ;
```

then while defining **area_circle( )** function later in the program, care should be taken to make it capable of returning a floating point value. Note that unless otherwise mentioned the compiler would assume that this function returns a value of the type **int**.

[8] Inserting a semicolon at the end of a macro definition.

How do you recognize a C programmer? Ask him to write a paragraph in English and watch whether he ends each sentence with a semicolon. This usually happens because a C programmer becomes habitual to ending all statements with a semicolon. However, a semicolon at the end of a macro definition might create a problem. For example,

```
#define UPPER 25 ;
```

would lead to a syntax error if used in an expression such as

```
if ( counter == UPPER )
```

This is because on preprocessing, the **if** statement would take the form

```
if ( counter == 25 )
```

[9] Omitting parentheses around a macro expansion.

```
#define SQR(x) x * x
main( )
{
    int a ;

    a = 25 / SQR ( 5 ) ;
    printf ( "\n%d", a ) ;
}
```

In this example we expect the value of **a** to be 1, whereas it turns out to be 25. This so happens because on preprocessing the arithmetic statement takes the following form:

```
a = 25 / 5 * 5 ;
```

[10] Leaving a blank space between the macro template and the macro expansion.

```
#define ABS (a) ( a = 0  ? a : -a )
```

Here, the space between **ABS** and **(a)** makes the preprocessor believe that you want to expand **ABS** into **(a)**, which is certainly not what you want.

[11] Using an expression that has side effects in a macro call.

```
#define SUM ( a ) ( a + a )
main( )
{
    int w, b = 5 ;

    w = SUM( b++ ) ;
    printf ( "\n%d", w ) ;
}
```

On preprocessing, the macro would be expanded to,

```
w = ( b++ ) + ( b++ ) ;
```

If you are wanting to first get sum of 5 and 5 and then increment **b** to 6, that would not happen using the above macro definition.

[12] Confusing a character constant and a character string.

In the statement

```
ch = 'z' ;
```

a single character is assigned to **ch**. In the statement

```
ch = "z" ;
```

a pointer to the character string "a" is assigned to **ch**.

Note that in the first case, the declaration of **ch** would be,

```
char ch ;
```

whereas in the second case it would be,

```
char *ch ;
```

[13] Forgetting the bounds of an array.

```
main( )
{
    int num[50], i ;

    for ( i = 1 ; i <= 50 ; i++ )
        num[i] = i * i ;
}
```

Here, in the array **num** there is no such element as **num[50]**, since array counting begins with 0 and not 1. Compiler would not give a warning if our program exceeds the bounds. If not taken care of, in extreme cases the above code might even hang the computer.

[14] Forgetting to reserve an extra location in a character array for the null terminator.

Remember each character array ends with a '\0', therefore its dimension should be declared big enough to hold the normal characters as well as the '\0'.

For example, the dimension of the array **word[ ]** should be 9 if a string "Jamboree" is to be stored in it.

[15] Confusing the precedences of the various operators.

```
main( )
{
    char ch ;
    FILE *fp ;

    fp = fopen ( "text.c", "r" ) ;

    while ( ch = getc ( fp ) != EOF )
        putch ( ch ) ;

    fclose ( fp ) ;
}
```

Here, the value returned by **getc( )** will be first compared with EOF, since **!=** has a higher priority than =. As a result, the value that is assigned to **ch** will be the true/false result of the test—1 if the value returned by **getc( )** is not equal to **EOF**, and 0 otherwise. The correct form of the above **while** would be,

```
while ( ( ch = getc ( fp ) ) != EOF )
    putch ( ch ) ;
```

[16] Confusing the operator -> with the operator **.** while referring to a structure element.

Remember, on the left of the operator **.** only a structure variable can occur, whereas on the left of the operator -> only a pointer to a structure can occur. Following example demonstrates this.

```
main( )
```

```
{
    struct emp
    {
        char name[35] ;
        int age ;
    } ;
    struct emp e = { "Dubhashi", 40 } ;
    struct emp *ee ;

    printf ( "\n%d", e.age ) ;
    ee = &e ;
    printf ( "\n%d", ee->>age ) ;
}
```

[17] Forgetting to use the **far** keyword for referring memory locations beyond the data segment.

```
main( )
{
    unsigned int *s ;

    s = 0x413 ;
    printf ( "\n%d", *s ) ;
}
```

Here, it is necessary to use the keyword **far** in the declaration of variable **s,** since the address that we are storing in **s** (0x413) is a address of location present in BIOS Data Area, which is far away from the data segment. Thus, the correct declaration would look like,

```
unsigned int far *s ;
```

The far pointers are 4-byte pointers and are specific to DOS. Under Windows every pointer is 4-byte pointer.

[18] Exceeding the range of integers and chars.

```
main( )
{
    char ch ;

    for ( ch = 0 ; ch <= 255 ; ch++ )
        printf ( "\n%c %d", ch, ch ) ;
}
```

Can you believe that this is an indefinite loop? Probably, a closer look would confirm it. Reason is, **ch** has been declared as a **char** and the valid range of **char** constant is -128 to +127. Hence, the moment **ch** tries to become 128 (through **ch++**), the value of character range is exceeded, therefore the first number from the negative side of the range, -128, gets assigned to **ch**. Naturally the condition is satisfied and the control remains within the loop externally.

D Hexadecimal Numbering

- Numbering Systems
- Relation Between Binary and Hex

While working with computers we are often required to use hexadecimal numbers. The reason for this is—everything a computer does is based on binary numbers, and hexadecimal notation is a convenient way of expressing binary numbers. Before justifying this statement let us first discuss what numbering systems are, why computers use binary numbering system, how binary and hexadecimal numbering systems are related and how to use hexadecimal numbering system in everyday life.

Numbering Systems

When we talk about different numbering systems we are really talking about the base of the numbering system. For example, binary numbering system has base 2 and hexadecimal numbering system has base 16, just the way decimal numbering system has base 10. What in fact is the 'base' of the numbering system? Base represents number of digits you can use before you run out of digits. For example, in decimal numbering system, when we have used digits from 0 to 9, we run out of digits. That's the time we put a 1 in the column to the left - the ten's column - and start again in the one's column with 0, as shown below:

```
0
1
2
3
4
5
6
7
8
9    last available digit
10   start using a new column
11
12
13
```

14
...
...

Since decimal numbering system is a base 10 numbering system any number in it is constructed using some combination of digits 0 to 9. This seems perfectly natural. However, the choice of 10 as a base is quite arbitrary, having its origin possibly in the fact that man has 10 fingers. It is very easy to use other bases as well. For example, if we wanted to use base 8 or octal numbering system, which uses only eight digits (0 to 7), here's how the counting would look like:

0
1
2
3
4
5
6
7 last available digit
10 start using a new column
11
12
...
...

Similarly, a hexadecimal numbering system has a base 16. In hex notation, the ten digits 0 through 9 are used to represent the values zero through nine, and the remaining six values, ten through fifteen, are represented by symbols A to F. The hex digits A to F are usually written in capitals, but lowercase letters are also perfectly acceptable. Here is how the counting in hex would look like:

0

```
2
3
4
5
6
7
8
9
A
B
C
D
E
F   last available digit
10  start using a new column
11
...
...
```

Many other numbering systems can also be imagined. For example, we use a base 60 numbering system, for measuring minutes and seconds. From the base 12 system we retain our 12 hour system for time, the number of inches in a foot and so on. The moral is that any base can be used in a numbering system, although some bases are convenient than others.

The hex numbers are built out of hex digits in much the same way the decimal numbers are built out of decimal digits. For example, when we write the decimal number 342, we mean,

```
  3 times 100 (square of 10)
+ 4 times 10
+ 2 times 1
```

Similarly, if we use number 342 as a hex number, we mean,

```
  3 times 256 (square of 16)
```

+ 4 times 16
+ 2 times 1

Relation Between Binary and Hex

As it turns out, computers are more comfortable with binary numbering system. In a binary system, there are only two digits 0 and 1. This means you can't count very far before you need to start using the next column:

0
1 last available digit
10 start using a new column
11
...
...

Binary numbering system is a natural system for computers because each of the thousands of electronic circuits in the computer can be in one of the two states—on or off. Thus, binary numbering system corresponds nicely with the circuits in the computer—0 means off, and 1 means on. 0 and 1 are called bits, a short-form of binary digits.

Hex numbers are used primarily as shorthand for binary numbers that the computers work with. Every hex digit represents four bits of binary information (Refer Figure D.1). In binary numbering system 4 bits taken at a time can give rise to sixteen different numbers, so the only way to represent each of these sixteen 4-bit binary numbers in a simple and short way is to use a base sixteen numbering system.

Suppose we want to represent a binary number 11000101 in a short way. One way is to find it decimal equivalent by multiplying each binary digit with an appropriate power of 2 as shown below:

$1*2^7+1*2^6+0*2^5+0*2^4+0*2^3+1*2^2+0*2^1+1*2^0$

which is equal to 197.

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Figure D.1

Another method is much simpler. Just look at Figure D.1. From it find out the hex digits for the two four-bit sets (1100 and 0101). These happen to be C and 5. Therefore, the binary number's hex equivalent is C5. You would agree this is a easier way to represent the binary number than to find its decimal equivalent. In this method neither multiplication nor addition is needed. In fact, since there are only 16 hex digits, it's fairly easy to memorize the binary equivalent of each one. Quick now, what's binary 1100 in hex? That's right C. You are already getting the feel of it. With a little practice it is easy to translate even long numbers into hex. Thus, 1100 0101 0011 1010 binary is C53A hex.

As it happens with many unfamiliar subjects, learning hexadecimal requires a little practice. Try your hand at converting some binary numbers and vice versa. Soon you will be talking hexadecimal as if you had known it all your life.

E ASCII Chart

There are 256 distinct characters used by IBM compatible family of microcomputers. Their values range from 0 to 255. These can be grouped as under:

Character Type	No. of Characters
Capital letters	26
Small-case Letters	26
Digits	10
Special Symbols	32
Control Character	34
Graphics Character	128
Total	256

Figure E.1

Out of the 256 character set, the first 128 are often called ASCII characters and the next 128 as Extended ASCII characters. Each ASCII character has a unique appearance. The following simple program can generate the ASCII chart:

```
main( )
{
    int ch ;

    for ( ch = 0 ; ch <<= 255 ; ch++ )
        printf ( "%d %c\n", ch, ch ) ;
}
```

This chart is shown on the following page. Out of the 128 graphic characters (Extended ASCII characters), there are characters that are used for drawing single line and double line boxes in text mode. For convenience these characters are shown in Figure E.2.

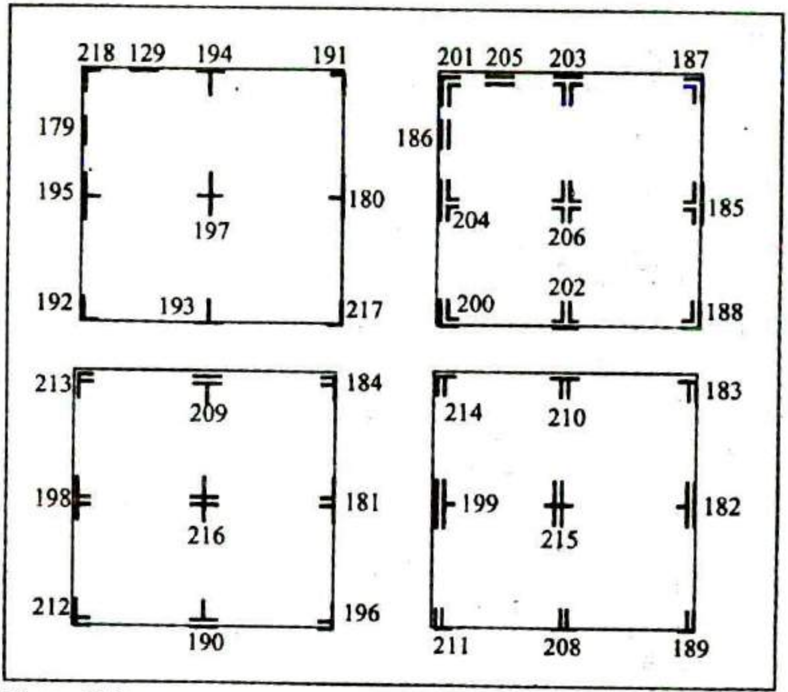
218 129 194 191
179
195
197
180
192 193 217
201 205 203 187
186
204
206
185
202
200
188
213 184
209
198
216
181
212 196
190
183
214 210
199
215
182
211 208 189

Figure E.2

Value	Char	Value	Char	Value	Char	Value	Char	Value	Char	Value	Char
0		22	▬	44	.	66	R	88	X	110	n
1	☺	23	↨	45	-	67	C	89	Y	111	o
2	●	24	↑	46	.	68	D	90	Z	112	p
3	♥	25	↓	47	/	69	E	91	[	113	q
4	♦	26	→	48	0	70	F	92	\	114	r
5	♣	27	←	49	1	71	G	93	]	115	s
6	♠	28	⌐	50	2	72	H	94	^	116	t
7	•	29	↔	51	3	73	I	95		117	u
8	◘	30	▲	52	4	74	J	96	.	118	v
9	○	31	▼	53	5	75	K	97	a	119	ẇ
10	■	32		54	6	76	L	98	b	120	x
11	♂	33	!	55	7	77	M	99	c	121	y
12	♀	34	"	56	8	78	N	100	d	122	z
13	♪	35	#	57	9	79	O	101	e	123	{
14	♫	36	$	58	:	80	P	102	f	124	|
15	☼	37	%	59	;	81	Q	103	g	125	}
16	►	38	&	60	<	82	R	104	h	126	~
17	◄	39	'	61	=	83	S	105	i	127	"H
18	↕	40	(	62	>	84	T	106	j	128	Ç
19	‼	41	)	63	?	85	U	107	k	129	ü
20	¶	42	*	64	@	86	V	108	l	130	é
21	§	43	+	65	A	87	W	109	m	131	â

Value	Char	Value	Char	Value	Char	Value	Char	Value	Char	Value	Char
132	ä	154	Ü	176	░	198	╞	220	▄	242	≥
133	à	155	¢	177	▒	199	╟	221	▌	243	≤
134	å	156	£	178	▓	200	╚	222	▐	244	⌠
135	ç	157	¥	179	│	201	╔	223	▀	245	⌡
136	ê	158	₧	180	┤	202	╩	224	α	246	÷
137	ë	159	ƒ	181	╡	203	╦	225	ß	247	≈
138	è	160	á	182	╢	204	╠	226	Γ	248	°
139	ï	161	í	183	╖	205	═	227	π	249	•
140	î	162	ó	184	╕	206	╬	228	Σ	250	·
141	ì	163	ú	185	╣	207	╧	229	σ	251	√
142	Ä	164	ñ	186	║	208	╨	230	µ	252	η
143	Å	165	Ñ	187	╗	209	╤	231	τ	253	²
144	É	166	ª	188	╝	210	╥	232	Φ	254	■
145	æ	167	º	189	╜	211	╙	233	θ	255	
146	Æ	168	¿	190	╛	212	╘	234	Ω		
147	ô	169	⌐	191	┐	213	╒	235	δ		
148	ö	170	¬	192	└	214	╓	236	∝		
149	ò	171	½	193	┴	215	╫	237	ø		
150	û	172	¼	194	┬	216	╪	238	ε		
151	ù	173	¡	195	├	217	┘	239	∩		
152	ÿ	174	«	196	─	218	┌	240	≡		
153	Ö	175	»	197	┼	219	█	241	±		

F Helper.h File

```
LRESULT CALLBACK WndProc ( HWND, UINT, WPARAM, LPARAM ) ;

HINSTANCE hInst ; // current instance

/* FUNCTION: InitInstance ( HANDLE, int
   PURPOSE: Saves instance handle and creates main window
   COMMENTS: In this function, we save the instance handle in a global
             variable and create and display the main program window.
*/
BOOL InitInstance ( HINSTANCE hInstance, int nCmdShow, char* pTitle )
{
    char classname[ ] = "MyWindowClass" ;
    HWND hWnd ;

    WNDCLASSEX wcex ;
    wcex.cbSize          = sizeof ( WNDCLASSEX ) ;
    wcex.style           = CS_HREDRAW | CS_VREDRAW ;
    wcex.lpfnWndProc     = ( WNDPROC ) WndProc ;
    wcex.cbClsExtra      = 0 ;
    wcex.cbWndExtra      = 0 ;
    wcex.hInstance       = hInstance ;
    wcex.hIcon           = NULL ;
    wcex.hCursor         = LoadCursor ( NULL, IDC_ARROW ) ;
    wcex.hbrBackground   = ( HBRUSH )( COLOR_WINDOW + 1 ) ;
    wcex.lpszMenuName    = NULL ;
    wcex.lpszClassName   = classname ;
    wcex.hIconSm         = NULL ;

    if ( !RegisterClassEx ( &wcex ) )
        return FALSE ;

    hInst = hInstance ; // Store instance handle in our global variable

    hWnd = CreateWindow ( classname, pTitle,
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL,
                NULL, hInstance, NULL ) ;
    if ( !hWnd )
```

```
        return FALSE ;

    ShowWindow ( hWnd, nCmdShow ) ;
    UpdateWindow ( hWnd ) ;

    return TRUE ;
}
```

G Boot Parameters

The disk drives in DOS and Windows are organized as zero-based drives. That is, drive A is drive number 0, drive B is drive number 1, drive C is drive number 2, etc. The hard disk drive can be further partitioned into logical partitions. Each drive consists of four logical parts—Boot Sector, File Allocation Table (FAT), Directory and Data space. When a file/directory is created on the disk, instead of allocating a sector for it, a group of sectors is allocated. This group of sectors is often known as a **cluster**. How many sectors together form one cluster depends upon the capacity of the disk. As the capacity goes on increasing, so also does the maximum cluster number. Accordingly, we have 12-bit, 16-bit or 32-bit FAT. In a 12-bit FAT each entry is of 12 bits. Since each entry in FAT represents a cluster number, the maximum cluster number possible in a 12-bit FAT is 2^{12} (4096). Similarly, in case of a 16-bit FAT the maximum cluster number is 2^{16} (65536). Also, for a 32-bit FAT the maximum cluster number is 2^{28} (268435456. Only 28 of the 32 bits are used in this FAT). All FAT systems are not supported by all versions of DOS and Windows. For example, the 32-bit FAT system is supported only in Win 95 OSR2 version or later. There are differences in the organization of contents of Boot Sector, FAT and Directory in FAT12/FAT16 system on one hand and FAT32 on the other.

In Chapter 19 Figure 19.6 we saw the breakup of the contents of the boot sector of a 12-bit FAT. Given below are the contents of a boot sector of 16-bit FAT and a 32-bit FAT.

Description	**Length**	**Typical Values**
Jump instruction	3	EB3C90
OEM name	8	MSWIN4.1
Bytes per sector	2	512
Sectors per cluster	1	64
Reserved sectors	2	1
Number of FAT copies	1	2
Max. Root directory entries	2	512
Total sectors	2	0
Media descriptor	1	F8
Sectors per FAT	2	256
Sectors per track	2	63
No. of sides	2	255
Hidden sectors	4	63
Huge sectors	4	4192902
BIOS drive number	1	128
Reserved sectors	1	1
Boot signature	1	41
Volume ID	4	4084677574
Volume label	11	ICIT
File system type	8	FAT16

Figure G.1

Let us now take a look at the 32-bit FAT system's boot sector contents. These are shown in Figure G.2.

Description	Length	Typical Values
Jump instruction	3	EB5890
OEM name	8	MSWIN4.1
Bytes per sector	2	512
Sectors per cluster	1	8
Reserved sectors	2	51
Number of FAT copies	1	2
Root directory entries	2	0
Total sectors	2	0
Media descriptor	1	F8
Sectors per FAT	2	0
Sectors per track	2	63
No. of sides	2	255
Hidden sectors	2	63
High word of hidden sectors	4	63
Huge sectors	4	4192902
High word of huge sectors	2	4192902
Sectors per FAT	2	4095
High word of sectors per FAT	2	4095
Drive description flag	2	0
File system version	2	0
Root directory starting cluster	2	2
High word of root directory starting cluster	2	2
File system information sector	2	1
Back up boot sector	2	6
Reserved	6	0

continued...

...continued

BIOS drive number	1	128
Reserved	1	0
Boot signature	1	41
Volume ID	4	649825316
Volume label	11	ICIT
File system type	8	FAT32

Figure G.2

There are significant changes in the contents of the boo t sector of a 32-bit FAT system. The entries 'Number of hidden sectors' and 'Huge sectors' have now been made 4-byte entries. The first two bytes contain the low word of the value, whereas, the next two bytes contain the high word value.

The number of sectors per FAT in a 32-bit file system is likely to exceed what can be accommodated in two bytes. Hence the entry 'Sectors per FAT' for a disk with a 32-bit file system would typically have a value 0. The value of 'Sectors per FAT' is now stored as a 4-byte entity, with the similar arrangement of low word and high word as discussed earlier.

The boot sector of a 32-bit FAT system also has new entries like 'Drive description flag', 'File system version' 'Starting cluster number of the root directory', 'Sector number of the file system information sector', and the sector number of the 'Backup boot sector'.

The 'Drive description flag' is a two-byte entity. Bit 8 of this flag indicates whether or not the information written to the active FAT will be written to all copies of the FAT. The low four bits of this entry contains the 0-based FAT number of the active FAT. These bits are meaningful only if bit 8 is set.

In the entry 'File system version number' the high byte contains the major version number, whereas, the low byte contains the minor version number.

The entry 'File system information sector' contains a value indicating the sector number where the file system information is present. This file system information consists of the fields shown in Figure G.3.

Description	Length
File system signature	4
Total number of free clusters	4
Sector number of the next free cluster	4
Reserved	6

Figure G.3

The entry 'File information sector' contains a value OFFFFh if there is no such sector. The entry 'Backup boot sector' contains a value 0FFFFh is there is no backup boot sector. Otherwise this value is any non-zero value less than the reserved sector count.

H Linux Installation

This appendix gives the steps that are to be carried out for installing Red Hat Linux 9.0. In addition I have also indicated a few commands that are necessary to compile and execute the programs given in Chapters 20 and 21. Follow the steps mentioned below to carry out the installation.

(a) Configure the system to boot from CDROM drive.
(b) Insert the first CD in the drive and boot the system from it.
(c) Hit 'Enter' key when the 'boot' prompt appears.
(d) Select the 'Skip' option in the "CD Found" dialog box.
(e) Click on the 'Next' button in the 'Welcome' screen.
(f) Click on the 'Next' button in the 'Language selection' screen.
(g) Click on the 'Next' button in the 'Keyboard' screen.
(h) Click on the 'Next' button in the 'Mouse Configuration' screen.
(i) Select the 'Custom' option in the 'Installation Type' screen and then click on the 'Next' button.
(j) Click on the 'Next' button in the 'Disk Partitioning Setup' screen.
(k) Select the 'Keep all partitions and use existing free space' option in the 'Automatic Partitioning' screen and then click on the 'Next' button. Ignore any warnings generated by clicking on the 'OK' button.
(l) Click on the 'Next' button in the 'Boot loader configuration' screen.
(m) Click on the 'Next' button in the 'Network configuration' screen.
(n) Click on the 'Next' button in the 'Firewall configuration' screen.
(o) Click on the 'Next' button in the 'additional language support' screen.
(p) Select a suitable option in the 'Time zone offset' screen and click on the 'Next' button.
(q) Type a password for the root account in the 'Set root password' screen and then click on the 'Next' button.
(r) Click on the 'Next' button in the 'Authentication configuration' screen.

(s) In the 'Package group selection' screen make sure that you select the following options—X window system, K desktop environment, Development tools, GNOME software development and then click on the 'Next' button.
(t) Select 'No' option in the 'Boot diskette creation' screen
(u) Click on the 'Next' button in the 'Graphical Interface (x) configuration' screen.
(v) Click on the 'Next' button in the 'Monitor configuration' screen.
(w) In the 'Customize graphical configuration' screen select the 'Graphical' option and then click on 'Next' button.
(x) Once the system restarts configure the system to boot from Hard Disk.

Using Red Hat Linux

For logging into the system enter the username and password and select the session as KDE (K Desktop Environment). Once you have logged in, to start typing the program use the following menu options:

KMenu | Run Command

A dialog would now pop up. In this dialog in the command edit box type **KWrite** and then click on the Ok button. Now you can type the program and save it.

To compile the program you need to go the command prompt. This can be done using the following menu option.

KMenu | System Tools | Terminal

Once at the command prompt you can use the **gcc** compiler to compile and execute your programs. You can launch another instance of the command prompt by repeating the step mentioned above.

Index

D

E

F

G

H

I

J

K

L

M

R

S

T

U

V

W

X