x. This would overwrite the earlier value 3, since a memory location can hold only one value at a time. This is shown in Figure 1.3.

x = 3 x = 5

Figure 1.3

x 3 x 5

Since the location whose name is **x** can hold different values at different times **x** is known as a variable. As against this, 3 or 5 do not change, hence are known as constants.

Types of C Constants

C constants can be divided into two major categories:

(a) Primary Constants

(b) Secondary Constants

These constants are further categorized as shown in Figure 1.4.

Figure 1.4

C Constants

Primary Constants Secondary Constants

Integer Constant

Real Constant

Character Constant

Array

Pointer

Structure

Union

Enum, etc.

At this stage we would restrict our discussion to only Primary Constants, namely, Integer, Real and Character constants. Let us see the details of each of these constants. For constructing these different types of constants certain rules have been laid down. These rules are as under:

Rules for Constructing Integer Constants

(a) An integer constant must have at least one digit.

(b) It must not have a decimal point.

(c) It can be either positive or negative.

(d) If no sign precedes an integer constant it is assumed to be positive.

(e) No commas or blanks are allowed within an integer constant.

(f) The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the

```
float r, si ;
printf ( "Enter values of p, n, r" ) ;
scanf ( "%d %d %f", &p, &n, &r ) ;
si = p * n * r / 100 ;
printf ( "%f" , si ) ;
}
```


The first **printf( )** outputs the message 'Enter values of p, n, r' on the screen. Here we have not used any expression in **printf( )** which means that using expressions in **printf( )** is optional.

Note that the ampersand (**&**) before the variables in the **scanf( )** function is a must. **&** is an 'Address of' operator. It gives the location number used by the variable in memory. When we say **&a**, we are telling **scanf( )** at which memory location should it store the value supplied by the user from the keyboard. The detailed working of the **&** operator would be taken up in Chapter 5.

Note that a blank, a tab or a new line must separate the values supplied to **scanf( )**. Note that a blank is creating using a spacebar, tab using the Tab key and new line using the Enter key. This is shown below:

Ex.: The three values separated by blank

```
1000 5 15.5
```

Ex.: The three values separated by tab.

```
1000	5	15.5
```

Ex.: The three values separated by newline.

```
1000
5
15.5
```

So much for the tips. How about another program to give you a feel of things...

```
/* Just for fun. Author: Bozo */
main( )
{
int num ;
printf ( "Enter a number" ) ;
```



```
scanf ( "%d", &num ) ;
printf ( "Now I am letting you on a secret..." ) ;
printf ( "You have just entered the number %d", num ) ;
}
```

C Instructions

Now that we have written a few programs let us look at the instructions that we used in these programs. There are basically three types of instructions in C:

(a)
(b)
(c)
(a)
(b)

a = 3 / 2 * 5 ;
Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity. Figure 1.10 shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
/	3	2 or 2 * 5	Left operand is unambiguous, Right is not
*	3 / 2 or 2	5	Right operand is unambiguous, Left is not

Figure 1.10

Since both / and * have L to R associativity and only / has unambiguous left operand (necessary condition for L to R associativity) it is performed earlier.
Consider one more expression
a = b = 3 ;
Here both assignment operators have the same priority and same associativity (Right to Left). Figure 1.11 shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
=	a	b or b = 3	Left operand is unambiguous, Right is not
=	b or a = b	3	Right operand is unambiguous, Left is not

Figure 1.11

Since both = have R to L associativity and only the second = has unambiguous right operand (necessary condition for R to L associativity) the second = is performed earlier.
Consider yet another expression
z = a * b + c / d ;
Here ***** and / enjoys same priority and same associativity (Left to Right). Figure 1.12 shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
*	a	b	Both operands are unambiguous
/	c	d	Both operands are unambiguous

Figure 1.12

Here since left operands for both operators are unambiguous Compiler is free to perform ***** or / operation as per its convenience

since no matter which is performed earlier the result would be same.
Appendix A gives the associativity of all the operators available in C.

Now let us learn each of these and their variations in turn.

The *if* Statement

Like most languages, C uses the keyword **if** to implement the decision control instruction. The general form of **if** statement looks like this:

if (this condition is true)
execute this statement ;

The keyword **if** tells the compiler that what follows is a decision control instruction. The condition following the keyword **if** is always enclosed within a pair of parentheses. If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed; instead the program skips past it. But how do we express the condition itself in C? And how do we evaluate its truth or falsity? As a general rule, we express a condition using C's 'relational' operators. The relational operators allow us to compare two values to see whether they are equal to each other, unequal, or whether one is greater than the other. Here's how they look and how they are evaluated in C.

this expression is true if
x == y x is equal to y
x != y x is not equal to y
x < y x is less than y
x > y x is greater than y
x <= y x is less than or equal to y
x >= y x is greater than or equal to y
Figure 2.1

The relational operators should be familiar to you except for the equality operator == and the inequality operator !=. Note that = is used for assignment, whereas, == is used for comparison of two quantities. Here is a simple program, which demonstrates the use of **if** and the relational operators.

```
/* Demonstration of if statement */
main( )
{
int num ;
printf ( "Enter a number less than 10 " ) ;
scanf ( "%d", &num ) ;
if ( num <= 10 )
printf ( "What an obedient servant you are !" ) ;
}
```

On execution of this program, if you type a number less than or equal to 10, you get a message on the screen through **printf( )**. If you type some other number the program doesn't do anything. The following flowchart would help you understand the flow of control in the program.

INPUT num
is

```
int i ;
printf ( "Enter value of i " ) ;
scanf ( "%d", &i ) ;
if ( i = 5 )
printf ( "You entered 5" ) ;
else
printf ( "You entered something other than 5" ) ;
}
```

And here is the output of two runs of this program...

```
Enter value of i 200
You entered 5
Enter value of i 9999
You entered 5
```

Surprising? You have entered 200 and 9999, and still you find in either case the output is 'You entered 5'. This is because we have written the condition wrongly. We have used the assignment operator = instead of the relational operator ==. As a result, the condition gets reduced to **if (5)**, irrespective of what you supply as the value of **i**. And remember that in C 'truth' is always nonzero, whereas 'falsity' is always zero. Therefore, **if (5)** always evaluates to true and hence the result.

Another common mistake while using the **if** statement is to write a semicolon (**;**) after the condition, as shown below:

```
main( )
{
int i ;
printf ( "Enter value of i " ) ;
scanf ( "%d", &i ) ;
if ( i == 5 ) ;
printf ( "You entered 5" ) ;
}
```


The ; makes the compiler to interpret the statement as if you have written it in following manner:

```
if ( i == 5 )
;
printf ( "You entered 5" ) ;
```

Here, if the condition evaluates to true the ; (null statement, which does nothing on execution) gets executed, following which the **printf( )** gets executed. If the condition fails then straightaway the **printf( )** gets executed. Thus, irrespective of whether the condition evaluates to true or false the **printf( )** is bound to get executed. Remember that the compiler would not point out this as an error, since as far as the syntax is concerned nothing has gone wrong, but the logic has certainly gone awry. Moral is, beware of such pitfalls.

The following figure summarizes the working of all the three logical operators.

Operands		**Results**			
x	**y**	**!x**	**!y**	**x && y**	**x || y**

Any character is entered through the keyboard, write a program to determine whether the character entered is a capital letter, a small case letter, a digit or a special symbol. The following table shows the range of ASCII values for various characters.

Characters ASCII Values

A – Z
a – z
0 – 9
special symbols
65 – 90
97 – 122
48 – 57
0 - 47, 58 - 64, 91 - 96, 123 - 127

(c) An Insurance company follows following rules to calculate premium.

(1) If a person's health is excellent and the person is between 25 and 35 years of age and lives in a city and is a male then the premium is Rs. 4 per thousand and his policy amount cannot exceed Rs. 2 lakhs.

(2) If a person satisfies all the above conditions except that the sex is female then the premium is Rs. 3 per thousand and her policy amount cannot exceed Rs. 1 lakh.

(3) If a person's health is poor and the person is between 25 and 35 years of age and lives in a village and is a male

then the premium is Rs. 6 per thousand and his policy cannot exceed Rs. 10,000.

(4) In all other cases the person is not insured.

Write a program to output whether the person should be insured or not, his/her premium rate and maximum amount for which he/she can be insured.

(d)

(e)

A certain grade of steel is graded according to the following conditions:

(i) Hardness must be greater than 50
(ii) Carbon content must be less than 0.7
(iii) Tensile strength must be greater than 5600

The grades are as follows:

Grade is 10 if all three conditions are met
Grade is 9 if conditions (i) and (ii) are met
Grade is 8 if conditions (ii) and (iii) are met
Grade is 7 if conditions (i) and (iii) are met
Grade is 6 if only one condition is met
Grade is 5 if none of the conditions are met

Write a program, which will require the user to give values of hardness, carbon content and tensile strength of the steel under consideration and output the grade of the steel.

A library charges a fine for every book returned late. For first 5 days the fine is 50 paise, for 6-10 days fine is one rupee and

example,
while (i <= 10)
i = i + 1
is same as
while (i <= 10
{
i = i + 1 ;
}
indefinitely.
main( )
{
int i = 1 ;
while (
printf ("
}

p, since **i** remains equal to 1 forever.
e correct form would be as under:
int i = 1 ;
i <= 10)
{
"%d\n", i) ;
− enting a loop counter, we can even decrement
manage to get the body of the loop executed
repeatedly. This is shown below:
i >= 1)
{
"\nMake the computer literate!") ;
− that a loop counter must only be an int. It
a float.
a <= 10.5)
{
indrops on roses...") ;
skers on kittens") ;
This is an indefinite loo
Th
main( )
{
while (
printf (
i = i + 1 ;
}
}
Instead of increm
it and still
main( )
{
int i = 5 ;
while (
printf (

i = i + 1 ;
}
}

another indefinite loop, and it doesn't give any output all. The reason is, we have carelessly given a **;** after the **while**. This would make the loop work like this...

{
", i) ;
1 ;

in alue of **i** is not getting incremented the control ould keep rotating within the loop, eternally. Note that enclosing **printf( )** and **i = i +1** within a pair of braces is not

Mo

h are frequently used with ir usage Complete Guide To Consider a problem wherein numbers from 1 to 10 are to be printed on the screen. The program

int i = 1 ;
i <= 10)
{
"%d\n", i) ;

This is at

while (i <= 10)
;
printf ("%d\n
i = i +
}

S ce the v w an error. In fact we can put a pair of braces around any individual statement or set of statements without affecting the execution of the program.

re Operators

There are variety of operators whic **while**. To illustrate the for performing this task can be written using **while** in the following different ways:

(a) main( )
{
while (
printf (
i = i + 1 ;
}
}

(b)
int i = 1 ;
i <= 10)

{
"%d\n", i) ;
at the increment operator ++ increments the value of **i**
1, every time the statement **i++** gets executed. Similarly, to
reduce the value of a variable by 1 a decrement operator -- is
+++.
(c)
int i = 1 ;
i <= 10)
{
"%d\n", i) ;
hat += is a compound assignment operator. It
crements the value of **i** by 1. Similarly, **j = j + 10** can also
be written as **j += 10**. Other compound assignment operators
(d)
int i = 0 ;
(i++ < 10)
main( )
{
while (
printf (
i++ ;
}
}
Note th
by
also available.
However, never use **n+++** to incr[illegible] the value of **n** by 2,
since C doesn't recognize the operator
main( )
{
while (
printf (
i += 1 ;
}
}
Note t
in
are **-=, *=, / =** and **%=**.
main( )
{
while

printf ("%d\n", i) ;
i++ < 10), firstly the comparison of
lue of **i** with 10 is performed, and then the incrementation
of **i** takes place. Since the incrementation of **i** happens after its
(e)
int i = 0 ;
(++i <= 10)

In this program the moment **num % i** turns out to be zero, (i.e. **num** is exactly divisible by **i**) the message "Not a prime number" is printed and the control breaks out of the **while** loop. Why does the program require the **if** statement after the **while** loop at all? Well, there are two ways the control could have reached outside the **while** loop:
(a)
(b)
It jumped out because the number proved to be not a prime.
The loop came to an end because the value of **i** became equal to **num**.
When the loop terminates in the second case, it means that there was no number between 2 to **num - 1** that could exactly divide **num**. That is, **num** is indeed a prime. If this is true, the program should print out the message "Prime number".
The keyword **break**, breaks the control only from the **while** in which it is placed. Consider the following program, which illustrates this fact.

```
main( )
{
int i = 1 , j = 1 ;
while ( i++ <= 100 )
{
while ( j++ <= 200 )
{
if ( j == 150 )
break ;
else
printf ( "%d %d\n", i, j ) ;
}
}
}
```


In this program when **j** equals 150, **break** takes the control outside the inner **while** only, since it is placed inside the inner **while**.

The *continue* Statement

In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop.
A **continue** is usually associated with an **if**. As an example, let's consider the following program.

```
main( )
{
int i, j ;
for ( i = 1 ; i <= 2 ; i++ )
{
for ( j = 1 ; j <= 2 ; j++ )
{
```

required to make a choice between a number of alternatives rather than only one or two. For example, which school to join or which hotel to visit or still harder which girl to marry (you almost always end up making a wrong decision is a different matter altogether!). Serious C programming is same; the choice we are asked to make is more complicated than merely selecting between two alternatives. C provides a special control statement that allows us to handle such cases effectively; rather than using a series of **if** statements. This control instruction is in fact the topic of this chapter. Towards the end of the chapter we would also study a keyword called **goto**, and understand why we should avoid its usage in C programming.

I

Decisions Using *switch*

The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switchcase-default**, since these three keywords go together to make up
the control statement. They most often appear as follows:

```
switch ( integer expression )
{
case constant 1 :
do this ;
case constant 2 :
do this ;
case constant 3 :
do this ;
default :
do this ;
}
```

The integer expression following the keyword **switch** is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an

integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others. The "do this" lines in the above form of **switch** represent any valid C statement.

What happens when we run a program containing a **switch**? First, the integer expression following the keyword **switch** is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statements. When a match is found, the program executes the statements following that **case**, and all subsequent **case** and **default** statements as well. If no match is found with any of the **case** statements, only the statements following the **default** are executed. A few examples will show how this control structure works.

Cases can never have variable expressions (for example it is wrong to say **case a +3 :**)
Multiple cases cannot use same expressions. Thus the following **switch** is illegal:


```
switch ( a )
{
case 3 :
...
case 1 + 2 :
...
}
```

(a), (b) and (c) above may lead you to believe that these are obvious disadvantages with a **switch**, especially since there weren't any such limitations with **if-else**. Then why use a **switch** at all? For speed—**switch** works faster than an equivalent **if-else** ladder. How come? This is because the compiler generates a jump table for a **switch** during compilation. As a result, during execution it simply refers the jump table to decide which case should be executed, rather than actually checking which case is satisfied. As against this, **if-else**s are slower because they are evaluated at execution time. A **switch** with 10 cases would work faster than an equivalent **if-else** ladder. Also, a **switch** with 2 **case**s would work slower than **if-else** ladder. Why? If the 10th **case** is satisfied then jump table would be referred and statements for the 10th **case** would be executed. As against this, in an **if-else** ladder 10 conditions would be evaluated at execution time, which makes it slow. Note that a lookup in the jump table is faster than evaluation of a condition, especially if the condition is complex.
If on the other hand the conditions in the **if-else** were simple and less in number then **if-else** would work out faster than the lookup mechanism of a **switch**. Hence a **switch** with two **case**s would work slower than an equivalent **if-else**. Thus, you as a programmer should take a decision which of the two should be used when.

The *goto* Keyword

Avoid **goto** keyword! They make a C programmer's life miserable. There is seldom a legitimate reason for using **goto**, and its use is

one of the reasons that programs become unreliable, unreadable, and hard to debug. And yet many programmers find **goto** seductive.
In a difficult programming situation it seems so easy to use a **goto** to take the control where you want. However, almost always, there is a more elegant way of writing the same program using **if**, **for**, **while** and **switch**. These constructs are far more logical and easy to understand.
The big problem with **goto**s is that when we do use them we can never be sure how we got to a certain point in our code. They obscure the flow of control. So as far as possible skip them. You

can always get the job done without them. Trust me, with good programming skills **goto** can always be avoided. This is the first and last time that we are going to use **goto** in this book. However, for sake of completeness of the book, the following program shows how to use **goto**.

```
main( )
{
int goals ;
printf ( "Enter the number of goals scored against India" ) ;
scanf ( "%d", &goals ) ;
if ( goals <= 5 )
goto sos ;
else
{
printf ( "About time soccer players learnt C\n" ) ;
printf ( "and said goodbye! adieu! to soccer" ) ;
exit( ) ; /* terminates program execution */
}
sos :
printf ( "To err is human!" ) ;
```



```
}
```

And here are two sample runs of the program...

```
Enter the number of goals scored against India 3
To err is human!
Enter the number of goals scored against India 7
About time soccer players learnt C
and said goodbye! adieu! to soccer
```

A few remarks about the program would make things clearer.

− If the condition is satisfied the **goto** statement transfers control to the label 'sos', causing **printf( )** following **sos** to be executed.

− The label can be on a separate line or on the same line as the statement following it, as in,

```
sos : printf ( "To err is human!" ) ;
```

− Any number of **goto**s can take the control to the same label.

− The **exit( )** function is a standard library function which terminates the execution of the program. It is necessary to use this function since we don't want the statement

```
printf ( "To err is human!" )
```

to get executed after execution of the **else** block.

− The only programming situation in favour of using **goto** is when we want to take the control out of the loop that is contained in several other loops. The following program illustrates this.


```
main( )
{
int i, j, k ;
for ( i = 1 ; i <= 3 ; i++ )
```

```
}
(b) main( )
{
int c = 3 ;
switch ( c )
{
case 'v' :
printf ( "I am in case v \n" ) ;
break ;
case 3 :
printf ( "I am in case 3 \n" ) ;
break ;
case 12 :
printf ( "I am in case 12 \n" ) ;
break ;
default :
printf ( "I am in default \n" ) ;
}
```



```
}
(c) main( )
{
int k, j = 2 ;
switch ( k = j + 1 )
{
case 0 :
printf ( "\nTailor") ;
case 1 :
printf ( "\nTutor") ;
case 2
printf ( "\nTramp") ;
default :
printf ( "\nPure Simple Egghead!" ) ;
}
}
(d) main( )
{
int i = 0 ;
switch ( i )
{
case 0 :
printf ( "\nCustomers are dicey" ) ;
case 1 :
printf ( "\nMarkets are pricey" ) ;
case 2 :
printf ( "\nInvestors are moody" ) ;
case 3 :
printf ( "\nAt least employees are good" ) ;
}
}
```

statement 2 ;
statement 3 ;
}

(d)
(e)
(f)

Any function can be called from any other function. Even **main( )** can be called from other functions. For example,

```
main( )
{
message( ) ;
}
message( )
{
printf ( "\nCan't imagine life without C" ) ;
main( ) ;
}
```

A function can be called any number of times. For example,

```
main( )
{
message( ) ;
message( ) ;
}
message( )
{
printf ( "\nJewel Thief!!" ) ;
}
```

The order in which the functions are defined in a program and the order in which they get called need not necessarily be same. For example,

```
main( )
{
message1( ) ;
message2( ) ;
}
message2( )
{
printf ( "\nBut the butter was bitter" ) ;
```



```
}
message1( )
{
printf ( "\nMary bought some butter" ) ;
}
```

Here, even though **message1( )** is getting called before **message2( )**, still, **message1( )** has been defined after **message2( )**. However, it is advisable to define the functions in the same order in which they are called. This makes the program easier to understand.

(g)
(h)
(i)
A function can call itself. Such a process is called 'recursion'. We would discuss this aspect of C functions later in this chapter.
A function can be called from other function, but a function cannot be defined in another function. Thus, the following program code would be wrong, since **argentina( )** is being defined inside another function, **main( )**.

```
main( )
{
printf ( "\nI am in main" ) ;
argentina( )
{
printf ( "\nI am in argentina" ) ;
}
}
```

There are basically two types of functions:
Library functions Ex. **printf( )**, **scanf( )** etc.
User-defined functions Ex. **argentina( )**, **brazil( )** etc.
As the name suggests, library functions are nothing but commonly required functions grouped together and stored in

what is called a Library. This library of functions is present on the disk and is written for us by people who write compilers for us. Almost always a compiler comes with a library of standard functions. The procedure of calling both types of functions is exactly same.

Why Use Functions

Why write separate functions at all? Why not squeeze the entire logic into one function, **main( )**? Two reasons:
(a) Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If later in the program you want to calculate the area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.
(b) Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.
What is the moral of the story? Don't try to cram the entire logic in one function. It is a very bad style of programming. Instead, break

(a) In this program, from the function **main( )** the values of **a**, **b** and **c** are passed on to the function **calsum( )**, by making a call to the function **calsum( )** and mentioning **a**, **b** and **c** in the parentheses:

sum = calsum (a, b, c) ;

In the **calsum( )** function these values get collected in three variables **x**, **y** and **z**:

calsum (x, y, z)
int x, y, z ;

(b) The variables **a**, **b** and **c** are called 'actual arguments', whereas the variables **x**, **y** and **z** are called 'formal arguments'. Any number of arguments can be passed to a function being called. However, the type, order and number of the actual and formal arguments must always be same.

Instead of using different variable names **x**, **y** and **z**, we could have used the same variable names **a**, **b** and **c**. But the compiler would still treat them as different variables since they are in different functions.

(c) There are two methods of declaring the formal arguments. The one that we have used in our program is known as Kernighan and Ritchie (or just K & R) method.

calsum (x, y, z)
int x, y, z ;

Another method is,

calsum (int x, int y, int z)

This method is called ANSI method and is more commonly used these days.

(d) In the earlier programs the moment closing brace (}) of the called function was encountered the control returned to the calling function. No separate **return** statement was necessary to send back the control.

This approach is fine if the called function is not going to return any meaningful value to the calling function. In the above program, however, we want to return the sum of **x**, **y** and **z**. Therefore, it is necessary to use the **return** statement.

The **return** statement serves two purposes:

(1) On executing the **return** statement it immediately transfers the control back to the calling program.

(2) It returns the value present in the parentheses after **return**, to th3e calling program. In the above program the value of sum of three numbers is being returned.

(e) There is no restriction on the number of **return** statements that may be present in a function. Also, the **return** statement need not always be present at the end of the called function. The following program illustrates these facts.

```
fun( )
{
char ch ;
```

The above functions get successfully compiled even though there is a mismatch in the format specifiers and the variables in the list. This is because **printf( )** accepts *variable* number of arguments (sometimes 2 arguments, sometimes 3 arguments, etc.), and even with the mismatch above the call still matches with the prototype of **printf( )** present in 'stdio.h'. At run-time when the first **printf( )** is executed, since there is no variable matching with the last specifier **%d**, a garbage integer gets printed. Similarly, in the second **printf( )** since the format specifier for **j** has not been mentioned its value does not get printed.

Advanced Features of Functions

With a sound basis of the preliminaries of C functions, let us now get into their intricacies. Following advanced topics would be considered here.

(a) Function Declaration and Prototypes
(b) Calling functions by value or by reference
(c) Recursion

Let us understand these features one by one.

Function Declaration and Prototypes

Any C function by default returns an **int** value. More specifically, whenever a call is made to a function, the compiler assumes that this function would return a value of the type **int**. If we desire that a function should return a value other than an **int**, then it is necessary to explicitly mention so in the calling function as well as in the called function. Suppose we want to find out square of a number using a function. This is how this simple program would look like:

```
main( )
{
float a, b ;
printf ( "\nEnter any number " ) ;
scanf ( "%f", &a ) ;
b = square ( a ) ;
printf ( "\nSquare of %f is %f", a, b ) ;
}
square ( float x )
{
float y ;
y = x * x ;
return ( y ) ;
}
```

And here are three sample runs of this program...

```
Enter any number 3
Square of 3 is 9.000000
Enter any number 1.5
Square of 1.5 is 2.000000
Enter any number 2.5
Square of 2.5 is 6.000000
```

want to do is explain the rationale of C's pointer notation.

Pointer Notation

Consider the declaration,

int i = 3 ;

This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i** with this memory location.

(c) Store the value 3 at this location.

We may represent **i**'s location in memory by the following memory map.

location name

3

i

location number

value at location

65524

Figure 5.1

We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.

We can print this address number through the following program:

```
main( )
{
int i = 3 ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nValue of i = %d", i ) ;
}
```

The output of the above program would be:

Address of i = 65524

Value of i = 3

Look at the first **printf( )** statement carefully. '&' used in this statement is C's 'address of' operator. The expression **&i** returns the address of the variable **i**, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer. We have been using the '&' operator all the time in the **scanf( )** statement.

The other pointer operator available in C is '*****', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program:


```
main( )
{
int i = 3 ;
```

```
printf ( "\nEnter any number " ) ;
scanf ( "%d", &a ) ;
fact = rec ( a ) ;
printf ( "Factorial value = %d", fact ) ;
}
rec ( int x )
{
int f ;
if ( x == 1 )
return ( 1 ) ;
else
f = x * rec ( x - 1 ) ;
return ( f ) ;
}
```

And here is the output for four runs of the program

```
Enter any number 1
Factorial value = 1
Enter any number 2
Factorial value = 2
Enter any number 3
```



```
Factorial value = 6
Enter any number 5
Factorial value = 120
```

Let us understand this recursive factorial function thoroughly. In the first run when the number entered through **scanf( )** is 1, let us see what action does **rec( )** take. The value of **a** (i.e. 1) is copied into **x**. Since **x** turns out to be 1 the condition **if (x == 1)** is satisfied and hence 1 (which indeed is the value of 1 factorial) is returned through the **return** statement.

When the number entered through **scanf( )** is 2, the **(x == 1)** test fails, so we reach the statement,

```
f = x * rec ( x - 1 ) ;
```

And here is where we meet recursion. How do we handle the expression **x * rec (x - 1)**? We multiply **x** by **rec (x - 1)**. Since the current value of **x** is 2, it is same as saying that we must calculate the value (2 * rec (1)). We know that the value returned by **rec (1)** is 1, so the expression reduces to (2 * 1), or simply 2. Thus the statement,

```
x * rec ( x - 1 ) ;
```

evaluates to 2, which is stored in the variable **f**, and is returned to **main( )**, where it is duly printed as

```
Factorial value = 2
```

Now perhaps you can see what would happen if the value of **a** is 3, 4, 5 and so on.

In case the value of **a** is 5, **main( )** would call **rec( )** with 5 as its actual argument, and **rec( )** will send back the computed value. But before sending the computed value, **rec( )** calls **rec( )** and waits for a value to be returned. It is possible for the **rec( )** that has just been

function to return without recursive call being executed. If you don't do this and you call the function, you will fall in an indefinite loop, and the stack will keep on getting filled with parameters and the return address each time there is a call. Soon the stack would become full and you would get a run-time error indicating that the stack has become full. This is a very common error while writing recursive functions. My advice is to use **printf( )** statement liberally during the development of recursive function, so that you can watch what is going on and can abort execution if you see that you have made a mistake.

Adding Functions to the Library

Most of the times we either use the functions present in the standard library or we define our own functions and use them. Can we not add our functions to the standard library? And would it make any sense in doing so? We can add user-defined functions to the library. It makes sense in doing so as the functions that are to be added to the library are first compiled and then added. When we use these functions (by calling them) we save on their compilation time as they are available in the library in the compiled form.

Let us now see how to add user-defined functions to the library. Different compilers provide different utilities to add/delete/modify functions in the standard library. For example, Turbo C/C++

compilers provide a utility called 'tlib.exe' (Turbo Librarian). Let us use this utility to add a function **factorial( )** to the library. Given below are the steps to do so:

(a)
(b)
(c)
(d)
(e)

Write the function definition of **factorial( )** in some file, say 'fact.c'.

```
int factorial ( int num )
{
int i, f = 1 ;
for ( i = 1 ; i <= num ; i++ )
f = f * i ;
return ( f ) ;
}
```

Compile the 'fact.c' file using Alt F9. A new file called 'fact.obj' would get created containing the compiled code in machine language.

Add the function to the library by issuing the command

```
C:\>tlib math.lib + c:\fact.obj
```

Here, 'math.lib' is a library filename, + is a switch, which means we want to add new function to library and 'c:\fact.obj' is the path of the '.obj' file.

Declare the prototype of the **factorial( )** function in the header file, say 'fact.h'. This file should be included while calling the

```
message( message ( ) ) ;
}
void message( )
{
printf ( "\nPraise worthy and C worthy are synonyms" ) ;
}
```

[C] Answer the following:

(a) Is this a correctly written function:

```
sqr ( a ) ;
int a ;
{
return ( a * a ) ;
}
```

(b) State whether the following statements are True or False:

1. The variables commonly used in C functions are available to all the functions in a program.
2. To return the control back to the calling function we must use the keyword **return**.
3. The same variable names can be used in different functions without any conflict.
4. Every called function must contain a **return** statement.
5. A function may contain more than one **return** statements.
6. Each **return** statement in a function may return a different value.
7. A function can still be useful even if you don't pass any arguments to it and the function doesn't return any value back.
8. Same names can be used for different functions without any conflict.
9. A function may be called more than once from any other function.
10. It is necessary for a function to return some value.

[D] Answer the following:

(a) Write a function to calculate the factorial value of any integer entered through the keyboard.

(b) Write a function **power (a, b)**, to calculate the value of **a** raised to **b**.

(c) Write a general-purpose function to convert any given year into its roman equivalent. The following table shows the roman equivalents of decimal numbers:

Decimal	Roman	Decimal	Roman
1	i	100	c
5	v	500	d
10	x	1000	m
50	l		

Example:

Roman equivalent of 1988 is mdcccclxxxviii

Roman equivalent of 1525 is mdxxv

To fully define a variable one needs to mention not only its type but also its storage class. In this chapter we would be exploring the different storage classes and their relevance in C programming.

Integers, *long* and *short*

We had seen earlier that the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be –2147483648 to +2147483647. Here a 16-bit compiler means that when it compiles a C program it generates machine language code that is targeted towards working on a 16-bit microprocessor like Intel 8086/8088. As against this, a 32-bit compiler like VC++ generates machine language code that is targeted towards a 32-bit microprocessor like Intel Pentium. Note that this does not mean that a program compiled using Turbo C would not work on 32-bit processor. It would run successfully but at that time the 32-bit processor would work as if it were a 16-bit processor. This happens because a 32-bit processor provides support for programs compiled using 16-bit compilers. If this backward compatibility support is not provided the 16-bit program

would not run on it. This is precisely what happens on the new Intel Itanium processors, which have withdrawn support for 16-bit code.

Remember that out of the two/four bytes used to store an integer the highest bit (16th/32nd bit) is used to store the sign of the integer. This bit is 1 if the number is negative, and 0 if the number is positive.

C offers a variation of the integer data type that provides what are called **short** and **long** integer values. The intention of providing these variations is to provide integers with different ranges wherever possible. Though not a rule, **short** and **long** integers would usually occupy two and four bytes respectively. Each compiler can decide appropriate sizes depending on the operating system and hardware for which it is being written, subject to the following rules:

(a) **short**s are at least 2 bytes big
(b) **long**s are at least 4 bytes big
(c) **short**s are never bigger than **int**s
(d) **int**s are never bigger than **long**s

Figure 6.1 shows the sizes of different integers based upon the OS used.

Compiler	short	int	long
16-bit (Turbo C/C++)	2	2	4
32-bit (Visual C++)	2	4	4

Figure 6.1

long variables which hold **long** integers are declared using the

number of bytes it occupies in memory.
By default all the variables are **signed**. We can declare a variable as **unsigned** to accommodate greater value without increasing the bytes occupied.

(d) We can make use of proper storage classes like **auto**, **register**, **static** and **extern** to control four properties of the variable—storage, default initial value, scope and life.

Exercise

[A] What would be the output of the following programs:

(a)
```
main( )
{
int i ;
for ( i = 0 ; i <= 50000 ; i++ )
printf ( "\n%d", i ) ;
}
```

(b)
```
main( )
{
float a = 13.5 ;
double b = 13.5 ;
printf ( "\n%f %lf", a, b ) ;
}
```

(c)
```
int i = 0 ;
main( )
{
printf ( "\nmain's i = %d", i ) ;
i++ ;
val( ) ;
printf ( "\nmain's i = %d", i ) ;
val( ) ;
}
val( )
{
i = 100 ;
printf ( "\nval's i = %d", i ) ;
i++ ;
}
```


(d)
```
main( )
{
int x, y, s = 2 ;
s *= 3 ;
y = f ( s ) ;
x = g ( s ) ;
printf ( "\n%d %d %d", s, y, x ) ;
}
int t = 8 ;
f ( int a )
{
```

```
a += -5 ;
t -= 4 ;
return ( a + t ) ;
}
g ( int a )
{
a = 1 ;
t += a ;
return ( a + t ) ;
}
(e) main( )
{
static int count = 5 ;
printf ( "\ncount = %d", count-- ) ;
if ( count != 0 )
main( ) ;
}
(f) main( )
{
int i, j ;
for ( i = 1 ; i < 5 ; i++ )
{
j = g ( i ) ;
printf ( "\n%d", j ) ;
}
```



```
}
g ( int x )
{
static int v = 1 ;
int b = 3 ;
v += x ;
return ( v + x + b ) ;
}
(g) float x = 4.5 ;
main( )
{
float y, float f ( float ) ;
x *= 2.0 ;
y = f ( x ) ;
printf ( "\n%f %f", x, y ) ;
}
float f ( float a )
{
a += 1.3 ;
x -= 4.5 ;
return ( a + x ) ;
}
(h) main( )
{
```

```
for ( i = 1 ; i <= UPPER ; i++ )
printf ( "\n%d", i ) ;
}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement,

```
#define UPPER 25
```

This statement is called 'macro definition' or more commonly, just a 'macro'. What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25. Here is another example of macro definition.

```
#define PI 3.1415
main( )
{
float r = 6.25 ;
float area ;
area = PI * r * r ;
printf ( "\nArea of circle = %f", area ) ;
}
```


UPPER and PI in the above programs are often called 'macro templates', whereas, 25 and 3.1415 are called their corresponding 'macro expansions'.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

In C programming it is customary to use capital letters for macro template. This makes it easy for programmers to pick out all the macro templates when reading through the program.

Note that a macro template and its macro expansion are separated by blanks or tabs. A space between # and **define** is optional. Remember that a macro definition is never to be terminated by a semicolon.

And now a million dollar question... why use **#define** in the above programs? What have we gained by substituting PI for 3.1415 in our program? Probably, we have made the program easier to read. Even though 3.1415 is such a common constant that it is easily recognizable, there are many instances where a constant doesn't reveal its purpose so readily. For example, if the phrase "\x1B[2J" causes the screen to clear. But which would you find easier to understand in the middle of your program "\x1B[2J" or "CLEARSCREEN"? Thus, we would use the macro definition

```
#define CLEARSCREEN "\x1B[2J"
```

Then wherever CLEARSCREEN appears in the program it would automatically be replaced by "\x1B[2J" before compilation begins.

One solution in such a situation is to put the old code within a pair of /* */ combination. But we might have already written a comment in the code that we are about to "comment out". This would mean we end up with nested comments. Obviously, this solution won't work since we can't nest comments in C.
Therefore the solution is to use conditional compilation as shown below.

```
main( )
{
#ifdef OKAY
statement 1 ;
statement 2 ; /* detects virus */
statement 3 ;
statement 4 ; /* specific to stone virus */
#endif
statement 5 ;
statement 6 ;
statement 7 ;
}
```

Here, statements 1, 2, 3 and 4 would get compiled only if the macro OKAY has been defined, and we have purposefully omitted the definition of the macro OKAY. At a later date, if we want that these statements should also get compiled all that we are required to do is to delete the **#ifdef** and **#endif** statements.
(b) A more sophisticated use of **#ifdef** has to do with making the programs portable, i.e. to make them work on two totally different computers. Suppose an organization has two

different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with **#ifdef**. For example:

```
main( )
{
#ifdef INTEL
code suitable for a Intel PC
#else
code suitable for a Motorola PC
#endif
code common to both the computers
}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else** control instruction of C.
If you want to run your program on a Motorola PC, just add a statement at the top saying,

Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520

Observe the last three lines of the output. 65526 is original value in **x** plus 2, 65524 is original value in **y** plus 4, and 65520 is original value in **z** plus 1. This so happens because every time a pointer is incremented it points to the immediately next location of its type. That is why, when the integer pointer **x** is incremented, it points to an address two locations after the current location, since an **int** is always 2 bytes long (under Windows/Linux since **int** is 4 bytes long, new value of **x** would be 65528). Similarly, **y** points to an address 4 locations after the current location and **z** points 1 location after the current location. This is a very important result and can be effectively used while passing the entire array to a function.

The way a pointer can be incremented, it can be decremented as well, to point to earlier locations. Thus, the following operations can be performed on a pointer:

(a) Addition of a number to a pointer. For example,

```
int i = 4, *j, *k ;
j = &i ;
j = j + 1 ;
j = j + 9 ;
k = j + 3 ;
```

(b) Subtraction of a number from a pointer. For example,


```
int i = 4, *j, *k ;
j = &i ;
j = j - 2 ;
j = j - 5 ;
k = j - 6 ;
```

(c) Subtraction of one pointer from another.

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements. This is illustrated in the following program.

```
main( )
{
int arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
int *i, *j ;
i = &arr[1] ;
j = &arr[5] ;
printf ( "%d %d", j - i, *j - *i ) ;
}
```

```
for ( j = 0 ; j <= 1 ; j++ )
printf ( "%d ", *( *( s + i ) + j ) ) ;
}
}
```
And here is the output...
```
1234 56
1212 33
1434 80
1312 78
```
Pointer to an Array

If we can have a pointer to an integer, a pointer to a float, a pointer to a char, then can we not have a pointer to an array? We certainly can. The following program shows how to build and use it.


```
/* Usage of pointer to an array */
main( )
{
int s[5][2] = {
{ 1234, 56 },
{ 1212, 33 },
{ 1434, 80 },
{ 1312, 78 }
} ;
int ( *p )[2] ;
int i, j, *pint ;
for ( i = 0 ; i <= 3 ; i++ )
{
p = &s[i] ;
pint = p ;
printf ( "\n" ) ;
for ( j = 0 ; j <= 1 ; j++ )
printf ( "%d ", *( pint + j ) ) ;
}
}
```
And here is the output...
```
1234 56
1212 33
1434 80
1312 78
```
Here **p** is a pointer to an array of two integers. Note that the parentheses in the declaration of **p** are necessary. Absence of them would make **p** an array of 2 integer pointers. Array of pointers is covered in a later section in this chapter. In the outer **for** loop each time we store the address of a new one-dimensional array. Thus first time through this loop **p** would contain the address of the zeroth 1-D array. This address is then assigned to an integer pointer **pint**. Lastly, in the inner **for** loop using the pointer **pint** we

have printed the individual elements of the 1-D array to which **p** is

pointing.
But why should we use a pointer to an array to print elements of a 2-D array. Is there any situation where we can appreciate its usage better? The entity pointer to an array is immensely useful when we need to pass a 2-D array to a function. This is discussed in the next section.

Passing 2-D Array to a Function

There are three ways in which we can pass a 2-D array to a function. These are illustrated in the following program.

```
/* Three ways of accessing a 2-D array */
main( )
{
int a[3][4] = {
1, 2, 3, 4,
5, 6, 7, 8,
9, 0, 1, 6
} ;
clrscr( ) ;
display ( a, 3, 4 ) ;
show ( a, 3, 4 ) ;
print ( a, 3, 4 ) ;
}
display ( int *q, int row, int col )
{
int i, j ;
for ( i = 0 ; i < row ; i++ )
{
for ( j = 0 ; j < col ; j++ )
printf ( "%d ", * ( q + i * col + j ) ) ;
printf ( "\n" ) ;
}
printf ("\n" ) ;
}
show ( int ( *q )[4], int row, int col )
{
int i, j ;
int *p ;
for ( i = 0 ; i < row ; i++ )
{
p = q + i ;
for ( j = 0 ; j < col ; j++ )
printf ( "%d ", * ( p + j ) ) ;
printf ( "\n" ) ;
}
printf ( "\n" ) ;
}
print ( int q[ ][4], int row, int col )
{
int i, j ;
```

[E] What would be the output of the following programs:

(a)
```
main( )
{
int b[ ] = { 10, 20, 30, 40, 50 } ;
int i ;
for ( i = 0 ; i <= 4 ; i++ )
printf ( "\n%d" *( b + i ) ) ;
}
```

(b)
```
main( )
{
int b[ ] = { 0, 20, 0, 40, 5 } ;
int i, *k ;
k = b ;
for ( i = 0 ; i <= 4 ; i++ )
{
printf ( "\n%d" *k ) ;
```



```
k++ ;
}
}
```

(c)
```
main( )
{
int a[ ] = { 2, 4, 6, 8, 10 } ;
int i ;
change ( a, 5 ) ;
for ( i = 0 ; i <= 4 ; i++ )
printf( "\n%d", a[i] ) ;
}
change ( int *b, int n )
{
int i ;
for ( i = 0 ; i < n ; i++ )
*( b + i ) = *( b + i ) + 5 ;
}
```

(d)
```
main( )
{
int a[5], i, b = 16 ;
for ( i = 0 ; i < 5 ; i++ )
a[i] = 2 * i ;
f ( a, b ) ;
for ( i = 0 ; i < 5 ; i++ )
printf ( "\n%d", a[i] ) ;
printf( "\n%d", b ) ;
}
f ( int *x, int y )
{
int i ;
for ( i = 0 ; i < 5 ; i++ )
*( x + i ) += 2 ;
y += 2 ;
```

```
{
char name[ ] = "Klinsman" ;
int i = 0 ;
while ( i <= 7 )
{
printf ( "%c", name[i] ) ;
i++ ;
}
```



```
}
```

And here is the output...

```
Klinsman
```

No big deal. We have initialized a character array, and then printed out the elements of this array within a **while** loop. Can we write the **while** loop without using the final value 7? We can; because we know that each character array always ends with a '\0'. Following program illustrates this.

```
main( )
{
char name[ ] = "Klinsman" ;
int i = 0 ;
while ( name[i] != `\0' )
{
printf ( "%c", name[i] ) ;
i++ ;
}
}
```

And here is the output...

```
Klinsman
```

This program doesn't rely on the length of the string (number of characters in it) to print out its contents and hence is definitely more general than the earlier one. Here is another version of the same program; this one uses a pointer to access the array elements.

```
main( )
{
char name[ ] = "Klinsman" ;
char *ptr ;
```



```
ptr = name ; /* store base address of string */
while ( *ptr != `\0' )
{
printf ( "%c", *ptr ) ;
ptr++ ;
}
}
```

As with the integer array, by mentioning the name of the array we get the base address (address of the zeroth element) of the array. This base address is stored in the variable **ptr** using,

```
ptr = name ;
```

Once the base address is obtained in **ptr**, ***ptr** would yield the

The output of the program would be...
source string = Sayonara
target string = Sayonara
Note that having copied the entire source string into the target string, it is necessary to place a '\0' into the target string, to mark its end.
If you look at the prototype of **strcpy( )** standard library function, it looks like this…
strcpy (char *t, const char *s) ;
We didn't use the keyword **const** in our version of **xstrcpy( )** and still our function worked correctly. So what is the need of the **const** qualifier?
What would happen if we add the following lines beyond the last statement of **xstrcpy( )**?.
s = s - 8 ;
*s = 'K' ;
This would change the source string to "Kayonara". Can we not ensure that the source string doesn't change even accidentally in **xstrcpy( )**? We can, by changing the definition as follows:
void xstrcpy (char *t, const char *s)
{
while (*s != '\0')
{

*t = *s ;
s++ ;
t++ ;
}
*t = '\0' ;
}
By declaring **char *s** as **const** we are declaring that the source string should remain constant (should not change). Thus the **const** qualifier ensures that your program does not inadvertently alter a variable that you intended to be a constant. It also reminds anybody reading the program listing that the variable is not intended to change.
We can use **const** in several situations. The following code fragment would help you to fix your ideas about **const** further.
char *p = "Hello" ; /* pointer is variable, so is string */
p = 'M' ; / works */
p = "Bye" ; /* works */
const char *q = "Hello" ; /* string is fixed pointer is not */
q = 'M' ; / error */
q = "Bye" ; /* works */
char const *s = "Hello" ; /* string is fixed pointer is not */
s = 'M' ; / error */
s = "Bye" ; /* works */
char * const t = "Hello" ; /* pointer is fixed string is not */
t = 'M' ; / works */
t = "Bye" ; /* error */

```
const char * const u = "Hello" ; /* string is fixed so is pointer */
*u = 'M' ; /* error */
u = "Bye" ; /* error */
```


The keyword **const** can be used in context of ordinary variables like **int**, **float**, etc. The following program shows how this can be done.

```
main( )
{
float r, a ;
const float pi = 3.14 ;
printf ( "\nEnter radius of circle " ) ;
scanf ( "%f", &r ) ;
a = pi * r * r ;
printf ( "\nArea of circle = %f", a ) ;
}
```

strcat( )

This function concatenates the source string at the end of the target string. For example, “Bombay” and “Nagpur” on concatenation would result into a string “BombayNagpur”. Here is an example of **strcat( )** at work.

```
main( )
{
char source[ ] = "Folks!" ;
char target[30] = "Hello" ;
strcat ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
```

And here's the output...

```
source string = Folks!
target string = HelloFolks!
```


Note that the target string has been made big enough to hold the final string. I leave it to you to develop your own **xstrcat( )** on lines of **xstrlen( )** and **xstrcpy( ).**

strcmp( )

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, **strcmp( )** returns a value zero. If they’re not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts **strcmp( )** in action.

```
main( )
{
char string1[ ] = "Jerry" ;
char string2[ ] = "Ferry" ;
```

```
for ( i = 0 ; i <= 5 ; i++ )
{
a = strcmp ( &masterlist[i][0], yourname ) ;
if ( a == 0 )
{
printf ( "Welcome, you can enter the palace" ) ;
flag = FOUND ;
break ;
}
}
if ( flag == NOTFOUND )
printf ( "Sorry, you are a trespasser" ) ;
}
```

And here is the output for two sample runs of this program...

```
Enter your name dinesh
Sorry, you are a trespasser
Enter your name raman
Welcome, you can enter the palace
```

Notice how the two-dimensional character array has been initialized. The order of the subscripts in the array declaration is important. The first subscript gives the number of names in the array, while the second subscript gives the length of each item in the array.

Instead of initializing names, had these names been supplied from the keyboard, the program segment would have looked like this.

```
for ( i = 0 ; i <= 5 ; i++ )
scanf ( "%s", &masterlist[i][0] ) ;
```


While comparing the strings through **strcmp( )**, note that the addresses of the strings are being passed to **strcmp( )**. As seen in the last section, if the two strings match **strcmp( )** would return a value 0, otherwise it would return a non-zero value.

The variable **flag** is used to keep a record of whether the control did reach inside the **if** or not. To begin with, we set **flag** to NOTFOUND. Later through the loop if the names match, **flag** is set to FOUND. When the control reaches beyond the **for** loop, if **flag** is still set to NOTFOUND, it means none of the names in the **masterlist[ ][ ]** matched with the one supplied from the keyboard.

The names would be stored in the memory as shown in Figure 9.3. Note that each string ends with a '\0'. The arrangement as you can appreciate is similar to that of a two-dimensional numeric array.

```
65454 a k s h a y \0
65464 p a r a g \0
65474 r a m a n \0
65484 s r i n i v a s \0
65494 g o p a l \0
65504 r a j e s h \0 65513
(last location)
```

Figure 9.3

obtain greater ease in manipulation of the strings. This is shown by the following programs. The first one uses a two-dimensional array of characters to store the names, whereas the second uses an array of pointers to strings. The purpose of both the programs is very simple. We want to exchange the position of the names "raman" and "srinivas".

```
/* Exchange names using 2-D array of characters */
main( )
{
char names[ ][10] = {
```



```
"akshay",
"parag",
"raman",
"srinivas",
"gopal",
"rajesh"
} ;
int i ;
char t ;
printf ( "\nOriginal: %s %s", &names[2][0], &names[3][0] ) ;
for ( i = 0 ; i <= 9 ; i++ )
{
t = names[2][i] ;
names[2][i] = names[3][i] ;
names[3][i] = t ;
}
printf ( "\nNew: %s %s", &names[2][0], &names[3][0] ) ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

Note that in this program to exchange the names we are required to exchange corresponding characters of the two names. In effect, 10 exchanges are needed to interchange two names.

Let us see, if the number of exchanges can be reduced by using an array of pointers to strings. Here is the program...

```
main( )
{
char *names[ ] = {
```



```
"akshay",
"parag",
"raman",
"srinivas",
"gopal",
"rajesh"
} ;
char *temp ;
printf ( "Original: %s %s", names[2], names[3] ) ;
```

(d) main( )
{
char s[ ] = "Churchgate: no church no gate" ;
char t[25] ;
char *ss, *tt ;
ss = s ;
while (*ss != '\0')
*ss++ = *tt++ ;

printf ("\n%s", t) ;
}
(e) main( )
{
char str1[ ] = { 'H', 'e', 'l', 'l', 'o' } ;
char str2[ ] = "Hello" ;
printf ("\n%s", str1) ;
printf ("\n%s", str2) ;
}
(f) main( )
{
printf (5 + "Good Morning ") ;
}
(g) main( )
{
printf ("%c", "abcdefgh"[4]) ;
}
(h) main( )
{
printf ("\n%d%d", sizeof ("3"), sizeof ("3"), sizeof (3)) ;
}

[B] Point out the errors, if any, in the following programs:
(a) main( )
{
char *str1 = "United" ;
char *str2 = "Front" ;
char *str3 ;
str3 = strcat (str1, str2) ;
printf ("\n%s", str3) ;
}
(b) main( )
{

int arr[ ] = { 'A', 'B', 'C', 'D' } ;
int i ;
for (i = 0 ; i <= 3 ; i++)
printf ("\n%d", arr[i]) ;
}
(c) main( )
{
char arr[8] = "Rhombus" ;

September 2004
Mon Tue Wed Thu Fri Sat Sun
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
Note that according to the Gregorian calendar 01/01/1900 was Monday. With this as the base the calendar should be generated.
(e) Modify the above program suitably so that once the calendar for a particular month and year has been displayed on the

screen, then using arrow keys the user must be able to change the calendar in the following manner:
Up arrow key : Next year, same month
Down arrow key : Previous year, same month
Right arrow key : Same year, next month
Left arrow key : Same year, previous month
If the escape key is hit then the procedure should stop.
Hint: Use the **getkey( )** function discussed in Chapter 8, problem number [L](c).
(f)
(g)
(h)
(i)
A factory has 3 division and stocks 4 categories of products. An inventory table is updated for each division and for each product as they are received. There are three independent suppliers of products to the factory:
(a) Design a data format to represent each transaction.
(b) Write a program to take a transaction and update the inventory.
(c) If the cost per item is also given write a program to calculate the total inventory values.
A dequeue is an ordered set of elements in which elements may be inserted or retrieved from either end. Using an array simulate a dequeue of characters and the operations retrieve left, retrieve right, insert left, insert right. Exceptional conditions such as dequeue full or empty should be indicated. Two pointers (namely, left and right) are needed in this simulation.
Write a program to delete all vowels from a sentence. Assume that the sentence is not more than 80 characters long.
Write a program that will read a line and delete from it all occurrences of the word ‘the’.

(j)
(k)
Write a program that takes a set of names of individuals and

Actually the structure elements are stored in memory as shown in the Figure 10.1.

65518 65519 65523
'B' 130.00 550
b1.name b1.price b1.pages
Figure 10.1

Array of Structures

Our sample program showing usage of structure is rather simple minded. All it does is, it receives values into various structure elements and output these values. But that's all we intended to do anyway... show how structure types are created, how structure variables are declared and how individual elements of a structure variable are referenced.

In our sample program, to store data of 100 books we would be required to use 100 different structure variables from **b1** to **b100**, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures.


```
/* Usage of an array of structures */
main( )
{
struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b[100] ;
int i ;
for ( i = 0 ; i <= 99 ; i++ )
{
printf ( "\nEnter name, price and pages " ) ;
scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
}
for ( i = 0 ; i <= 99 ; i++ )
printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;
}
linkfloat( )
{
float a = 0, *b ;
b = &a ; /* cause emulator to be linked */
a = *b ; /* suppress the warning - variable not used */
}
```

Now a few comments about the program:

(a) Notice how the array of structures is declared...

```
struct book b[100] ;
```


This provides space in memory for 100 structures of the type

```
int pin ;
} ;
struct emp
{
char name[25] ;
struct address a ;
} ;
struct emp e = { "jeru", "531046", "nagpur", 10 };
printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;
printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;
}
```

And here is the output...

```
name = jeru phone = 531046
city = nagpur pin = 10
```

Notice the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression,

e.a.pin or e.a.city

Of course, the nesting process need not stop at this level. We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves. Such construction however gives rise to variable names that can be surprisingly self descriptive, for example:

maruti.engine.bolt.large.qty

This clearly signifies that we are referring to the quantity of large sized bolts that fit on an engine of a maruti car.

(c) Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go. Let us examine both the approaches one by one using suitable programs.

```
/* Passing individual structure elements */
main( )
{
struct book
{
char name[25] ;
char author[25] ;
int callno ;
} ;
struct book b1 = { "Complete Guide To C", "YPK", 101 } ;
display ( b1.name, b1.author, b1.callno ) ;
}
display ( char *s, char *t, int n )
{
printf ( "\n%s %s %d", s, t, n ) ;
}
```

And here is the output...

```
int minutes ;
int seconds ;
} t ;
struct time *tt ;
tt = &t ;
```

Looking at the above declarations, which of the following refers to **seconds** correctly:

1. tt.seconds
2. (*tt).seconds
3. time.t
4. tt -> seconds

[D] Attempt the following:

(a) Create a structure to specify data on students given below: Roll number, Name, Department, Course, Year of joining Assume that there are not more than 450 students in the collage.

(a) Write a function to print names of all students who joined in a particular year.

(b) Write a function to print the data of a student whose roll number is given.

(b) Create a structure to specify data of customers in a bank. The data to be stored is: Account number, Name, Balance in account. Assume maximum of 200 customers in the bank.

(a) Write a function to print the Account number and name of each customer with balance below Rs. 100.

(b) If a customer request for withdrawal or deposit, it is given in the form:

Acct. no, amount, code (1 for deposit, 0 for withdrawal)

Write a program to give a message, "The balance is insufficient for the specified withdrawal".

(c) An automobile company has serial number for engine parts starting from AA0 to FF9. The other characteristics of parts to be specified in a structure are: Year of manufacture, material and quantity manufactured.

(a) Specify a structure to store information corresponding to a part.

(b) Write a program to retrieve information on parts with serial numbers between BB1 and CC6.

(d) A record contains name of cricketer, his age, number of test matches that he has played and the average runs that he has scored in each test match. Create an array of structure to hold records of 20 such cricketer and then write a program to read these records and arrange them in ascending order by average runs. Use the **qusort( )** standard library function.

(e) There is a structure called **employee** that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years

```
main( )
{
printf ( "You\tmust\tbe\tcrazy\nto\thate\tthis\tbook" ) ;
}
```


And here's the output...

```
1 2 3 4
01234567890123456789012345678901234567890
You must be crazy
to hate this book
```

The **\n** character causes a new line to begin following 'crazy'. The tab and newline are probably the most commonly used escape sequences, but there are others as well. Figure 11.4 shows a complete list of these escape sequences.

Esc. Seq.	Purpose	Esc. Seq.	Purpose
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Form feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

Figure 11.4

The first few of these escape sequences are more or less selfexplanatory. **\b** moves the cursor one position to the left of its current position. **\r** takes the cursor to the beginning of the line in which it is currently placed. **\a** alerts the user by sounding the speaker inside the computer. Form feed advances the computer stationery attached to the printer to the top of the next page. Characters that are ordinarily used as delimiters... the single quote, double quote, and the backslash can be printed by preceding them with the backslash. Thus, the statement,

```
printf ( "He said, \"Let's do it!\" ) ;
```


will print...

```
He said, "Let's do it!"
```

So far we have been describing **printf( )'s** specification as if we are forced to use only **%d** for an integer, only **%c** for a char, only **%s** for a string and so on. This is not true at all. In fact, **printf( )** uses the specification that we mention and attempts to perform the specified conversion, and does its best to produce a proper result. Sometimes the result is nonsensical, as in case when we ask it to print a string using **%d**. Sometimes the result is useful, as in the case we ask **printf( )** to print ASCII value of a character using **%d**. Sometimes the result is disastrous and the entire program blows up.

The following program shows a few of these conversions, some sensible, some weird.

```
main( )
{
char ch = 'z' ;
int i = 125 ;
```

```
float a = 12.55 ;
char s[ ] = "hello there !" ;
printf ( "\n%c %d %f", ch, ch, ch ) ;
printf ( "\n%s %d %f", s, s, s ) ;
printf ( "\n%c %d %f",i ,i, i ) ;
printf ( "\n%f %d\n", a, a ) ;
}
```

And here's the output ...

```
z 122 -9362831782501783000000000000000000000000000000.000000
hello there ! 3280 -
9362831782501783000000000000000000000000000000.000000
} 125 -9362831782501783000000000000000000000000000000.000000
```



```
12.550000 0
```

I would leave it to you to analyze the results by yourselves. Some of the conversions you would find are quite sensible.

Let us now turn our attention to **scanf( )**. **scanf( )** allows us to enter data from keyboard that will be formatted in a certain way. The general form of **scanf( )** statement is as follows:

```
scanf ( "format string", list of addresses of variables ) ;
```

For example:

```
scanf ( "%d %f %c", &c, &a, &ch ) ;
```

Note that we are sending addresses of variables (addresses are obtained by using '**&**' the 'address of' operator) to **scanf( )** function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s). Do not include these escape sequences in the format string.

All the format specifications that we learnt in **printf( )** function are applicable to **scanf( )** function as well.

sprintf( ) and *sscanf( )* Functions

The **sprintf( )** function works similar to the **printf( )** function except for one small difference. Instead of sending the output to the screen as **printf( )** does, this function writes the output to an array of characters. The following program illustrates this.

```
main( )
{
```



```
int i = 10 ;
char ch = 'A' ;
float a = 3.14 ;
char str[20] ;
printf ( "\n%d %c %f", i, ch, a ) ;
sprintf ( str, "%d %c %f", i, ch, a ) ;
printf ( "\n%s", str ) ;
}
```

In this program the **printf( )** prints out the values of **i**, **ch** and **a** on the screen, whereas **sprintf( )** stores these values in the character

```
for ( i = 0 ; i < 5 ; i++ )
scanf ( "%s", mess[i] ) ;
}
(e) main( )
{
int dd, mm, yy ;
printf ( "\nEnter day, month and year\n" ) ;
scanf ( "%d%*c%d%*c%d", &dd, &mm, &yy ) ;
printf ( "The date is: %d - %d - %d", dd, mm, yy ) ;
}
(f) main( )
{
char text ;
sprintf ( text, "%4d\t%2.2f\n%s", 12, 3.452, "Merry Go Round" ) ;
printf ( "\n%s", text ) ;
}
(g) main( )
{
char buffer[50] ;
```



```
int no = 97;
double val = 2.34174 ;
char name[10] = "Shweta" ;
sprintf ( buffer, "%d %lf %s", no, val, name ) ;
printf ( "\n%s", buffer ) ;
sscanf ( buffer, "%4d %2.2lf %s", &no, &val, name ) ;
printf ( "\n%s", buffer ) ;
printf ( "\n%d %lf %s", no, val, name ) ;
}
```

[C] Answer the following:

(a)

(b)

(c)

To receive the string "We have got the guts, you get the glory!!" in an array **char str[100]** which of the following functions would you use?

1. scanf ("%s", str) ;
2. gets (str) ;
3. getche (str) ;
4. fgetchar (str) ;

Which function would you use if a single key were to be received through the keyboard?

1. scanf( )
2. gets( )
3. getche( )
4. getchar( )

If an integer is to be entered through the keyboard, which function would you use?

1. scanf( )
2. gets( )

```
if ( ch == '\n' )
nol++ ;
if ( ch == '\t' )
not++ ;
}
fclose ( fp ) ;
printf ( "\nNumber of characters = %d", noc ) ;
printf ( "\nNumber of blanks = %d", nob ) ;
printf ( "\nNumber of tabs = %d", not ) ;
printf ( "\nNumber of lines = %d", nol ) ;
}
```


Here is a sample run...

```
Number of characters = 125
Number of blanks = 25
Number of tabs = 13
Number of lines = 22
```

The above statistics are true for a file "PR1.C", which I had on my disk. You may give any other filename and obtain different results. I believe the program is self-explanatory.

In this program too we have opened the file for reading and then read it character by character. Let us now try a program that needs to open a file for writing.

A File-copy Program

We have already used the function **fgetc( )** which reads characters from a file. Its counterpart is a function called **fputc( )** which writes characters to a file. As a practical use of these character I/O functions we can copy the contents of one file into another, as demonstrated in the following program. This program takes the contents of a file and copies them into another file, character by character.

```
#include "stdio.h"
main( )
{
FILE *fs, *ft ;
char ch ;
fs = fopen ( "pr1.c", "r" ) ;
if ( fs == NULL )
{
puts ( "Cannot open source file" ) ;
exit( ) ;
```



```
}
ft = fopen ( "pr2.c", "w" ) ;
if ( ft == NULL )
{
puts ( "Cannot open target file" ) ;
fclose ( fs ) ;
exit( ) ;
```

```
}
while ( 1 )
{
ch = fgetc ( fs ) ;
if ( ch == EOF )
break ;
else
fputc ( ch, ft ) ;
}
fclose ( fs ) ;
fclose ( ft ) ;
}
```

I hope most of the stuff in the program can be easily understood, since it has already been dealt with in the earlier section. What is new is only the function **fputc( )**. Let us see how it works.

Writing to a File

The **fputc( )** function is similar to the **putch( )** function, in the sense that both output characters. However, **putch( )** function always writes to the VDU, whereas, **fputc( )** writes to the file. Which file? The file signified by **ft**. The writing process continues till all characters from the source file have been written to the target file, following which the **while** loop terminates.

Note that our sample file-copy program is capable of copying only text files. To copy files with extension .EXE or .COM, we need to open the files in binary mode, a topic that would be dealt with in sufficient detail in a later section.

File Opening Modes

In our first program on disk I/O we have opened the file in read ("r") mode. However, "r" is but one of the several modes in which we can open a file. Following is a list of all possible modes in which a file can be opened. The tasks performed by **fopen( )** when a file is opened in each of these modes are also mentioned.

"r" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened **fopen( )** returns NULL.

Operations possible – reading from the file.

"w" Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible – writing to the file.

"a" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible - adding new contents at the end of file.

"r+" Searches file. If is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first

reached. Hence we use the **!** operator to negate this 0 to the truth value. When the end of file is reached **feof( )** returns a non-zero

value, **!** makes it 0 and since now the condition evaluates to false the **while** loop gets terminated.

Note that in each one of them the following three methods for opening a file are same, since in each one of them, essentially a base address of the string (pointer to a string) is being passed to **fopen( )**.

```
fs = fopen ( "PR1.C" , "r" ) ;
fs = fopen ( filename, "r" ) ;
fs = fopen ( argv[1] , "r" ) ;
```

Detecting Errors in Reading/Writing

Not at all times when we perform a read or write operation on a file are we successful in doing so. Naturally there must be a provision to test whether our attempt to read/write was successful or not.

The standard library function **ferror( )** reports any error that might have occurred during a read/write operation on a file. It returns a zero if the read/write is successful and a non-zero value in case of a failure. The following program illustrates the usage of **ferror( )**.

```
#include "stdio.h"
main( )
{
FILE *fp ;
char ch ;
fp = fopen ( "TRIAL", "w" ) ;
while ( !feof ( fp ) )
{
ch = fgetc ( fp ) ;
if ( ferror( ) )
{
```



```
printf ( "Error in reading file" ) ;
break ;
}
else
printf ( "%c", ch ) ;
}
fclose ( fp ) ;
}
```

In this program the **fgetc( )** function would obviously fail first time around since the file has been opened for writing, whereas **fgetc( )** is attempting to read from the file. The moment the error occurs **ferror( )** returns a non-zero value and the **if** block gets executed. Instead of printing the error message using **printf( )** we can use the standard library function **perror( )** which prints the error message specified by the compiler. Thus in the above program the **perror( )** function can be used as shown below.

}
And here is the output...
Decimal 0 is same as binary 0000000000000000
Decimal 1 is same as binary 0000000000000001
Decimal 2 is same as binary 0000000000000010
Decimal 3 is same as binary 0000000000000011
Decimal 4 is same as binary 0000000000000100
Decimal 5 is same as binary 0000000000000101
Let us now explore the various bitwise operators one by one.

One's Complement Operator

On taking one's complement of a number, all 1's present in the number are changed to 0's and all 0's are changed to 1's. For example one's complement of 1010 is 0101. Similarly, one's complement of 1111 is 0000. Note that here when we talk of a number we are talking of binary equivalent of the number. Thus, one's complement of 65 means one's complement of 0000 0000 0100 0001, which is binary equivalent of 65. One's complement of 65 therefore would be, 1111 1111 1011 1110. One's complement operator is represented by the symbol ~. Following program shows one's complement operator in action.

```
main( )
{
int j, k ;
for ( j = 0 ; j <= 3 ; j++ )
{
printf ( "\nDecimal %d is same as binary ", j ) ;
showbits ( j ) ;
k = ~j ;
printf ( "\nOne's complement of %d is ", j ) ;
showbits ( k ) ;
}
}
```


And here is the output of the above program...

```
Decimal 0 is same as binary 0000000000000000
One's complement of 0 is 1111111111111111
Decimal 1 is same as binary 0000000000000001
One's complement of 1 is 1111111111111110
Decimal 2 is same as binary 0000000000000010
One's complement of 2 is 1111111111111101
Decimal 3 is same as binary 0000000000000011
One's complement of 3 is 1111111111111100
```

In real-world situations where could the one's complement operator be useful? Since it changes the original number beyond recognition, one potential place where it can be effectively used is in development of a file encryption utility as shown below:

```
/* File encryption utility */
#include "stdio.h"
main( )
{
```

5225 left shift 0 gives 0001010001101001
5225 left shift 1 gives 0010100011010010
5225 left shift 2 gives 0101000110100100
5225 left shift 3 gives 1010001101001000
5225 left shift 4 gives 0100011010010000

Having acquainted ourselves with the left shift and right shift operators, let us now find out the practical utility of these operators.

In DOS/Windows the date on which a file is created (or modified) is stored as a 2-byte entry in the 32 byte directory entry of that file. Similarly, a 2-byte entry is made of the time of creation or modification of the file. Remember that DOS/Windows doesn't store the date (day, month, and year) of file creation as a 8 byte string, but as a codified 2 byte entry, thereby saving 6 bytes for each file entry in the directory. The bitwise distribution of year, month and date in the 2-byte entry is shown in Figure 14.3.

month
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Y Y Y Y Y Y Y M M M M D D D D D
year day

Figure 14.3

DOS/Windows converts the actual date into a 2-byte value using the following formula:

date = 512 * (year - 1980) + 32 * month + day

Suppose 09/03/1990 is the date, then on conversion the date will be,

date = 512 * (1990 - 1980) + 32 * 3 + 9 = 5225

The binary equivalent of 5225 is 0001 0100 0110 1001. This binary value is placed in the date field in the directory entry of the file as shown below.

Figure 14.4

0 0 0 1 0 1 0 0 0 1 1 0 1 0 0 1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
year month day

Just to verify this bit distribution, let us take the bits representing the month,

month = 0011
= 1 * 2 + 1 * 1
= 3

Similarly, the year and the day can also be verified.

When we issue the command DIR or use Windows Explorer to list the files, the file's date is again presented on the screen in the usual date format of mm/dd/yy. How does this integer to date conversion take place? Obviously, using left shift and right shift operators.

When we take a look at Figure 14.4 depicting the bit pattern of the 2- byte date field, we see that the year, month and day exist as a bunch of bits in contiguous locations. Separating each of them is a matter of applying the bitwise operators.

For example, to get year as a separate entity from the two bytes

. . . . . 1 . . System
. . . . 1 . . . Volume label entry
. . . 1 . . . . Sub-directory entry
. . 1 . . . . . Archive bit
. 1 . . . . . Unused
1 . . . . . Unused
Figure 14.10
Now, suppose we want to check whether a file is a hidden file or not. A hidden file is one, which is never shown in the directory, even though it exists on the disk. From the above bit classification of attribute byte, we only need to check whether bit number 1 is ON or OFF.
So, our first operand in this case becomes the attribute byte of the file in question, whereas the second operand is the $1 * 2_1 = 2$, as discussed earlier. Similarly, it can be checked whether the file is a system file or not, whether the file is read-only file or not, and so on.
The second, and equally important use of the AND operator is in changing the status of the bit, or more precisely to switch OFF a particular bit.

If the first operand happens to be 00000111, then to switch OFF bit number 1, our AND mask bit pattern should be 11111101. On applying this mask, we get,
00000111 Original bit pattern
11111101 AND mask

00000101 Resulting bit pattern
Here in the AND mask we keep the value of all other bits as 1 except the one which is to be switched OFF (which is purposefully kept as 0). Therefore, irrespective of whether the first bit is ON or OFF previously, it is switched OFF. At the same time the value 1 provided in all the other bits of the AND mask (second operand) keeps the bit values of the other bits in the first operand unaltered.
Let's summarize the uses of bitwise AND operator:
(a) It is used to check whether a particular bit in a number is ON or OFF.
(b) It is used to turn OFF a particular bit in a number.

Bitwise OR Operator

Another important bitwise operator is the OR operator which is represented as |. The rules that govern the value of the resulting bit obtained after ORing of two bits is shown in the truth table below.
| 0 1
0 0 1
1 1 1
Figure 14.11

Using the Truth table confirm the result obtained on ORing the

Windows. So in the rest of this book whenever I refer to Windows I mean Windows NT family, unless explicitly specified.
Before we start writing C programs under Windows let us first see some of the changes that have happened under Windows environment.

Integers

Under 16-bit environment the size of integer is of 2 bytes. As against this, under 32-bit environment an integer is of 4 bytes. Hence its range is -2147483648 to +2147483647. Thus there is no difference between an **int** and a **long int**. But what if we wish to store the age of a person in an integer? It would be improper to sacrifice a 4-byte integer when we know that the number to be stored in it is hardly going to exceed hundred. In such as case it would be more sensible to use a **short int** since it is only 2 bytes long.

The Use of *typedef*

Take a look at the following declarations:
COLORREF color ;
HANDLE h ;
WPARAM w ;
LPARAM l ;
BOOL b ;

Are COLORREF, HANDLE, etc. new datatypes that have been added in C under Windows compiler? Not at all. They are merely typedef's of the normal integer datatype.
A typical C under Windows program would contain several such **typedef**s. There are two reasons why Windows-based C programs heavily make use of **typedef**s. These are:
(a)
(b)
A typical Windows program is required to perform several complex tasks. For example a program may print documents, send mails, perform file I/O, manage multiple threads of execution, draw in a window, play sound files, perform operations over the network apart from normal data processing tasks. Naturally a program that carries out so many tasks would be very big in size. In such a program if we start using the normal integer data type to represent variables that hold different entities we would soon lose track of what that integer value actually represents. This can be overcome by suitably typedefining the integer as shown above.
At several places in Windows programming we are required to gather and work with dissimilar but inter-related data. This can be done using a structure. But when we define any structure variable we are required to precede it with the keyword **struct**. This can be avoided by using **typedef** as shown below:
struct rect

DOS programs are always required to bother about the details of the hardware on which they are running. This is because for every new piece of hardware introduced there are new interrupt numbers and new register details. Hence DOS programmers are under the constant fear that if the hardware on which the programs are running changes then the program may crash.

Moreover the DOS programmer has to write lot of code to detect the hardware on which his program is running and suitably make use of the relevant interrupts and registers. Not only does this make the program lengthy, the programmer has to understand a lot of technical details of the hardware. As a result the programmer has to spend more time in understanding the hardware than in the actual application programming.

Windows Programming Model

From the perspective of the user the shift from MS-DOS to Windows OS involves switching over to a Graphical User Interface from the typical Text Interface that MS-DOS offers. Another change that the user may feel and appreciate is the ability of Windows OS to execute several programs simultaneously, switching effortlessly from one to another by pointing at windows and clicking them with the mouse. Mastering this new GUI environment and getting comfortable with the multitasking feature is at the most a matter of a week or so. However, from the programmer's point of view programming for Windows is a whole new ball game!

Windows programming model is designed with a view to:

(a)
(b)
(c)
(d)

Eliminate the messy calling mechanism of DOS
Permit true reuse of commonly used functions
Provide consistent look and feel for all applications
Eliminate hardware dependency

Let us discuss how Windows programming model achieves this.

Better Calling Mechanism

Instead of calling functions using Interrupt numbers and registers Windows provides functions within itself which can be called using names. These functions are called API (Application Programming Interface) functions. There are literally hundreds of

API functions available. They help an application to perform various tasks such as creating a window, drawing a line, performing file input/output, etc.

True Reuse

A C under Windows program calls several API functions during course of its execution. Imagine how much disk space would have been wasted had each of these functions become part of the EXE

interact with the user-interface elements of the program cannot be predicted the order of occurrence of events, and hence the order of messages, also becomes unpredictable. As a result, the order of

calling the functions in the program (that react to different messages) is dictated by the order of occurrence of events. Hence this programming model is called 'Event Driven Programming Model'.

That's really all that is there to event-driven programming. Your job is to anticipate what users are likely to do with your application's user interface objects and have a function waiting, ready to execute at the appropriate time. Just when that time is, no one except the user can really say.

Windows Programming, a Closer Look

There can be hundreds of ways in which the user may interact with an application. In addition to this some events may occur without any user interaction. For example, events occur when we create a window, when the window's contents are to be drawn, etc. Not only this, occurrence of one event may trigger a few more events. Thus literally hundreds of messages may be sent to an application thereby creating a chaos. Naturally, a question comes—in which order would these messages get processed by the application. Order is brought to this chaos by putting all the messages that reach the application into a 'Queue'. The messages in the queue are processed in First In First Out (FIFO) order.

In fact the OS maintains several such queues. There is one queue, which is common for all applications. This queue is known as 'System Message Queue'. In addition there is one queue per application. Such queues are called 'Application Message Queues'. Let us understand the need for maintaining so many queues.

When we click a mouse and an event occurs the device driver posts a message into the System Message Queue. The OS retrieves this message finds out with regard to which application the message has been sent. Next it posts a message into the

Application Message Queue of the application in which the mouse was clicked. Refer Figure 16.5.

Application2 Application2
Msg. Queue
Application1
Msg. Queue
Application1
Event Event
Device Driver Device Driver
Other OS
Mess
Other
Messa
System Msg.

17 Windows Programming

event driven programming requires a change in mind set. I hope Chapter 16 has been able to bring about this change. However this change would be bolstered by writing event driven programs. This is what this chapter intends to do. I am hopeful that by the time you reach the end of this chapter you would be so comfortable with it as if you have been using it all your life.

The Role of a Message Box

Often we are required to display certain results on the screen during the course of execution of a program. We do this to ascertain whether we are getting the results as per our expectations. In a sequential DOS based program we can easily achieve this using **printf( )** statements. Under Windows screen is a shared resource. So you can imagine what chaos would it create if all running applications are permitted to write to the screen. You would not be able to make out which output is of what application. Hence no Windows program is permitted to write anything directly to the screen. That's where a message box enters the scene. Using it we can display intermediate results during the course of execution of a program. It can be dismissed either by clicking the 'close button' in its title bar or by clicking the OK button present in it. There are numerous variations that you can try with the **MessageBox( )**. Some of these are given below

```
MessageBox ( 0, "Are you sure", "Caption", MB_YESNO ) ;
MessageBox ( 0, "Print to the Printer", "Caption", MB_YESNO CANCEL) ;
MessageBox ( 0, "icon is all about style", "Caption", MB_OK |
```

provide ‘device independence’. Device independence means that the same program should be able to work using different screens, keyboards and printers without modification to the program. Windows takes care of the hardware, allowing the programmer to concentrate on the program itself. If you have ever had to update the code of an MS-DOS program for the latest printer, plotter, video display, or keyboard, you will recognize device independence as a huge advantage for the developer.

Windows programs do not send data directly to the screen or printer. A Windows program knows where (screen/printer) its output is being sent. However, it does not know how it would be sent there, neither does it need to bother to know this. This is because Windows uses a standard and consistent way to send the output to screen/printer. This standard way uses an entity called Device Context, or simply a DC. Different DC’s are associated with different devices. For example, a screen DC is associated with a screen, a printer DC is associated with a printer, etc. Any drawing that we do using the screen DC is directed to the screen. Similarly, any drawing done using the printer DC is directed to the printer. Thus, the only thing that changes from drawing to screen and drawing to printer is the DC that is used.

A windows program obtains a handle (ID value) for the screen or printer’s DC. The output data is sent to the screen/printer using its DC, and then Windows and the Device Driver for the device takes care of sending it to the real hardware. The advantage of using the DC is that the graphics and text commands that we send us ng the DC are always the same, regardless of where the physical output is showing up.

The part of Windows that converts the Windows graphics function calls to the actual commands sent to the hardware is the GDI, or Graphics Device Interface. The GDI is a program file called GDI32.DLL and is stored in the Windows System directory. The

Windows environment loads GDI32.DLL into memory when it is needed for graphical output. Windows also loads a ‘device driver’ program if the hardware conversions are not part of GDI32.DLL. Common examples are VGA.SYS for VGA video screen and HPPLC.SYS for the HP LaserJet printer. Drivers are just programs that assist the GDI in converting Windows graphics commands to hardware commands.

Thus GDI provides all the basic drawing functionality for Windows; the device context represents the device providing a layer of abstraction that insulates your applications from the trouble of drawing directly to the hardware. The GDI provides this insulation by calling the appropriate device driver in response to windows graphics function calls.

Hello Windows

We would begin our tryst with graphics programming under windows by displaying a message “Hello Windows” in different fonts. Note that though we are displaying text under Windows

even text gets drawn graphically in the window. First take a look at the program given below before we set out to understand it.

```
# include <windows.h>
# include "helper.h"
void OnPaint ( HWND ) ;
void OnDestroy ( HWND ) ;
int __stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpszCmdline, int nCmdShow )
{
MSG m ;
/* Perform application initialization */
InitInstance ( hInstance, nCmdShow, "Text" ) ;
```



```
/* Main message loop */
while ( GetMessage ( &m, NULL, 0, 0 ) )
DispatchMessage(&m);
return 0 ;
}
LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
WPARAM wParam, LPARAM lParam )
{
switch ( message )
{
case WM_DESTROY :
OnDestroy ( hWnd ) ;
break ;
case WM_PAINT :
OnPaint ( hWnd ) ;
break ;
default :
return DefWindowProc ( hWnd, message, wParam, lParam ) ;
}
return 0 ;
}
void OnDestroy ( HWND hWnd )
{
PostQuitMessage ( 0 ) ;
}
void OnPaint ( HWND hWnd )
{
HDC hdc ;
PAINTSTRUCT ps ;
HFONT hfont ;
LOGFONT f = { 0 } ;
HGDIOBJ holdfont ;
char *fonts[ ] = { "Arial", "Times New Roman", "Comic Sans MS" } ;
int i ;
```



```
hdc = BeginPaint ( hWnd, &ps ) ;
for ( i = 0 ; i < 3 ; i++ )
```

arc's starting point and ending point respectively.
In **Polygon (lpPoints, nCount)**, **lpPoints** points to an array of points that specifies the vertices of the polygon. Each point in the array is a **POINT** structure. **nCount** specifies the number of vertices stored in the array. The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first.

Once we are through with drawing the shapes the old brush is selected back in the DC and then the brush created by us is deleted using **DeleteObject( )** function.

Types of Pens

In the previous program we have used the default solid black pen of thickness 1 pixel. We can create pens of different style, color and thickness to do our drawing. The following **OnPaint( )** handler shows how this can be achieved.

```
void OnPaint ( HWND hWnd )
{
HDC hdc ;
PAINTSTRUCT ps ;
HPEN hpen ;
HGDIOBJ holdpen ;
hdc = BeginPaint ( hWnd, &ps ) ;
hpen = CreatePen ( PS_DASH, 1, RGB ( 255, 0, 0 ) ) ;
holdpen = SelectObject ( hdc, hpen ) ;
MoveToEx ( hdc, 10, 10, NULL ) ;
LineTo ( hdc, 500, 10 ) ;
SelectObject ( hdc, holdpen ) ;
DeleteObject ( hpen ) ;
hpen = CreatePen ( PS_DOT, 1, RGB ( 255, 0, 0 ) ) ;
holdpen = SelectObject ( hdc, hpen ) ;
MoveToEx ( hdc, 10, 60, NULL ) ;
LineTo ( hdc, 500, 60 ) ;
SelectObject ( hdc, holdpen ) ;
DeleteObject ( hpen ) ;
```



```
hpen = CreatePen ( PS_DASHDOT, 1, RGB ( 255, 0, 0 ) ) ;
holdpen = SelectObject ( hdc, hpen ) ;
MoveToEx ( hdc, 10, 110, NULL ) ;
LineTo ( hdc, 500, 110 ) ;
SelectObject ( hdc, holdpen ) ;
DeleteObject ( hpen ) ;
hpen = CreatePen ( PS_DASHDOTDOT, 1, RGB ( 255, 0, 0 ) ) ;
holdpen = SelectObject ( hdc, hpen ) ;
MoveToEx ( hdc, 10, 160, NULL ) ;
LineTo ( hdc, 500, 160 ) ;
SelectObject ( hdc, holdpen ) ;
DeleteObject ( hpen ) ;
hpen = CreatePen ( PS_SOLID, 10, RGB ( 255, 0, 0 ) ) ;
```

we have created a memory device context and made its properties compatible with that of the screen DC. To do this we have called the API function **CreateCompatibleDC( )**. Note that we have passed the handle to the screen DC to this function. The function in turn returns the handle to the memory DC. After this we have selected the loaded bitmap into the memory DC. Lastly, we have performed a bit block transfer (a bit by bit copy) from memory DC to screen DC using the function **BitBlt( )**. As a result of this the vulture now appears in the window.

We have made the call to **BitBlt( )** as shown below:

```
BitBlt ( hdc, 10, 20, 190, 220, hmemdc, 0, 0, SRCCOPY ) ;
```


Let us now understand its parameters. These are as under:

hdc – Handle to target DC where the bitmap is to be blitted

10, 20 – Position where the bitmap is to be blitted

190, 220 – Width and height of bitmap being blitted

0, 0 – Top left corner of the source image. If we give 10, 20 then the image from 10, 20 to bottom right corner of the bitmap would get blitted.

SRCCOPY – Specifies one of the raster-operation codes. These codes define how the color data for the source rectangle is to be combined with the color data for the destination rectangle to achieve the final color. SRCCOPY means that the pixel color of source should be copied onto the destination pixel of the target.

Animation at Work

Speed is the essence of life. So having the ability to display a bitmap in a window is fine, but if we can add movement and sound to it then nothing like it. So let us now see how to achieve this animation and sound effect.

If we are to animate an object in the window we need to carry out the following steps:

(a) Create an image that is to be animated as a resource.

(b) Prepare the image for later display.

(c) Repeatedly display this prepared image at suitable places in the window taking care that when the next image is displayed the previous image is erased.

(d) Check for collisions while displaying the prepared image.

Let us now write a program that on execution makes a red colored ball move in the window. As the ball strikes the walls of the

window a noise occurs. Note that the width and height of the redcolored ball is 22 pixels. Given below is the **WndProc( )** function and the various message handlers that help achieve animation and sound effect.

```
HBITMAP hbmp ;
```

Driven Programming model. Once you have understood it thoroughly rest is just a matter of understanding and calling the suitable API functions to get your job done. Windows API is truly an endless world. It covers areas like Networking, Internet programming, Telephony, Drawing and Printing, Device I/O, Imaging, Messaging, Multimedia, Windowing, Database programming, Shell programming, to name a few. The programs that we have written have merely scratched the surface. No matter how many programs that we write under Windows, several still remain to be written. The intention of this chapter was to unveil before you, to give you the first glimpse of what is possible under Windows. The intention all along was not to catch fish for you but to show you how to catch fish so that you can do fishing all your life. Having made a sound beginning, rest is for you to explore. Good luck and happy fishing!

Summary

(a) In DOS, programmers had to write separate graphics code for every new video adapter. In Windows, the code once written works on any video adapter.
(b) A Windows program cannot draw directly on an output device like screen or printer. Instead, it draws to the logical display surface using device context.
(c) When the window is displayed for the first time, or when it is moved or resized **OnPaint( )** handler gets called.
(d) It is necessary to obtain the device context before drawing text or graphics in the client area.
(j) A device context is a structure containing information required to draw on a display surface. The information includes color of pen and brush, screen resolution, color palettes, etc.
(e) To draw using a new pen or brush it is necessary to select them into the device context.

(f) If we don't select any brush or pen into the device context then the drawing drawn in the client area would be drawn with the default pen (black pen) and default brush (white brush).
(g) RGB is a macro representing the Red, Green and Blue elements of a color. RGB (0, 0, 0) gives black color, whereas, RGB (255, 255, 255) gives white color.
(h) Animation involves repeatedly drawing the same image at successive positions.

Exercise

[A] State True or False:
(a) Device independence means the same program is able to work using different screens, keyboards and printers without modifications to the program.
(b) The WM_PAINT message is generated whenever the client area of the window needs to be redrawn.

here are two types of Windows programmers those who are happy in knowing the things the way they are under Windows and those who wish to know why the things are the way they are. This chapter is for the second breed of programmers. They are the real power users of Windows. Because it is they who first understand the default working of different mechanisms that Windows uses and then are able to make those mechanisms work to their advantage. The focus here would be restricted to mechanisms that are involved in interaction with the hardware under the Windows world. Read on and I am sure you would be on your path to become a powerful Windows programmer.

T

Hardware Interaction

Primarily interaction with hardware suggests interaction with peripheral devices. However, its reach is not limited to interaction with peripherals. The interaction may also involve communicating with chips present on the motherboard. Thus more correctly, interaction with hardware would mean interaction with any chip other than the microprocessor. During this interaction one or more of the following activities may be performed:

(a)

(b)

(c)

Reacting to events that occur because of user's interaction with the hardware. For example, if the user presses a key or clicks the mouse button then our program may do something.

Reacting to events that do not need explicit user's interaction. For example, on ticking of a timer our program may want to do something.

parallel function for every DOS/BIOS function. Hence at some point of time one has to learn how to call DOS/BIOS functions.

Directly interacting with the hardware

At times the programs are needed to directly interact with the hardware. This has to be done because either there are no library functions or DOS/BIOS functions to do this, or if they are there their reach is limited. For example, while writing good video games one is required to watch the status of multiple keys simultaneously. The library functions as well as the DOS/BIOS functions are unable to do this. At such times we have to interact with the keyboard controller chip directly. However, direct interaction with the hardware is difficult because one has to have good knowledge of technical details of the chip to be able to do so. Moreover, not every technical detail about how the hardware from a particular manufacturer works is well documented.

Hardware Interaction, Windows Perspective

Like DOS, under Windows too a hardware interrupt gets generated whenever an external event occurs. As a reaction to this signal a table called Interrupt Descriptor Table (IDT) is looked up and a corresponding routine for the interrupt gets called. Unlike DOS the IDT contains addresses of various kernel routines (instead of BIOS routines). These routines are part of the Windows OS itself. When the kernel routine is called, it in turn calls the ISR present in the appropriate device driver. This ISR interacts with the hardware.

Two questions may now occur to you:

(a) Why the kernel routine does not interact with the hardware directly?

(b) Why the ISR of the device driver not registered directly in the IDT?

Let us find answer to the first question. Every piece of hardware works differently than the other. As new pieces of hardware come into existence new code has to be written to be able to interact with them. If this code is written in the kernel then the kernel would have to be rewritten and recompiled every time a new hardware comes into existence. This is practically impossible. Hence the new code to interact with the device is written in a separate program called device driver. With every new piece of hardware a new device driver is provided. This device driver is an extension of the OS itself.

Let us now answer the second question. Out of the several components of Windows OS a component called kernel is tightly integrated with the processor architecture. If the processor architecture changes then the kernel is bound to change. One of goals of Windows NT family was to keep the other components of OS and the device drivers portable across different microprocessor

parameter is the number of sectors that we wish to read. This parameter is specified as 1 since the boot sector occupies only a single sector. The last parameter is the address of a buffer/variable that would collect the data that is read from the floppy. Here we have passed the address of the boot structure variable **b**. As a result, the structure variable would be setup with the contents of the boot sector data at the end of the function call.

Once the contents of the boot sector have been read into the structure variable **b** we have displayed the first few of them on the screen using **printf( )**. If you wish you can print the rest of the contents as well.

The *ReadSector( )* Function

With the preliminaries over let us now concentrate on the real stuff in this program, i.e. the **ReadSector( )** function. This function begins by making a call to the **CreateFile( )** API function as shown below:

```
h = CreateFile ( src, GENERIC_READ,
FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0 ) ;
```

The **CreateFile( )** API function is very versatile. Anytime we are to communicate with a device we have to firstly call this API function. The **CreateFile( )** function opens the specified device as a file. Windows treats all devices just like files on disk. Reading from this file means reading from the device.

The **CreateFile( )** API function takes several parameters. The first parameter is the string specifying the device to be opened. The second parameter is a set of flags that are used to specify the desired access to the file (representing the device) about to be opened. By specifying the **GENERIC_READ** flag we have indicated that we just wish to read from the file (device). The third parameter specifies the sharing access for the file (device). Since floppy drive is a shared resource across all the running applications we have specified the **FILE_SHARE_READ** flag. In general while interacting with any hardware the sharing flag for the file (device) must always be set to this value since every piece of hardware is shared amongst all the running applications. The fourth parameter indicates security access for the file (device). Since we are not concerned with security here we have specified the value as **0**. The fifth parameter specifies what action to take if

the file already exists. When using **CreateFile( )** for device access we must always specify this parameter as **OPEN_EXISTING**. Since the floppy disk file was already opened by the OS a long time back during the booting. The remaining two parameters are not used when using **CreateFile( )** API function for device access. Hence we have passed a **0** value for them. If the call to **CreateFile( )** succeeds then we obtain a handle to the file (device). The device file mechanism allows us to read from the file (device) by setting the file pointer using the **SetFilePointer( )** API function

and then reading the file using the **ReadFile( )** API function. Since every sector is **512** bytes long, to read from the nth sector we need to set the file pointer to the **512 * n** bytes from the start of the file. The first parameter to **SetFilePointer( )** is the handle of the device file that we obtained by calling the **CreateFile( )** function. The second parameter is the byte offset from where the reading is to begin. This second parameter is relative to the third parameter. We have specified the third parameter as **FILE_BEGIN** which means the byte offset is relative to the start of the file.

To actually read from the device file we have made a call to the **ReadFile( )** API function. The **ReadFile( )** function is very easy to use. The first parameter is the handle of the file (device), the second parameter is the address of a buffer where the read contents should be dumped. The third parameter is the count of bytes that have to be read. We have specified the value as **512 * num** so as to read **num** sectors. The fourth parameter to **ReadFile( )** is the address of an **unsigned int** variable which is set up with the count of bytes that the function was successfully able to read. Lastly, once our work with the device is over we should close the file (device) using the **CloseHandle( )** API function.

Though **ReadSector( )** doesn't need it, there does exist a counterpart of the **ReadFile( )** function. Its name is **WriteFile( )**. This API function can be used to write to the file (device). The parameters of **WriteFile( )** are same as that of **ReadFile( )**. Note

that when **WriteFile( )** is to be used we need to specify the **GENERIC_WRITE** flag in the call to **CreateFile( )** API function. Given below is the code of **WriteSector( )** function that works exactly opposite to the **ReadSector( )** function.

```
void WriteSector ( char *src, int ss, int num, void* buff )
{
HANDLE h ;
unsigned int br ;
h = CreateFile ( src, GENERIC_WRITE,
FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0 ) ;
SetFilePointer ( h, ( ss * 512 ), NULL, FILE_BEGIN ) ;
WriteFile ( h, buff, 512 * num, &br, NULL ) )
CloseHandle ( h ) ;
}
```

Accessing Other Storage Devices

Note that the mechanism of reading from or writing to any device remains standard under Windows. We simply need to change the string that specifies the device. Here are some sample calls for reading/writing from/to various devices:

```
ReadSector ( "\\\\.\\a:", 0, 1, &b ) ; /* reading from 2nd floppy drive */
ReadSector ( "\\\\.\\d:", 0, 1, buffer ) ; /* reading from a CD-ROM drive */
WriteSector ( "\\\\.\\c:", 0, 1, &b ) ; /* writing to a hard disk */
ReadSector ( "\\\\.\\physicaldrive0", 0, 1, &b ) ; /* reading partition table */
```

Here are a few interesting points that you must note.

(a)

```
hkb = SetWindowsHookEx ( WH_KEYBOARD,
( HOOKPROC ) KeyboardProc, ( HINSTANCE ) h, 0 ) ;
if ( hkb == NULL )
return FALSE ;
return TRUE ;
}
LRESULT __declspec ( dllexport ) __stdcall KeyboardProc ( int nCode,
WPARAM wParam, LPARAM lParam )
{
short int state ;
if ( nCode < 0 )
return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;
if ( ( nCode == HC_ACTION ) &&
( ( DWORD ) lParam & 0x40000000 ) )
{
state = GetKeyState ( VK_CAPITAL ) ;
if ( (state & 1 )== 0) /* if off */
```



```
{
keybd_event ( VK_CAPITAL , 0,
KEYEVENTF_EXTENDEDKEY, 0 ) ;
keybd_event ( VK_CAPITAL , 0,
KEYEVENTF_EXTENDEDKEY | KEYEVENTF_KEYUP, 0 ) ;
}
}
return CallNextHookEx ( hkb, nCode, wParam, lParam ) ;
}
BOOL __declspec ( dllexport ) removeHook( )
{
return UnhookWindowsHookEx ( hkb ) ;
}
```


Follow the steps mentioned below to create this program:

(a)
(b)
(c)
(d)
(e)
(f)

Select ‘File | New’ option to start a new project in VC++.
From the ‘Project’ tab select ‘Win32 Dynamic-Link Library’ and click on the ‘Next’ button.
In the ‘Win32 Dynamic-link Library Step 1 of 1’ select “An empty DLL project” and click on the ‘Finish’ button.
Select ‘File | New’ option.
From the ‘File’ tab select ‘C++ source file’ and give the file name as ‘hook.c’. Type the code listed above in this file.
Compile the program to generate the .DLL file.
Note that this program doesn’t contain **WinMain( )** since the program on compilation should not execute on its own. It has been replaced by a function called **DllMain( )**. This function acts as

Hardware interaction can happen in two ways: (1) When the user interacts with the hardware and the program reacts to it. (2) When the program interacts with the hardware without any user intervention.
In DOS when the user interacts with the hardware an ISR gets called which interacts with the hardware. In Windows the same thing is done by the device driver's ISR.
In DOS when the program has to interact with the hardware it can do so by using library functions, DOS/BIOS routines or by directly interacting with the hardware. In Windows the same thing can be done by using API functions.
Under Windows to gain finer control over the hardware we are required to write a device driver program.
Interaction with the any device can be done using API functions like **CreateFile( )**, **ReadFile( )**, **WriteFile( )** and **CloseHandle( )**.
Different strings have to be passed to the **CreateFile( )** functions for interacting with different devices.
Windows provides a powerful mechanism called hooks that can alter the flow of messages before they reach the application.
Windows hook procedures should be written in a DLL since they work on a system wide basis.
Windows hooks can be put to many good uses.

Exercise

[A] State True or False:
In MS-DOS on occurrence of an interrupt values from IDT are used to call the appropriate kernel routine.
Under Windows on occurrence of an interrupt the kernel routine calls the appropriate device driver's ISR.
Under Windows an application can interact with the hardware by directly calling its device driver's routines.

(d)
(e)
(f)
(g)
(h)
(a)
(b)
(c)
(d)
(e)
(f)
(g)
Under Windows we can write device drivers to extend the OS itself.
ReadSector( ) and **WriteSector( )** are API functions.
While reading a sector from the disk the **CreateFile( )** function creates a file on the disk.

might be different under different OS. For example, a **printf( )** would work under all OSs, but the way it is defined is likely to be different for different OSs. The programmer however doesn't suffer because of this since he can continue to call **printf( )** the same way no matter how it is implemented.

But there the similarity ends. If we are to build programs that utilize the features offered by the OS then things are bound to be different across OSs. For example, if we are to write a C program that would create a Window and display a message "hello" at the point where the user clicks the left mouse button. The architecture of this program would be very closely tied with the OS under which it is being built. This is because the mechanisms for creating a window, reporting a mouse click, handling a mouse click, displaying the message, closing the window, etc. are very closely tied with the OS for which the program is being built. In short the programming architecture (better known as programming model) for each OS is different. Hence naturally the program that achieves the same task under different OS would have to be different.

The 'Hello Linux' Program

As with any new platform we would begin our journey in the Linux world by creating a 'hello world' program. Here is the source code....

```
int main( )
{
printf ( "Hello Linux\n" ) ;
return 0 ;
}
```

The program is exactly same as compared to a console program under DOS/Windows. It begins with **main( )** and uses **printf( )** standard library function to produce its output. So what is the difference? The difference is in the way programs are typed, compiled and executed. The steps for typing, compiling and executing the program are discussed below.

The first hurdle to cross is the typing of this program. Though any editor can be used to do so, we have preferred to use the editor called 'KWrite'. This is because it is a very simple yet elegant

editor compared to other editors like 'vi' or 'emacs'. Note that KWrite is a text editor and is a part of K Desktop environment (KDE). Installation of Linux and KDE is discussed in Appendix H. Once KDE is started select the following command from the desktop panel to start KWrite:

K Menu | Accessories | More Accessories | KWrite

If you face any difficulty in starting the KWrite editor please refer Appendix H. Assuming that you have been able to start KWrite successfully, carry out the following steps:

(a)

(b)

child or parent) attempt to change the value of a variable it is no longer shared. Instead a new copy of the variable is made for the process that is attempting to change it. This not only ensures data integrity but also saves precious memory.

Summary

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(a)
(b)
(c)

Linux is a free OS whose kernel was built by Linus Torvalds and friends.

A Linux distribution consists of the kernel with source code along with a large collection of applications, libraries, scripts etc.

C programs under Linux can be compiled using the popular **gcc** compiler.

Basic scheduling unit in Linux is a 'Process'. Processes are scheduled by a special program called 'Scheduler'.

fork( ) library function can be used to create child processes.

Init process is the father of all processes.

execl( ) library function is used to execute another program from within a running program,.

execl( ) function overwrites the image (code and data) of the calling process.

execl( ) and **fork( )** usually go hand in hand.

ps command can be used to get a list of all processes.

kill command can be used to terminate a process.

A 'Zombie' is a child process that has terminated but its parent is running and has not called a function to get the exit code of the child process.

An 'Orphan' is a child process whose parent has terminated. Orphaned processes are adopted by **init** process automatically.

A parent process can avoid creation of a Zombie and Orphan processes using **waitpid( )** function.

by sending a signal to our program. Since we have done nothing to handle this signal the default signal handler gets called. In this

default signal handler there is code to terminate the program. Hence on pressing Ctrl + C the program gets terminated.

But how on earth would the default signal handler get called. Well, it is simple. There are several signals that can be sent to a program. A unique number is associated with each signal. To avoid remembering these numbers, they have been defined as macros like **SIGINT**, **SIGKILL**, **SIGCONT**, etc. in the file 'signal.h'. Every process contains several 'signal ID - function pointer' pairs indicating for which signal which function should be called. If we do not decide to handle a signal then against that signal ID the address of the default signal handler function is present. It is precisely this default signal handler for **SIGINT** that got called when we pressed Ctrl + C when the above program was executed. INT in **SIGINT** stands for interrupt.

Let us know see how can we prevent the termination of our program even after hitting Ctrl + C. This is shown in the following program:

```
# include <signal.h>
void sighandler ( int signum )
{
printf ( "SIGINT received. Inside sighandler\n" ) ;
}
int main( )
{
signal ( SIGINT, ( void* ) sighandler ) ;
while ( 1 )
printf ( "Program Running\n" ) ;
return 0 ;
}
```

In this program we have registered a signal handler for the SIGINT signal by using the **signal( )** library function. The first parameter

of this function specifies the ID of the signal that we wish to register. The second parameter is the address of a function that should get called whenever the signal is received by our program. This address has to be typecasted to a **void *** before passing it to the **signal( )** function.

Now when we press Ctrl + C the registered handler, namely, **sighandler( )** would get called. This function would display the message 'SIGINT received. Inside sighandler' and return the control back to **main( )**. Note that unlike the default handler, our handler does not terminate the execution of our program. So only way to terminate it is to kill the running process from a different terminal. For this we need to open a new instance of command prompt (terminal). How to start a new instace of command prompt is discussed in Appendix H. Next do a **ps –a** to obtain the list of processes running at all the command prompts that we have

```
{
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.

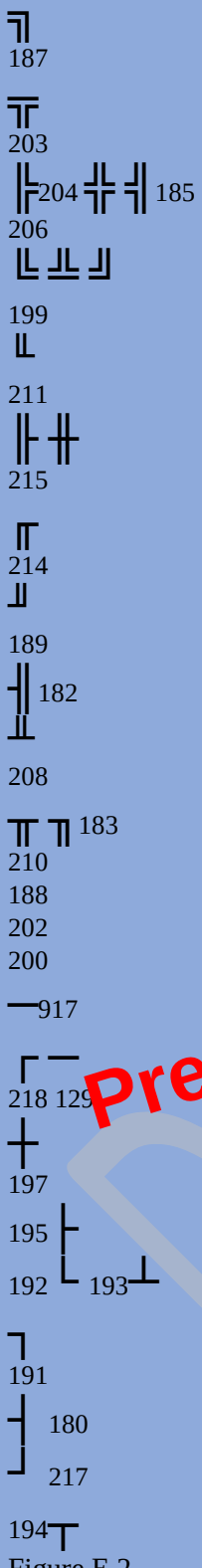

Figure E.2

Value Char Value Char Value Char Value Char Value Char Value Char
0 22 ▬ 44 , 66 B 88 X 110 n
1 ☺ 23 ↕ 45 - 67 C 89 Y 111 o
2 ☻ 24 ↑ 46 . 68 D 90 Z 112 p

```
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

he disk drives in DOS and Windows are organized as zerobased 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