Before we can begin to write serious programs in C, it would be interesting to find out what really is C, how it came into existence and how does it compare with other computer languages. In this chapter we would briefly outline these issues.

Four important aspects of any language are the way it stores data, the way it operates upon this data, how it accomplishes input and output and how it lets you control the sequence of execution of instructions in a program. We would discuss the first three of these building blocks in this chapter.

What is C

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc. No one pushed C. It wasn't made the 'official' Bell Labs language. Thus, without any advertisement C's reputation spread and its pool of users grew. Ritchie seems to have been rather surprised that so many programmers preferred C to older languages like FORTRAN or PL/I, or the newer ones like Pascal and APL. But, that's what happened.

Possibly why C seems so popular is because it is reliable, simple and easy to use. Moreover, in an industry where newer languages, tools and technologies emerge and vanish day in and day out, a language that has survived for more than 3 decades has to be really good.

An opinion that is often heard today is – "C has been already superceded by languages like C++, C# and Java, so why bother to a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program. This is illustrated in the Figure 1.1.

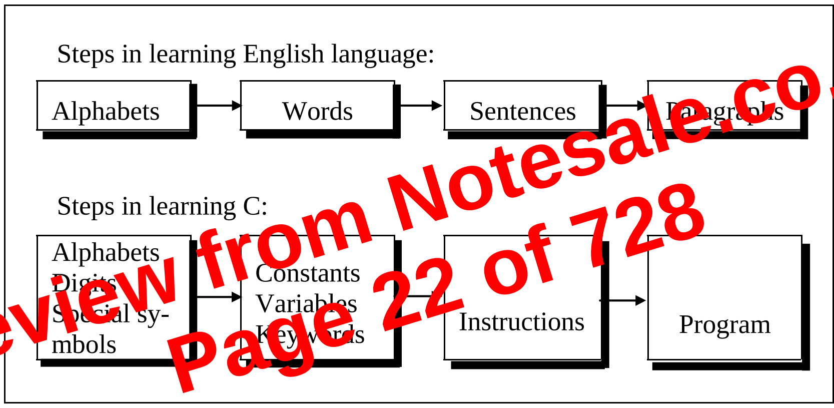

Figure 1.1

The C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. Figure 1.2 shows the valid alphabets, numbers and special symbols allowed in C.

(b) The statements in a program must appear in the same order in which we wish them to be executed; unless of course the logic of the problem demands a deliberate 'jump' or transfer of control to a statement, which is out of sequence.

(c) Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.

(d) All statements are entered in small case letters.

(e) C has no specific rules for the position at which a statement is to be written. That's why it is often called a free-form language.

(f) Every C statement must end with a **;**. Thus **;** acts as a statement terminator.

Let us now write down our first C program. It would simply calculate simple interest for a set of values representing principle, number of years and rate of interest.


```
/* Calculation of simple interest */
/* Author gekay Date: 25/05/2004 */
main( )
{
    int  p, n ;
    float  r, si ;

    p = 1000 ;
    n = 3 ;
    r = 8.5 ;

    /* formula for simple interest */
    si = p * n * r / 100 ;

    printf ( "%f" , si ) ;
```

– Although a lot of comments are probably not necessary in this program, it is usually the case that programmers tend to use too few comments rather than too many. An adequate number of comments can save hours of misery and suffering when you later try to figure out what the program does.

– The normal language rules do not apply to text written within **/* .. */**. Thus we can type this text in small case, capital or a combination. This is because the comments are solely given for the understanding of the programmer or the fellow programmers and are completely ignored by the compiler.

– Comments cannot be nested. For example,

```
/* Cal of SI /* Author sam date 01/01/2002 */ */
```

is invalid.

– A comment can be split over more than one line, as in,

```
/* This is
   a jazzy
   comment */
```

Such a comment is often called a multi-line comment.

– **main( )** is a collective name given to a set of statements. This name has to be **main( )**, it cannot be anything else. All statements that belong to **main( )** are enclosed within a pair of braces { } as shown below.

```
main( )
{
    statement 1 ;
    statement 2 ;
```

function is **printf( )**. We have used it display on the screen the value contained in **si**.

The general form of **printf( )** function is,

```
printf ( "<format string>", <list of variables> ) ;
```

```
<format string> can contain,

%f  for printing real values
%d  for printing integer values
%c  for printing character values
```

In addition to format specifiers like **%f**, **%d** and **%c** the format string may also contain any other characters. These characters are printed as they are when the **printf( )** is executed.

Following are some examples of usage of **printf( )** function:

```
printf ( "%f", si ) ;
printf ( "%d %d %f %f", p, n, r, si ) ;
printf ( "Simple interest = Rs. %f", si ) ;
printf ( "Prin = %d \nRate = %f", p, r ) ;
```


The output of the last statement would look like this...

```
Prin = 1000
Rate = 8.5
```

What is ‘\n’ doing in this statement? It is called newline and it takes the cursor to the next line. Therefore, you get the output split over two lines. ‘\n’ is one of the several Escape Sequences available in C. These are discussed in detail in Chapter 11. Right now, all that we can say is ‘\n’ comes in

Receiving Input

In the program discussed above we assumed the values of **p**, **n** and **r** to be 1000, 3 and 8.5. Every time we run the program we would get the same value for simple interest. If we want to calculate simple interest for some other set of values then we are required to make the relevant change in the program, and again compile and execute it. Thus the program is not general enough to calculate simple interest for any set of values without being required to make a change in the program. Moreover, if you distribute the EXE file of this program to somebody he would not even be able to make changes in the program. Hence it is a good practice to create a program that is general enough to work for any set of values.

To make the program general the program itself should ask the user to supply the values of **p**, **n** and **r** through the keyboard during execution. This can be achieved using a function called **scanf( )**. This function is a counter-part of the **printf( )** function. **printf( )** outputs the values to the screen whereas **scanf( )** receives them from the keyboard. This is illustrated in the program shown below.

```
/* Calculation of simple interest */
/* Author gekay Date 25/05/2004 */
main( )
{
    int   p, n ;
    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 ) ;
}
```

is not. This is because here we are trying to use **a** even before defining it.

(c) The following statements would work

```
int a, b, c, d ;
a = b = c = 10 ;
```

However, the following statement would not work

```
int a = b = c = d = 10 ;
```

Once again we are trying to use **b** (to assign to **a**) before defining it.

Arithmetic Instruction

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like +, -, *****, and /.

```
Ex.: int ad ;
     float  kot, deta, alpha, beta, gamma ;
     ad = 3200 ;
     kot = 0.0056 ;
     deta = alpha * beta / gamma + 3.2 * 2 / 5 ;
```


Here,

***, /, -, +** are the arithmetic operators.
= is the assignment operator.
2, 5 and 3200 are integer constants.
3.2 and 0.0056 are real constants.
ad is an integer variable.
kot, deta, alpha, beta, gamma are real variables.

Arithmetic Instruction	Result	Arithmetic Instruction	Result
k = 2 / 9	0	a = 2 / 9	0.0
k = 2.0 / 9	0	a = 2.0 / 9	0.2222
k = 2 / 9.0	0	a = 2 / 9.0	0.2222
k = 2.0 / 9.0	0	a = 2.0 / 9.0	0.2222
k = 9 / 2	4	a = 9 / 2	4.0
k = 9.0 / 2	4	a = 9.0 / 2	4.5
k = 9 / 2.0	4	a = 9 / 2.0	4.5
k = 9.0 / 2.0	4	a = 9.0 / 2.0	4.5

Figure 1.7

Note that though the following statements give the same result, 0, the results are obtained differently.

```
k = 2 / 9 ;
k = 2.0 / 9 ;
```

In the first statement, since both 2 and 9 are integers, the result is an integer, i.e. 0. This 0 is then assigned to **k**. In the second statement 9 is promoted to 9.0 and then the division is performed. Division yields 0.222222. However, this cannot be stored in **k**, **k** being an **int**. Hence it gets demoted to 0 and then stored in **k**.

Hierarchy of Operations

While executing an arithmetic statement, which has two or more operators, we may have some problems as to how exactly does it get executed. For example, does the expression 2 * x - 3 * y correspond to (2x)-(3y) or to 2(x-3y)? Similarly, does A / B * C correspond to A / (B * C) or to (A / B) * C? To answer these questions satisfactorily one has to understand the 'hierarchy' of operations. The priority or precedence in which the operations in

(g) si = principal * rateofinterest * numberofyears / 100 ;

(h) area = 3.14 * r ** 2 ;

(i) volume = 3.14 * r ^ 2 * h ;

(j) k = ((a * b) + c) (2.5 * a + b) ;

(k) a = b = 3 = 4 ;

(l) count = count + 1 ;

(m) date = '2 Mar 04' ;

[C] Evaluate the following expressions and show their hierarchy.

(a) g = big / 2 + big * 4 / big - big + abc / 3 ;
(abc = 2.5, big = 2, assume **g** to be a float)

(b) on = ink * act / 2 + 3 / 2 * act + 2 + tig ;
(ink = 4, act = 1, tig = 3.2, assume **on** to be an int)

(c) s = qui * add / 4 - 6 / 2 + 2 / 3 * 6 / god ;
(qui = 4, add = 2, god = 2, assume **s** to be an int)

(d) s = 1 / 3 * a / 4 - 6 / 2 + 2 / 3 * 6 / g ;
(a = 4, g = 3, assume **s** to be an int)

[D] Fill the following table for the expressions given below and then evaluate the result. A sample entry has been filled in the table for expression (a).

```
    int   i = 2, j = 3, k, l ;
    float   a, b ;
    k = i / j * j ;
    l = j / i * i ;
    a = i / j * j ;
    b = j / i * i ;
    printf( "%d %d %f %f", k, l, a, b ) ;
}
```

(b)
```
main( )
{
    int  a, b ;
    a = -3 - - 3 ;
    b = -3 - - ( - 3 ) ;
    printf ( "a = %d b = %d", a, b ) ;
}
```

(c)
```
main( )
{
    float a = 5,  b = 2 ;
    int c ;
    c = a % b ;
    printf ( "%d", c ) ;
}
```


(d)
```
main( )
{
    printf ( "nn \n\n nn\n" ) ;
    printf ( "nn /n/n nn/n" ) ;
}
```

(e)
```
main( )
{
    int a, b ;
    printf ( "Enter values of a and b" ) ;
    scanf ( " %d %d ", &a, &b ) ;
    printf ( "a  = %d b = %d", a, b ) ;
}
```

(b) The distance between two cities (in km.) is input through the keyboard. Write a program to convert and print this distance in meters, feet, inches and centimeters.

(c) If the marks obtained by a student in five different subjects are input through the keyboard, find out the aggregate marks and percentage marks obtained by the student. Assume that the maximum marks that can be obtained by a student in each subject is 100.

(d) Temperature of a city in Fahrenheit degrees is input through the keyboard. Write a program to convert this temperature into Centigrade degrees.

(e) The length & breadth of a rectangle and radius of a circle are input through the keyboard. Write a program to calculate the area & perimeter of the rectangle, and the area & circumference of the circle.

(f) Two numbers are input through the keyboard into two locations C and D. Write a program to interchange the contents of C and D.

(g) If a five-digit number is input through the keyboard, write a program to calculate the sum of its digits.

(Hint: Use the modulus operator '%')

(h) If a five-digit number is input through the keyboard, write a program to reverse the number.

(i) If a four-digit number is input through the keyboard, write a program to obtain the sum of the first and last digit of this number.

(j) In a town, the percentage of men is 52. The percentage of total literacy is 48. If total percentage of literate men is 35 of the total population, write a program to find the total number

```
    if ( per >= 60 )
        printf ( "First division ") ;
    else
    {
        if ( per >= 50 )
            printf ( "Second division" ) ;
        else
        {
            if ( per >= 40 )
                printf ( "Third division" ) ;
            else
                printf ( "Fail" ) ;
        }
    }
}
```

This is a straight forward program. Observe that the program uses nested **if-else**s. This leads to three disadvantages:

(a) As the number of conditions go on increasing the level of indentation also goes on increasing. As a result the whole program creeps to the right.

(b) Care needs to be exercised to match the corresponding **if**s and **else**s.

(c) Care needs to be exercised to match the corresponding pair of braces.

All these three problems can be eliminated by usage of ‘Logical operators’. The following program illustrates this.

```
/* Method – II */
main( )
{
    int  m1, m2, m3, m4, m5, per ;

    printf ( "Enter marks in five subjects " ) ;
    scanf ( "%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5 ) ;
    per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
```

to only two answers. For example, consider the following example:

Example 2.5: A company insures its drivers in the following cases:

– If the driver is married.
– If the driver is unmarried, male & above 30 years of age.
– If the driver is unmarried, female & above 25 years of age.

In all other cases the driver is not insured. If the marital status, sex and age of the driver are the inputs, write a program to determine whether the driver is to be insured or not.

Here after checking a complicated set of instructions the final output of the program would be one of the two—Either the driver should be ensured or the driver should not be ensured. As mentioned above, since these are the only two outcomes this problem can be solved using logical operators. But before we do that let us write a program that does not make use of logical operators.


```
/* Insurance of driver - without using logical operators */
main( )
{
    char  sex, ms ;
    int  age ;

    printf ( "Enter age, sex, marital status " ) ;
    scanf ( "%d %c %c", &age, &sex, &ms ) ;

    if ( ms == 'M' )
        printf ( "Driver is insured" ) ;
    else
    {
        if ( sex == 'M' )
        {
```

```
    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**
0	0	1	1	0	0
0	non-zero	1	0	0	0
non-zero	0	0	1	0	1
non-zero	non-zero	0	0	1	1

Figure 2.8

(a) It's not necessary that the conditional operators should be used only in arithmetic statements. This is illustrated in the following examples:

Ex.:
```
int  i ;
scanf ( "%d", &i ) ;
( i == 1 ? printf ( "Amit" ) : printf ( "All and sundry" ) ) ;
```

Ex.:
```
char  a = 'z' ;
printf ( "%c" , ( a >= 'a' ? a : '!' ) ) ;
```

(b) The conditional operators can be nested as shown below.

```
int  big, a, b, c ;
big = ( a > b ? ( a > c ? 3: 4 ) : ( b > c ? 6: 8 ) ) ;
```

(c) Check out the following conditional expression:

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

This will give you an error 'Lvalue Required'. The error can be overcome by enclosing the statement in the **:** part within a pair of parentheses. This is shown below:

```
a > b ? g = a : ( g = b ) ;
```

In absence of parentheses the compiler believes that **b** is being assigned to the result of the expression to the left of second =. Hence it reports an error.

The limitation of the conditional operators is that after the **?** or after the **:** only one C statement can occur. In practice rarely is this the requirement. Therefore, in serious C programming conditional operators aren't as frequently used as the **if-else**.

Summary

(a) There are three ways for taking decisions in a program. First way is to use the **if-else** statement, second way is to use the

```
}
```

[H] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int  tag = 0, code = 1 ;
    if ( tag == 0 )
        ( code > 1 ? printf ( "\nHello" ) ? printf ( "\nHi" ) ) ;
    else
        printf ( "\nHello Hi !!" ) ;
}
```

(b)
```
main( )
{
    int  ji = 65 ;
    printf ( "\nji >= 65 ? %d : %c", ji ) ;
}
```

(c)
```
main( )
{
    int  i = 10, j ;
    i >= 5 ? ( j = 10 ) : ( j = 15 ) ;
    printf ( "\n%d %d", i, j ) ;
}
```


(d)
```
main( )
{
    int a = 5 , b = 6 ;
    ( a == b ? printf( "%d",a) ) ;
}
```

(e)
```
main( )
{
    int n = 9 ;
    ( n == 9 ? printf( "You are correct" ) ; : printf( "You are wrong" ) ;) ;
}
```

(f)
```
main( )
{
    int  kk = 65 ,ll ;
    ll = ( kk == 65 : printf ( "\n kk is equal to 65" ) : printf ( "\n kk is not
equal to 65" ) ) ;
    printf( "%d", ll ) ;
}
```

(g)
```
main( )
{
    int  x = 10, y = 20 ;
    x == 20 && y != 10 ? printf( "True" ) : printf( "False" ) ;
}
```

[I] Rewrite the following programs using conditional operators.

(a)
```
main( )
{
    int  x, min, max ;
    scanf ( "\n%d %d", &max, &x ) ;
    if ( x > max )
        max = x ;
    else
        min = x ;
}
```


(b)
```
main( )
{
    int  code ;
    scanf ( "%d", &code ) ;
    if ( code > 1 )
        printf ( "\nJerusalem" ) ;
    else
        if ( code < 1 )
            printf ( "\nEddie" ) ;
        else
            printf ( "\nC Brain" ) ;
}
```

Here, both, the comparison and the incrementation is done through the same statement, **++i <= 10**. Since ++ precedes **i** firstly incrementation is done, followed by comparison. Note that it is necessary to initialize **i** to 0.

Nesting of Loops

The way **if** statements can be nested, similarly **while**s and **for**s can also be nested. To understand how nested loops work, look at the program given below:

```
/* Demonstration of nested loops */
main( )
{
    int   r, c, sum ;
    for ( r = 1 ; r <= 3 ; r++ )  /* outer loop */
    {
        for ( c = 1 ; c <= 2 ; c++ )  /* inner loop */
        {
            sum = r + c ;
            printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;
        }
    }
}
```


When you run this program you will get the following output:

```
r = 1 c = 1 sum = 2
r = 1 c = 2 sum = 3
r = 2 c = 1 sum = 3
r = 2 c = 2 sum = 4
r = 3 c = 1 sum = 4
r = 3 c = 2 sum = 5
```

Here, for each value of **r** the inner loop is cycled through twice, with the variable **c** taking values from 1 to 2. The inner loop

Though it is simpler to program such a requirement using a **do-while** loop, the same functionality if required, can also be accomplished using **for** and **while** loops as shown below:

```
/* odd loop using a for loop */
main( )
{
	char  another = 'y' ;
	int  num ;
	for ( ; another == 'y' ; )
	{
		printf ( "Enter a number " ) ;
		scanf ( "%d", &num ) ;
		printf ( "square of %d is %d", num, num * num ) ;
		printf ( "\nWant to enter another number y/n " ) ;
		scanf ( " %c", &another ) ;
	}
}
```

```
/* odd loop using a while loop */
main( )
{
	char  another = 'y' ;
	int  num ;

	while ( another == 'y' )
	{
		printf ( "Enter a number " ) ;
		scanf ( "%d", &num ) ;
		printf ( "square of %d is %d", num, num * num ) ;
		printf ( "\nWant to enter another number y/n " ) ;
		scanf ( " %c", &another ) ;
	}
}
```

Note that when the value of **i** equals that of **j**, the **continue** statement takes the control to the **for** loop (inner) bypassing rest of the statements pending execution in the **for** loop (inner).

The *do-while* Loop

The **do-while** loop looks like this:

```
do
{
      this ;
      and this ;
      and this ;
      and this ;
} while ( this condition is true ) ;
```

There is a minor difference between the working of **while** and **do-while** loops. This difference is the place where the condition is tested. The **while** tests the condition before executing any of the statements within the **while** loop. As against this, the **do-while** tests the condition after having executed the statements within the loop. Figure 3.5 would clarify the execution of **do-while** loop still further.

In real life we are often faced with situations where we are 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.

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 **switch-case-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 more so if there are multiple statements within each **case** of a **switch**.

(h) We can check the value of any expression in a **switch**. Thus the following **switch** statements are legal.

```
switch ( i + j * k )
switch ( 23 + 45 % 4 * k )
switch ( a < 4 && b > 7 )
```

Expressions can also be used in cases provided they are constant expressions. Thus **case 3 + 7** is correct, however, **case a + b** is incorrect.

(i) The **break** statement when used in a **switch** takes the control outside the **switch**. However, use of **continue** will not take the control to the beginning of **switch** as one is likely to believe.

(j) In principle, a **switch** may occur within another, but in practice it is rarely done. Such statements would be called nested **switch** statements.

(k) The **switch** statement is very useful while writing menu driven programs. This aspect of **switch** is discussed in the exercise at the end of this chapter.

switch Versus *if-else* Ladder

There are some things that you simply cannot do with a **switch**. These are:

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

```
}
```

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 the 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.

Exercise

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

(a)
```
main( )
{
    char  suite = 3 ;
    switch ( suite )
    {
        case 1 :
            printf ( "\nDiamond" ) ;
        case 2 :
            printf ( "\nSpade" ) ;
        default :
            printf ( "\nHeart") ;
    }
    printf ( "\nI thought one wears a suite" ) ;
}
```

(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" ) ;
    }
}
```

(e)
```
main( )
{
    int  k ;
    float j = 2.0 ;
```

```
}
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) A function can call itself. Such a process is called ‘recursion’. We would discuss this aspect of C functions later in this chapter.

(h) 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" ) ;
    }
}
```

(i) 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

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
```

Here, the **gospel( )** function has been defined to return **void**; means it would return nothing. Therefore, it would just flash the four messages about viruses and return the control back to the **main( )** function.

Call by Value and Call by Reference

By now we are well familiar with how to call functions. But, if you observe carefully, whenever we called a function and passed something to it we have always passed the 'values' of variables to the called function. Such function calls are called 'calls by value'. By this what we mean is, on calling a function we are passing values of variables to it. The examples of call by value are shown below:

```
sum = calsum ( a, b, c ) ;
f = factr ( a ) ;
```

We have also learnt that variables are stored somewhere in memory. So instead of passing the value of a variable, can we not pass the location number (also called address) of the variable to a function? If we were able to do so it would become a 'call by reference'. What purpose a 'call by reference' serves we would find out a little later. First we must equip ourselves with knowledge of how to make a 'call by reference'. This feature of C functions needs at least an elementary knowledge of a concept called 'pointers'. So let us first acquire the basics of pointers after which we would take up this topic once again.

An Introduction to Pointers

Which feature of C do beginners find most difficult to understand? The answer is easy: pointers. Other languages have pointers but few use them so frequently as C does. And why not? It is C's clever use of pointers that makes it the excellent language it is.

A stack is a Last In First Out (LIFO) data structure. This means that the last item to get stored on the stack (often called Push operation) is the first one to get out of it (often called as Pop operation). You can compare this to the stack of plates in a cafeteria—the last plate that goes on the stack is the first one to get out of it. Now let us see how the stack works in case of the following program.

```
main( )
{
    int a = 5, b = 2, c ;
    c = add ( a, b ) ;
    printf ( "sum = %d", c ) ;
}
add ( int i, int j )
{
    int sum ;
    sum = i + j ;
    return sum ;
}
```


In this program before transferring the execution control to the function **fun( )** the values of parameters **a** and **b** are pushed onto the stack. Following this the address of the statement **printf( )** is pushed on the stack and the control is transferred to **fun( )**. It is necessary to push this address on the stack. In **fun( )** the values of **a** and **b** that were pushed on the stack are referred as **i** and **j**. In **fun( )** the local variable **sum** gets pushed on the stack. When value of **sum** is returned **sum** is popped up from the stack. Next the address of the statement where the control should be returned is popped up from the stack. Using this address the control returns to the **printf( )** statement in **main( )**. Before execution of **printf( )** begins the two integers that were earlier pushed on the stack are now popped off.

How the values are being pushed and popped even though we didn't write any code to do so? Simple—the compiler on encountering the function call would generate code to push parameters and the address. Similarly, it would generate code to clear the stack when the control returns back from **fun( )**. Figure 5.5 shows the contents of the stack at different stages of execution.

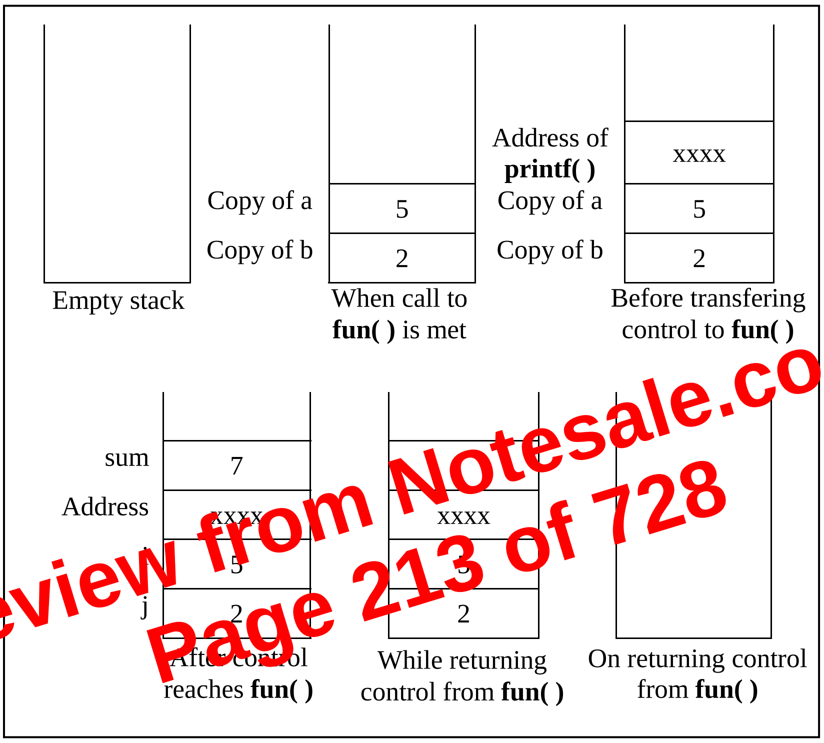

Figure 5.5

Note that in this program popping of **sum** and address is done by **fun( )**, whereas popping of the two integers is done by **main( )**. When it is done this way it is known as ‘CDecl Calling Convention’. There are other calling conventions as well where instead of **main( )**, **fun( )** itself clears the two integers. The calling convention also decides whether the parameters being passed to the function are pushed on the stack in left-to-right or right-to-left order. The standard calling convention always uses the right-to-left

```
        int  i = 3, j = 4, k, l ;
        k = addmult ( i, j ) ;
        l = addmult ( i, j ) ;
        printf ( "\n%d %d", k, l ) ;
    }
    addmult ( int  ii, int  jj )
    {
        int  kk, ll ;
        kk = ii + jj ;
        ll = ii * jj ;
        return ( kk, ll ) ;
    }
```

(b)
```
    main( )
    {
        int  a ;
        a = message( ) ;
    }
    message( )
    {
        printf ( "\nViruses are written in C" ) ;
        return ;
    }
```


(c)
```
    main( )
    {
        float  a = 15.5 ;
        char  ch = 'C' ;
        printit ( a, ch ) ;
    }
    printit ( a, ch )
    {
        printf ( "\n%f %c", a, ch ) ;
    }
```

(d)
```
    main( )
    {
        message( ) ;
```

(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

(d) Any year is entered through the keyboard. Write a function to determine whether the year is a leap year or not.

(e) A positive integer is entered through the keyboard. Write a function to obtain the prime factors of this number.

For example, prime factors of 24 are 2, 2, 2 and 3, whereas prime factors of 35 are 5 and 7.

Function Prototypes, Call by Value/Reference, Pointers

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

(a)
```
main( )
{
    float  area ;
    int  radius = 1 ;
    area = circle ( radius ) ;
    printf ( "\n%f", area ) ;
}
circle ( int  r )
```

```
{
    float  a ;
    a = 3.14 * r * r ;
    return ( a ) ;
}
```

(b)
```
main( )
{
    void  slogan( ) ;
    int  c = 5 ;
    c = slogan( ) ;
    printf ( "\n%d", c ) ;
}
void slogan( )
{
    printf ( "\nOnly He men use C!" ) ;
}
```

[F] Answer the following:

(a) Write a function which receives a **float** and an **int** from **main( )**, finds the product of these two and returns the product which is printed through **main( )**.

(b) Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from **main( )** and print the results in **main( )**.

(c) Write a function that receives marks received by a student in 3 subjects and returns the average and percentage of these marks. Call this function from **main( )** and print the results in **main( )**.

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

(a)
```
main( )
{
    int  i = 5, j = 2 ;
```

As seen in the first chapter the primary data types could be of three varieties—**char**, **int**, and **float**. It may seem odd to many, how C programmers manage with such a tiny set of data types. Fact is, the C programmers aren't really deprived. They can derive many data types from these three types. In fact, the number of data types that can be derived in C, is in principle, unlimited. A C programmer can always invent whatever data type he needs.

Not only this, the primary data types themselves could be of several types. For example, a **char** could be an **unsigned char** or a **signed char**. Or an **int** could be a **short int** or a **long int**. Sufficiently confusing? Well, let us take a closer look at these variations of primary data types in this chapter.

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 have 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 (16^{th}/32^{nd} 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 keyword **long**, as in,

```
long int  i ;
long int  abc ;
```

long integers cause the program to run a bit slower, but the range of values that we can use is expanded tremendously. The value of a **long** integer typically can vary from -2147483648 to +2147483647. More than this you should not need unless you are taking a world census.

If there are such things as **long**s, symmetry requires **short**s as well—integers that need less space in memory and thus help speed up program execution. **short** integer variables are declared as,

```
short int  j ;
short int  height ;
```

C allows the abbreviation of **short int** to **short** and of **long int** to **long**. So the declarations made above can be written as,

```
long  i ;
long  abc ;
short  j ;
short  height ;
```


Naturally, most C programmers prefer this short-cut.

Sometimes we come across situations where the constant is small enough to be an **int**, but still we want to give it as much storage as a **long**. In such cases we add the suffix 'L' or 'l' at the end of the number, as in 23L.

Integers, *signed* and *unsigned*

Sometimes, we know in advance that the value stored in a given integer variable will always be positive—when it is being used to overcome this difficulty? Would declaring **ch** as an **unsigned char** solve the problem? Even this would not serve the purpose since when **ch** reaches a value 255, **ch++** would try to make it 256 which cannot be stored in an **unsigned char**. Thus the only alternative is to declare **ch** as an **int**. However, if we are bent upon writing the program using **unsigned char**, it can be done as shown below. The program is definitely less elegant, but workable all the same.

```
main( )
{
    unsigned char  ch ;

    for ( ch = 0 ; ch <= 254 ; ch++ )
        printf ( "\n%d %c", ch, ch ) ;

    printf ( "\n%d %c", ch, ch ) ;
}
```

Floats and Doubles

A **float** occupies four bytes in memory and can range from -3.4e38 to +3.4e38. If this is insufficient then C offers a **double** data type that occupies 8 bytes in memory and has a range from -1.7e308 to +1.7e308. A variable of type **double** can be declared as,

```
double  a, population ;
```

If the situation demands usage of real numbers that lie even beyond the range offered by **double** data type, then there exists a **long double** that can range from -1.7e4932 to +1.7e4932. A **long double** occupies 10 bytes in memory.

You would see that most of the times in C programming one is required to use either **char**s or **int**s and cases where **float**s, **double**s or **long double**s would be used are indeed rare.

Let us now write a program that puts to use all the data types that we have learnt in this chapter. Go through the following program carefully, which shows how to use these different data types. Note the format specifiers used to input and output these data types.

```
main( )
{
    char  c ;
    unsigned char  d ;
    int  i ;
    unsigned int  j ;
    short  int  k ;
    unsigned  short  int  l ;
    long int  m ;
    unsigned long int  n ;
    float  x ;
    double  y ;
    long double  z ;

    /* char */
    scanf ( "%c %c", &c, &d ) ;
    printf ( "%c %c", c, d ) ;

    /* int */
    scanf ( "%d %u", &i, &j ) ;
    printf ( "%d %u", i, j ) ;

    /* short int */
    scanf ( "%d %u", &k, &l ) ;
    printf ( "%d %u", k, l ) ;

    /* long int */
    scanf ( "%ld %lu", &m, &n ) ;
    printf ( "%ld %lu", m, n ) ;

    /* float, double, long double */
    scanf ( "%f %lf %Lf", &x, &y, &z ) ;
    printf ( "%f %lf %Lf", x, y, z ) ;
```

accurately by VC++ compiler as compared to TC/TC++ compilers. This is because TC/TC++ targets its compiled code to 8088/8086 (16-bit) microprocessors. Since these microprocessors do not offer floating point support, TC/TC++ performs all float operations using a software piece called Floating Point Emulator. This emulator has limitations and hence produces less accurate results. Also, this emulator becomes part of the EXE file, thereby increasing its size. In addition to this increased size there is a performance penalty since this bigger code would take more time to execute.

(b) If you look at ranges of **char**s and **int**s there seems to be one extra number on the negative side. This is because a negative number is always stored as 2's compliment of its binary. For example, let us see how -128 is stored. Firstly, binary of 128 is calculated (10000000), then its 1's compliment is obtained (01111111). A 1's compliment is obtained by changing all 0s to 1s and 1s to 0s. Finally, 2's compliment of this number, i.e. 10000000, gets stored. A 2's compliment is obtained by adding 1 to the 1's compliment. Thus, for -128, 10000000 gets stored. This is an 8-bit number and it can be easily accommodated in a **char**. As against this, +128 cannot be stored in a **char** because its binary 010000000 (left-most 0 is for positive sign) is a 9-bit number. However +127 can be stored as its binary 01111111 turns out to be a 8-bit number.

(c) What happens when we attempt to store +128 in a **char**? The first number on the negative side, i.e. -128 gets stored. This is because from the 9-bit binary of +128, 010000000, only the right-most 8 bits get stored. But when 10000000 is stored the left-most bit is 1 and it is treated as a sign bit. Thus the value of the number becomes -128 since it is indeed the binary of -128, as can be understood from (b) above. Similarly, you can verify that an attempt to store +129 in a **char** results in storing -127 in it. In general, if we exceed the range from positive side we end up on the negative side. Vice versa is

(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 ) ;
    }
```

(i)
```
int x = 10 ;
main( )
{
    int x = 20 ;
    {
        int x  = 30 ;
        printf ( "\n%d", x ) ;
    }
    printf ("\n%d", x ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    long  num ;
    num = 2 ;
    printf ( "\n%ld", num ) ;
}
```

(b)
```
main( )
{
    char ch = 200 ;
    printf ( "\n%d", ch ) ;
}
```


(c)
```
main( )
{
    unsigned  a = 25 ;
    long unsigned  b = 25l ;
    printf ( "\n%lu %u", a, b ) ;
}
```

(d)
```
main( )
{
    long float  a = 25.345e454 ;
    unsigned double  b = 25 ;
    printf ( "\n%lf %d", a, b ) ;
```

```
}
```

(e)
```
main( )
{
    float  a = 25.345 ;
    float  *b ;
    b = &a ;
    printf ( "\n%f %u", a, b ) ;
}
```

(f)
```
static int y ;
main( )
{
    static int z ;
    printf ("%d %d", y, z ) ;
}
```

[C] State whether the following statements are True or False:

(a) Storage for a register storage class variable is allocated each time the control reaches the block in which it is present.

(b) An extern storage class variable is not available to the functions that precede its definition, unless the variable is explicitly declared in these functions.

(c) The value of an automatic storage class variable persists between various function invocations.

(d) If the CPU registers are not available, the register storage class variables are treated as static storage class variables.

(e) The register storage class variables cannot hold float values.

(f) If we try to use register storage class for a **float** variable the compiler will flash an error message.

(c) Conditional Compilation
(d) Miscellaneous directives

Let us understand these features of preprocessor one by one.

Macro Expansion

Have a look at the following program.

```
#define UPPER 25
main( )
{
    int  i ;
    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 ) ;
}
```

a dialog box appears. In this dialog box against ‘Include Directories’ we can specify the search path. We can also specify multiple include paths separated by ‘;’ (semicolon) as shown below:

```
c:\tc\lib ; c:\mylib ; d:\libfiles
```

The path can contain maximum of 127 characters. Both relative and absolute paths are valid. For example ‘..\dir\incfiles’ is a valid path.

Conditional Compilation

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**, which have the general form:

```
#ifdef macroname
    statement 1 ;
    statement 2 ;
    statement 3 ;
#endif
```


If **macroname** has been **#define**d, the block of code will be processed as usual; otherwise not.

Where would **#ifdef** be useful? When would you like to compile only a part of your program? In three cases:

(a) To “comment out” obsolete lines of code. It often happens that a program is changed at the last minute to satisfy a client. This involves rewriting some part of source code to the client’s satisfaction and deleting the old code. But veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was.

```
    #endif
    code common to both the computers
}
```

(c) Suppose a function **myfunc( )** is defined in a file 'myfile.h' which is **#include**d in a file 'myfile1.h'. Now in your program file if you **#include** both 'myfile.h' and 'myfile1.h' the compiler flashes an error 'Multiple declaration for **myfunc'**. This is because the same file 'myfile.h' gets included twice. To avoid this we can write following code in the header file.

```
/* myfile.h */
#ifndef __myfile_h
    #define __myfile_h

    myfunc( )
    {
        /* some code */
    }
#endif
```


First time the file 'myfile.h' gets included the preprocessor checks whether a macro called **__myfile_h** has been defined or not. If it has not been then it gets defined and the rest of the code gets included. Next time we attempt to include the same file, the inclusion is prevented since **__myfile_h** already stands defined. Note that there is nothing special about **__myfile_h**. In its place we can use any other macro as well.

#if and *#elif* Directives

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expression is nonzero, then subsequent lines upto a **#else**, **#elif** or **#endif** are compiled, otherwise they are skipped.

And here is the output of the program.

```
Inside fun1
Inside main
Inside fun2
```

Note that the functions **fun1( )** and **fun2( )** should neither receive nor return any value. If we want two functions to get executed at startup then their pragmas should be defined in the reverse order in which you want to get them called.

(b) **#pragma warn:** This directive tells the compiler whether or not we want to suppress a specific warning. Usage of this pragma is shown below.

```
#pragma warn –rvl     /* return value */
#pragma warn –par     /* parameter not used */
#pragma warn –rch     /* unreachable code */

int  f1( )
{
    int  a = 5 ;
}

void  f2 ( int  x )
{
    printf ( "\nInside f2" ) ;
}

int  f3( )
{
    int  x = 6 ;
    return x ;
    x++  ;
}

void  main( )
```

can be thought of as setting up an array of 5 elements, each of which is a one-dimensional array containing 2 integers. We refer to an element of a one-dimensional array using a single subscript. Similarly, if we can imagine **s** to be a one-dimensional array then we can refer to its zeroth element as **s[0]**, the next element as **s[1]** and so on. More specifically, **s[0]** gives the address of the zeroth one-dimensional array, **s[1]** gives the address of the first one-dimensional array and so on. This fact can be demonstrated by the following program.

```
/* Demo: 2-D array is an array of arrays */
main( )
{
    int  s[4][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
                    } ;
    int  i ;

    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\nAddress of %d th 1-D array = %u", i, s[i] ) ;
}
```

And here is the output...

```
Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65512
Address of 2 th 1-D array = 65516
Address of 3 th 1-D array = 65520
```

Let’s figure out how the program works. The compiler knows that **s** is an array containing 4 one-dimensional arrays, each containing 2 integers. Each one-dimensional array occupies 4 bytes (two bytes for each integer). These one-dimensional arrays are placed linearly (zeroth 1-D array followed by first 1-D array, etc.). Hence

```
        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 ;

    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", q[i][j] ) ;
        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}
```


And here is the output…

```
1 2 3 4
5 6 7 8
```

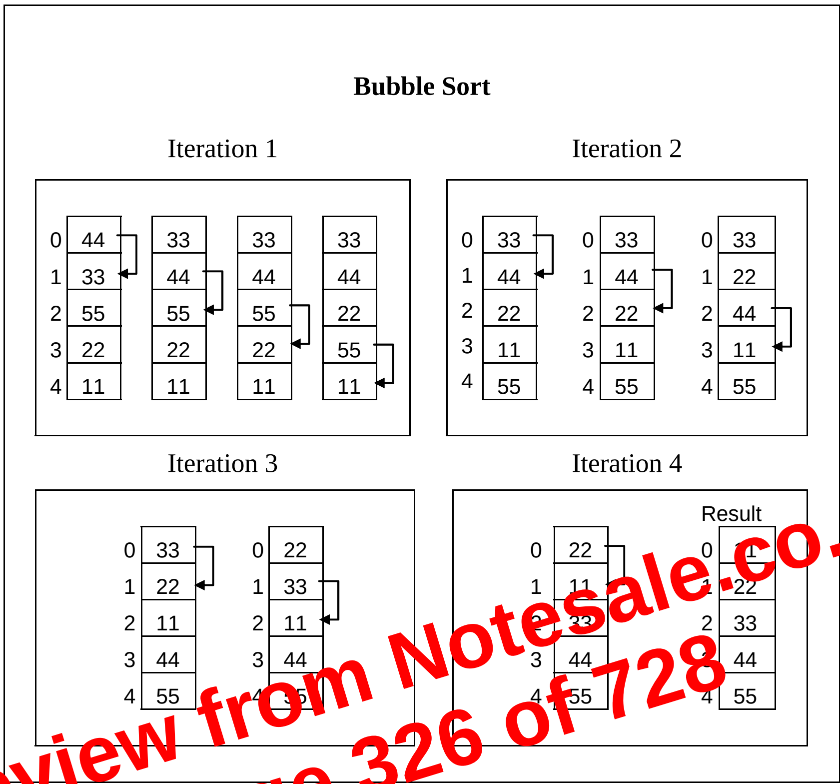

Figure 8.11 (b)

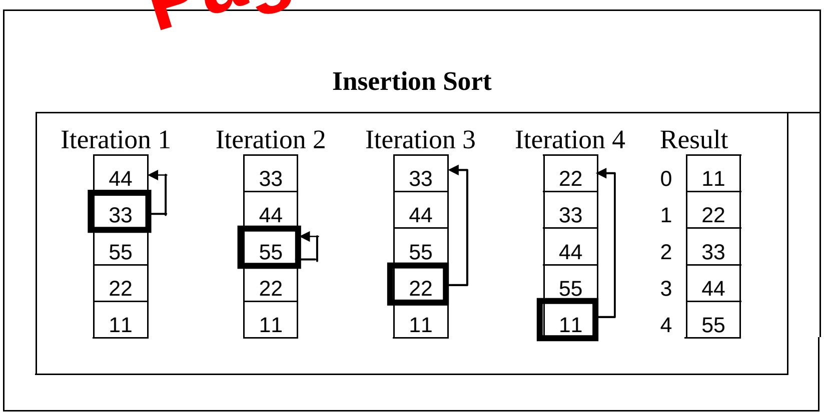

Figure 8.11 (c)

(d) Implement the following procedure to generate prime numbers from 1 to 100 into a program. This procedure is called sieve of Eratosthenes.

step 1	Fill an array **num[100]** with numbers from 1 to 100
step 2	Starting with the second entry in the array, set all its multiples to zero.
step 3	Proceed to the next non-zero element and set all its multiples to zero.
step 4	Repeat step 3 till you have set up the multiples of all the non-zero elements to zero
step 5	At the conclusion of step 4, all the non-zero entries left in the array would be prime numbers, so print out these numbers.

More on arrays, Arrays and pointers

[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 ) ;
```

[G] Answer the following:

(a) What would happen if you try to put so many values into an array when you initialize it that the size of the array is exceeded?

1. nothing
2. possible system malfunction
3. error message from the compiler
4. other data may be overwritten

(b) In an array **int arr[12]** the word **arr** represents the a__________ of the array

(c) What would happen if you put too few elements in an array when you initialize it?

1. nothing
2. possible system malfunction
3. error message from the compiler
4. unused elements will be filled with 0's or garbage

(d) What would happen if you assign a value to an element of an array whose subscript exceeds the size of the array?

1. the element will be set to 0
2. nothing, it's done all the time
3. other data may be overwritten
4. error message from the compiler

(e) When you pass an array as an argument to a function, what actually gets passed?

1. address of the array
2. values of the elements of the array
3. address of the first element of the array
4. number of elements of the array

9 Puppetting On Strings

- What are Strings
- More about Strings
- Pointers and Strings
- Standard Library String Functions
  - *strlen( )*
  - *strcpy*( )
  - *strcat*( )
  - *strcmp*( )
- Two-Dimensional Array of Characters
- Array of Pointers to Strings
- Limitation of Array of Pointers to Strings
  - Solution
- Summary
- Exercise

```
}
```

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 ;
```

While entering the string using **scanf( )** we must be cautious about two things:

(a) The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays. Hence, if you carelessly exceed the bounds there is always a danger of overwriting something important, and in that event, you would have nobody to blame but yourselves.

(b) **scanf( )** is not capable of receiving multi-word strings. Therefore names such as 'Debashish Roy' would be unacceptable. The way to get around this limitation is by using the function **gets( )**. The usage of functions **gets( )** and its counterpart **puts( )** is shown below.

```
main( )
{
    char  name[25] ;

    printf ( "Enter your full name " ) ;
    gets ( name ) ;
    puts ( "Hello!" ) ;
    puts ( name ) ;
}
```


And here is the output...

```
Enter your name Debashish Roy
Hello!
Debashish Roy
```

The program and the output are self-explanatory except for the fact that, **puts( )** can display only one string at a time (hence the use of two **puts( )** in the program above). Also, on displaying a string, unlike **printf( )**, **puts( )** places the cursor on the next line. Though **gets( )** is capable of receiving only

```
xstrlen ( char  *s )
{
    int  length = 0 ;

    while ( *s != '\0' )
    {
        length++ ;
        s++ ;
    }

    return ( length ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

The function **xstrlen( )** is fairly simple. All that it does is keep counting the characters till the end of string is not met. Or in other words keep counting characters till the pointer s doesn't point to '\0'.

strcpy( )

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of **strcpy( )** in action...

```
main( )
{
    char  source[ ] = "Sayonara" ;
```

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

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

(a)
```
main( )
{
    char  c[2] = "A" ;
    printf ( "\n%c", c[0] ) ;
    printf ( "\n%s", c ) ;
}
```

(b)
```
main( )
{
    char  s[ ] = "Get organised! learn C!!" ;
    printf ( "\n%s", &s[2] ) ;
    printf ( "\n%s", s ) ;
    printf ( "\n%s", &s ) ;
    printf ( "\n%c", s[2] ) ;
}
```

(c)
```
main( )
{
    char  s[ ] = "No two viruses work similarly" ;
    int  i = 0 ;
    while ( s[i] != 0 )
    {
        printf ( "\n%c %c", s[i], *( s + i ) ) ;
        printf ( "\n%c %c", i[s], *( i + s ) ) ;
        i++ ;
    }
}
```


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

Hint: Write a function **xstrrev (string)** which should reverse the contents of one string. Call this function for reversing each string stored in **s**.

(d) Develop a program that receives the month and year from the keyboard as integers and prints the calendar in the following format.

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

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure.

A structure contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
main( )
{
    struct book
    {
        char  name ;
        float  price ;
        int  pages ;
    } ;
    struct book  b1, b2, b3 ;

    printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;

    printf ( "\nAnd this is what you entered" ) ;
    printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
    printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
    printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
}
```


And here is the output...

```
Enter names, prices and no. of pages of 3 books
A  100.00  354
C  256.50  682
F  233.70  512
```

```
{
    char  name ;
    float  price ;
    int  pages ;
} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
    char  name[10] ;
    float  price ;
    int  pages ;
} ;
struct book  b1 = { "Basic", 130.00, 550 } ;
struct book  b2 = { "Physics", 150.80, 800 } ;
```

Note the following points while declaring a structure type:

(a) The closing brace in the structure type declaration must be followed by a semicolon.

(b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.

(c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

This provides space in memory for 100 structures of the type **struct book**.

(b) The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s. For example, we refer to zeroth book's price as **b[0].price**. Similarly, we refer first book's pages as **b[1].pages**.

(c) It should be appreciated what careful thought Dennis Ritchie has put into C language. He first defined array as a collection of similar elements; then realized that dissimilar data types that are often found in real life cannot be handled using arrays, therefore created a new data type called structure. But even using structures programming convenience could not be achieved, because a lot of variables (**b1** to **b100** for storing data about hundred books) needed to be handled. Therefore he allowed us to create an array of structures; an array of similar data types which themselves are a collection of dissimilar data types. Hats off to the genius!

(d) In an array of structures all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations you can very well visualise the arrangement of array of structures in memory. In our example, **b[0]**'s **name**, **price** and **pages** in memory would be immediately followed by **b[1]**'s **name**, **price** and **pages**, and so on.

(e) What is the function **linkfloat( )** doing here? If you don't define it you are bound to get the error "Floating Point Formats Not Linked" with majority of C Compilers. What causes this error to occur? When parsing our source file, if the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating-point emulator. A floating point emulator is used to manipulate floating point numbers in runtime library functions like we are passing the base addresses of the arrays **name** and **author**, but the value stored in **callno**. Thus, this is a mixed call—a call by reference as well as a call by value.

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time. This method is shown in the following program.

```
struct book
{
    char  name[25] ;
    char  author[25] ;
    int  callno ;
} ;

main( )
{
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    display ( b1 ) ;
}

display ( struct book b )
{
    printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```


And here is the output...

```
Let us C YPK 101
```

Note that here the calling of function **display( )** becomes quite compact,

```
display ( b1 ) ;
```

operator requires a structure variable on its left. In such cases C provides an operator **->**, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the **'.'** structure operator, there must always be a structure variable, whereas on the left hand side of the **'->'** operator there must always be a pointer to a structure. The arrangement of the structure variable and pointer to structure in memory is shown in the Figure 10.2.

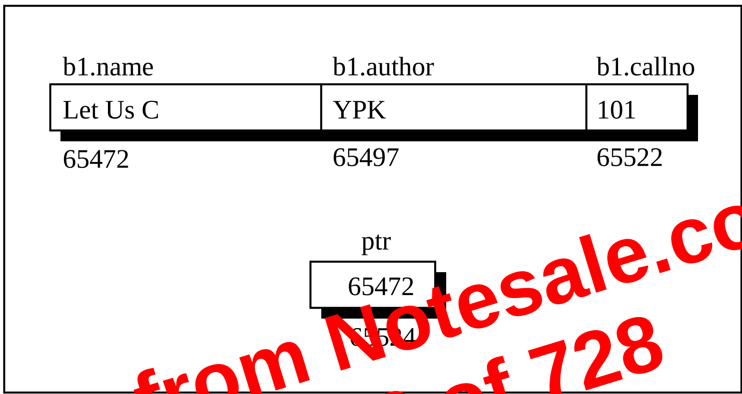

Figure 10.2

Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this.

```
/* Passing address of a structure variable */
struct book
{
    char  name[25] ;
    char  author[25] ;
    int  callno ;
} ;

main( )
{
    struct book  b1 = { "Let us C", "YPK", 101 } ;
    display ( &b1 ) ;
```

```
}

display ( struct book  *b )
{
    printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Again note that to access the structure elements using pointer to a structure we have to use the ‘**->**’ operator.

Also, the structure **struct book** should be declared outside **main( )** such that this data type is available to **display( )** while declaring pointer to the structure.

(e) Consider the following code snippet:

```
struct emp
{
    int a ;
    char ch ;
    float s ;
} ;
struct emp e ;
printf ( "%u %u %u", &e.a, &e.ch, &e.s ) ;
```

If we execute this program using TC/TC++ compiler we get the addresses as:

```
65518  65520  65521
```

As expected, in memory the **char** begins immediately after the **int** and **float** begins immediately after the **char**.

```
            char  language[10] ;
        } ;
        struct employee  e = { "Hacker", "C" } ;
        printf ( "\n%s %d", e.name, e.language ) ;
    }
```

(c)
```
    struct virus
    {
        char  signature[25] ;
        char  status[20] ;
        int  size ;
    } v[2] = {
                    "Yankee Doodle", "Deadly", 1813,
                    "Dark Avenger",  "Killer", 1795
              } ;
    main( )
    {
        int  i ;
        for ( i = 0 ; i <=1 ; i++ )
            printf ( "\n%s %s", v.signature, v.status ) ;
    }
```

(d)
```
    struct s
    {
        char  ch ;
        int  i ;
        float  a ;
    } ;
    main( )
    {
        struct s  var = { 'C', 100, 12.55 } ;
        f ( var ) ;
        g ( &var ) ;
    }
    f ( struct s  v )
    {
        printf ( "\n%c %d %f", v -> ch, v -> i, v -> a ) ;
    }
```

1. Add book information
2. Display book information
3. List all books of given author
4. List the title of specified book
5. List the count of books in the library
6. List the books in the order of accession number
7. Exit

Create a structure called **library** to hold accession number, title of the book, author name, price of the book, and flag indicating whether book is issued or not.

(g) Write a program that compares two given dates. To store date use structure say **date** that contains three members namely date, month and year. If the dates are equal then display message as "Equal" otherwise "Unequal".

(h) Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent location. The individual elements are stored “somewhere” in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. Thus, a linked list is a collection of elements called nodes, each of which stores two item of information—an element of the list, and a link, i.e., a pointer or an address that indicates explicitly the location of the node containing the successor of this list element.

Write a program to build a linked list by adding new nodes at the beginning, at the end or in the middle of the linked list. Also write a function **display( )** which display all the nodes present in the linked list.

(i) A stack is a data structure in which addition of new element or deletion of existing element always takes place at the same

```
    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 array **str**. Since the string **str** is present in memory what is written into **str** using **sprintf( )** doesn't get displayed on the screen. Once **str** has been built, its contents can be displayed on the screen. In our program this was achieved by the second **printf( )** statement.

The counterpart of **sprintf( )** is the **sscanf( )** function. It allows us to read characters from a string and to convert and store them in C variables according to specified formats. The **sscanf( )** function comes in handy for in-memory conversion of characters to values. You may find it convenient to read in strings from a file and then extract values from a string by using **sscanf( )**. The usage of **sscanf( )** is same as **scanf( )**, except that the first argument is the string from which reading is to take place.

Unformatted Console I/O Functions

There are several standard library functions available under this category—those that can deal with a single character and those that can deal with a string of characters. For openers let us look at those which handle one character at a time.

So far for input we have consistently used the **scanf( )** function. However, for some situations the **scanf( )** function has one glaring weakness... you need to hit the Enter key before the function can

Closing the File

When we have finished reading from the file, we need to close it. This is done using the function **fclose( )** through the statement,

```
fclose ( fp ) ;
```

Once we close the file we can no longer read from it using **getc( )** unless we reopen the file. Note that to close the file we don't use the filename but the file pointer **fp**. On closing the file the buffer associated with the file is removed from memory.

In this program we have opened the file for reading. Suppose we open a file with an intention to write characters into it. This time too a buffer would get associated with it. When we attempt to write characters into this file using **fputc( )** the characters would get written to the buffer. When we close this file using **fclose( )** three operations would be performed:

(a) The characters in the buffer would be written to the file on the disk.
(b) At the end of file a character with ASCII value 26 would get written.
(c) The buffer would be eliminated from memory.

You can imagine a possibility when the buffer may become full before we close the file. In such a case the buffer's contents would be written to the disk the moment it becomes full. All this buffer management is done for us by the library functions.

Counting Characters, Tabs, Spaces, …

Having understood the first file I/O program in detail let us now try our hand at one more. Let us write a program that will read a file and count how many characters, spaces, tabs and newlines are present in it. Here is the program…

```
    fp = fopen ( "POEM.TXT", "w" ) ;
    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    printf ( "\nEnter a few lines of text:\n" ) ;
    while ( strlen ( gets ( s ) ) > 0 )
    {
        fputs ( s, fp ) ;
        fputs ( "\n", fp ) ;
    }

    fclose ( fp ) ;
}
```

And here is a sample run of the program...

```
Enter a few lines of text:
Shining and bright, they are forever,
so true about diamonds,
more so of memories,
especially yours !
```


Note that each string is terminated by hitting enter. To terminate the execution of the program, hit enter at the beginning of a line. This creates a string of zero length, which the program recognizes as the signal to close the file and exit.

We have set up a character array to receive the string; the **fputs( )** function then writes the contents of the array to the disk. Since **fputs( )** does not automatically add a newline character to the end of the string, we must do this explicitly to make it easier to read the string back from the file.

Here is a program that reads strings from a disk file.

```
    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 ) ;
}
```


Using this program we can comfortably copy text as well as binary files. Note that here we have opened the source and target files in “rb” and “wb” modes respectively. While opening the file in text mode we can use either “r” or “rt”, but since text mode is the default mode we usually drop the ‘t’.

From the programming angle there are three main areas where text and binary mode files are different. These are:

(a) Handling of newlines
(b) Representation of end of file
(c) Storage of numbers

Let us explore these three differences.

Text versus Binary Mode: Storage of Numbers

The only function that is available for storing numbers in a disk file is the **fprintf( )** function. It is important to understand how numerical data is stored on the disk by **fprintf( ).** Text and characters are stored one character per byte, as we would expect. Are numbers stored as they are in memory, two bytes for an integer, four bytes for a float, and so on? No.

Numbers are stored as strings of characters. Thus, 1234, even though it occupies two bytes in memory, when transferred to the disk using **fprintf( )**, would occupy four bytes, one byte per character. Similarly, the floating-point number 1234.56 would occupy 7 bytes on disk. Thus, numbers with more digits would require more disk space.

Hence if large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions (**fread( )** and **fwrite( )** which are discussed later) which store the numbers in binary format. It means each number would occupy same number of bytes on disk as it occupies in memory.

Record I/O Revisited

The record I/O program that we did in an earlier section has two disadvantages:

(a) The numbers (basic salary) would occupy more number of bytes, since the file has been opened in text mode. This is because when the file is opened in text mode, each number is stored as a character string.

(b) If the number of fields in the structure increase (say, by adding address, house rent allowance etc.), writing structures

```
switch ( choice )
{
    case '1' :

        fseek ( fp, 0 , SEEK_END ) ;
        another = 'Y' ;

        while ( another == 'Y' )
        {
            printf ( "\nEnter name, age and basic sal. " ) ;
            scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
            fwrite ( &e, recsize, 1, fp ) ;
            printf ( "\nAdd another Record (Y/N) " ) ;
            fflush ( stdin ) ;
            another = getche( ) ;
        }

        break ;

    case '2' :

        rewind ( fp ) ;

        while ( fread ( &e, recsize, 1, fp ) == 1 )
            printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

        break ;

    case '3' :

        another = 'Y' ;
        while ( another == 'Y' )
        {
            printf ( "\nEnter name of employee to modify " ) ;
            scanf ( "%s", empname ) ;

            rewind ( fp ) ;
            while ( fread ( &e, recsize, 1, fp ) == 1 )
```

```
        {
            if ( strcmp ( e.name, empname ) == 0 )
            {
                printf ( "\nEnter new name, age & bs" ) ;
                scanf ( "%s %d %f", e.name, &e.age,
                        &e.bs ) ;
                fseek ( fp, - recsize, SEEK_CUR ) ;
                fwrite ( &e, recsize, 1, fp ) ;
                break ;
            }
        }

        printf ( "\nModify another Record (Y/N) " ) ;
        fflush ( stdin ) ;
        another = getche( ) ;
    }

    break ;

case '4' :

    another = 'Y' ;
    while ( another == 'Y' )
    {
        printf ( "\nEnter name of employee to delete " ) ;
        scanf ( "%s", empname ) ;

        ft = fopen ( "TEMP.DAT", "wb" ) ;

        rewind ( fp ) ;
        while ( fread ( &e, recsize, 1, fp ) == 1 )
        {
            if ( strcmp ( e.name, empname ) != 0 )
                fwrite ( &e, recsize, 1, ft ) ;
        }

        fclose ( fp ) ;
        fclose ( ft ) ;
```

```
                    remove ( "EMP.DAT" ) ;
                    rename ( "TEMP.DAT", "EMP.DAT" ) ;

                    fp = fopen ( "EMP.DAT", "rb+" ) ;

                    printf ( "Delete another Record (Y/N) " ) ;
                    fflush ( stdin ) ;
                    another = getche( ) ;
                }
                break ;

            case '0' :
                fclose ( fp ) ;
                exit( ) ;
        }
    }
}
```

To understand how this program works, you need to be familiar with the concept of pointers. A pointer is initiated whenever we open a file. On opening a file a pointer is set up which points to the first record in the file. To be precise this pointer is present in the structure to which the file pointer returned by **fopen( )** points to. On using the functions **fread( )** or **fwrite( )**, the pointer moves to the beginning of the next record. On closing a file the pointer is deactivated. Note that the pointer movement is of utmost importance since **fread( )** always reads that record where the pointer is currently placed. Similarly, **fwrite( )** always writes the record where the pointer is currently placed.

The **rewind( )** function places the pointer to the beginning of the file, irrespective of where it is present right now.

The **fseek( )** function lets us move the pointer from one record to another. In the program above, to move the pointer to the previous record from its current position, we used the function,

O_CREAT	-	Creates a new file for writing (has no effect if file already exists)
O_RDONLY	-	Creates a new file for reading only
O_RDWR	-	Creates a file for both reading and writing
O_WRONLY	-	Creates a file for writing only
O_BINARY	-	Creates a file in binary mode
O_TEXT	-	Creates a file in text mode

These 'O-flags' are defined in the file "fcntl.h". So this file must be included in the program while usng low level disk I/O. Note that the file "stdio.h" is not necessary for low level disk I/O. When two or more O-flags are used together, they are combined using the bitwise OR operator (|). Chapter 14 discusses bitwise operators in detail.

The other statement used in our program to open the file is,

```
outhandle = open ( target, O_CREAT | O_BINARY | O_WRONLY,
                   S_IWRITE ) ;
```


Note that since the target file is not existing when it is being opened we have used the O_CREAT flag, and since we want to write to the file and not read from it, therefore we have used O_WRONLY. And finally, since we want to open the file in binary mode we have used O_BINARY.

Whenever O_CREAT flag is used, another argument must be added to **open( )** function to indicate the read/write status of the file to be created. This argument is called 'permission argument'. Permission arguments could be any of the following:

S_IWRITE	-	Writing to the file permitted
S_IREAD	-	Reading from the file permitted

```
        while ( fscanf ( fp, "%s %d", name, &age ) != NULL )
        fclose ( fp ) ;
    }
```

(g)
```
    main( )
    {
        FILE  *fp ;
        char  names[20] ;
        int  i ;
        fp = fopen ( "students.c", "wb" ) ;
        for ( i = 0 ; i <= 10 ; i++ )
        {
            puts ( "\nEnter name " ) ;
            gets ( name ) ;
            fwrite ( name, size of ( name ), 1, fp ) ;
        }
        close ( fp ) ;
    }
```

(h)
```
    main( )
    {
        FILE  *fp ;
        char  name[20] = "Ajay" ;
        int  i ;
        fp = fopen ( "students.c", "r" ) ;
        for ( i = 0 ; i <= 10 ; i++ )
            fwrite ( name, sizeof ( name ), 1, fp ) ;
        close ( fp ) ;
    }
```

(i)
```
    #include "fcntl.h"
    main( )
    {
        int  fp ;
        fp = open ( "pr22.c" , "r" ) ;
        if ( fp == -1 )
            puts ( "cannot open file" ) ;
        else
            close ( fp ) ;
```

[C] Attempt the following:

(a) Write a program to read a file and display contents with its line numbers.

(b) Write a program to find the size of a text file without traversing it character by character.

(c) Write a program to add the contents of one file at the end of another.

(d) Suppose a file contains student's records with each record containing name and age of a student. Write a program to read these records and display them in sorted order by name.

(e) Write a program to copy one file to another. While doing so replace all lowercase characters to their equivalent uppercase characters.

(f) Write a program that merges lines alternately from two files and writes the results to new file. If one file has less number of lines than the other, the remaining lines from the larger file should be simply copied into the target file.

(g) Write a program to display the contents of a text file on the screen. Make following provisions:

Display the contents inside a box drawn with opposite corner co-ordinates being (0, 1) and (79, 23). Display the name of the file whose contents are being displayed, and the page numbers in the zeroth row. The moment one screenful of file has been displayed, flash a message 'Press any key...' in 24th row. When a key is hit, the next page's contents should be displayed, and so on till the end of file.

(h) Write a program to encrypt/decrypt a file using:

```
    float amount ;
} ;
```

The parameter **trans_type** contains D/W indicating deposit or withdrawal of amount. Write a program to update 'CUSTOMER.DAT' file, i.e. if the **trans_type** is 'D' then update the **balance** of 'CUSTOMER.DAT' by adding **amount** to balance for the corresponding **accno**. Similarly, if **trans_type** is 'W' then subtract the **amount** from **balance**. However, while subtracting the amount make sure that the amount should not get overdrawn, i.e. at least 100 Rs. Should remain in the account.

(j) There are 100 records present in a file with the following structure:

```
struct date
{
    int d, m, y ;
} ;

struct employee
{
    int empcode[6] ;
    char empname[20] ;
    struct date join_date ;
    float salary ;
} ;
```


Write a program to read these records, arrange them in ascending order of **join_date** and write them in to a target file.

(k) A hospital keeps a file of blood donors in which each record has the format:
Name: 20 Columns
Address: 40 Columns

shown in the following sample run. The Ctrl-Z character is often called end of file character.

```
C>UTIL.EXE
perhaps I had a wicked childhood,
perhaps I had a miserable youth,
but somewhere in my wicked miserable past,
there must have been a moment of truth ^Z
C>
```

Now let's see what happens when we invoke this program from in a different way, using redirection:

```
C>UTIL.EXE > POEM.TXT
C>
```

Here we are causing the output to be redirected to the file POEM.TXT. Can we prove that this the output has indeed gone to the file POEM.TXT? Yes, by using the TYPE command as follows:

```
C>TYPE POEM.TXT
perhaps I had a wicked childhood,
perhaps I had a miserable youth,
but somewhere in my wicked miserable past,
there must have been a moment of truth
C>
```

There's the result of our typing sitting in the file. The redirection operator, '>', causes any output intended for the screen to be written to the file whose name follows the operator.

Note that the data to be redirected to a file doesn't need to be typed by a user at the keyboard; the program itself can generate it. Any output normally sent to the screen can be redirected to a disk file. As an example consider the following program for generating the ASCII table on screen:

```
}
```

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 ) ;
```

```
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.

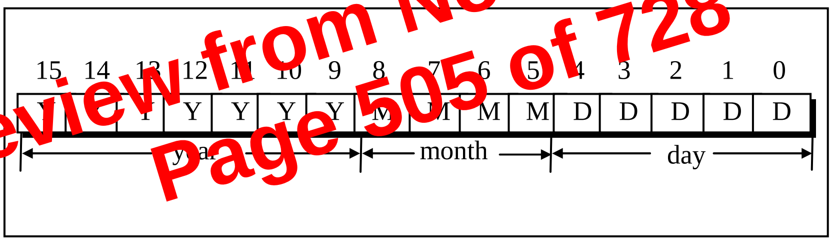

Figure 14.3

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
```

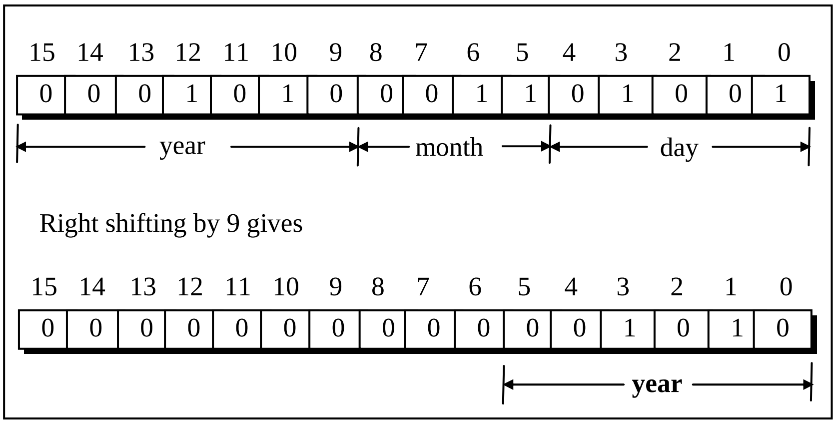

Figure 14.5

On similar lines, left shifting by 7, followed by right shifting by 12 yields month.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
H	H	H	H	H	M	M	M	M	M	M	S	S	S	S	S

Figure 14.13

(f) In order to save disk space information about student is stored in an integer variable. If bit number 0 is on then it indicates Ist year student, bit number 1 to 3 stores IInd year, IIIrd year and IVth year student respectively. The bit number 4 to 7 stores stream Mechanical, Chemical, Electronics and IT. Rest of the bits store room number. Based on the given data, write a program that asks for the room number and displays the information about the student, if its data exists in the array. The contents of array are,

```
int data[ ] = {  273, 548, 786, 1096 } ;
```

(g) What will be the output of the following program:

```
main( )
{
    int  i = 32, j = 65, k, l, m, n, o, p ;
    k = i | 35 ;   l = ~k ;   m = i & j ;
    n = j ^ 32 ;   o = j << 2 ;   p = i >> 5 ;
    printf ( "\nk = %d l = %d m = %d", k, l, m ) ;
    printf ( "\nn = %d o = %d p = %d", n, o, p ) ;
}
```

called single-tasking environment. Since only one program could run at any given time entire resources of the machine like memory and hardware devices were accessible to this program. Under 32-bit environment like Windows several programs reside and work in memory at the same time. Hence it is known as a multi-tasking environment. But the moment there are multiple programs running in memory there is a possibility of conflict if two programs simultaneously access the machine resources. To prevent this, Windows does not permit any application direct access to any machine resource. To channelize the access without resulting into conflict between applications several new mechanisms were created in the Microprocessor & OS. This had a direct bearing on the way the application programs are created. This is not a Windows OS book. So we would restrict our discussion about the new mechanisms that have been introduced in Windows to topics that are related, to C programming. These topics are ‘Memory Management and Device Access’.

Memory Management

Since users have become more demanding, modern day applications have to contend with these demands and provide several features in them. To add to this, under Windows several such applications run in memory simultaneously. The maximum allowable memory—1 MB—that was used in 16-bit environment was just too small for this. Hence Windows had to evolve a new memory management model. Since Windows runs on 32-bit microprocessors each CPU register is 32-bit long. Whenever we store a value at a memory location the address of this memory location has to be stored in the CPU register at some point in time. Thus a 32-bit address can be stored in these registers. This means that we can store 2^{32} unique addresses in the registers at different times. As a result, we can access 4 GB of memory locations using 32-bit registers. As pointers store addresses, every pointer under 32-bit environment also became a 4-byte entity.

However, if we decide to install 4 GB memory it would cost a lot. Hence Windows uses a memory model which makes use of as much of physical memory (say 128 MB) as has been installed and simulates the balance amount of memory (4 GB – 128 MB) on the hard disk. Be aware that this balance memory is simulated as and when the need to do so arises. Thus memory management is demand based.

Note that programs cannot execute straight-away from hard disk. They have to be first brought into physical memory before they can get executed. Suppose there are multiple programs already in memory and a new program starts executing. If this new program needs more memory than what is available right now, then some of the existing programs (or their parts) would be transferred to the disk in order to free the physical memory to accommodate the new program. This operation is often called page-out operation. Here page stands for a block of memory (usually of size 4096 bytes). When that part of the program that was paged out is needed it is brought back into memory (called page-in operation) and some other programs (or their parts) are paged out. This keeps on happening without a common user's knowledge all the time while working with Windows. A few more facts that you must note about paging are as follows:

(a) Part of the program that is currently executing might also be paged out to the disk.

(b) When the program is paged in (from disk to memory) there is no guarantee that it would be brought back to the same physical location where it was before it was paged out.

Now imagine how the paging operations would affect our programming. Suppose we have a pointer pointing to some data present in a page. If this page gets paged out and is later paged in to a different physical location then the pointer would obviously have a wrong address. Hence under Windows the pointer never holds the physical address of any memory location. It always holds a virtual address of that location. What is this virtual address? At offset (from the start of the page) of the physical memory location to be accessed.

Note that the CR3 register is not accessible from an application. Hence an application can never directly reach a physical address. Also, as the paging activity is going on the OS would suitably keep updating the values in the two tables.

Device Access

All devices under Windows are shared amongst all the running programs. Hence no program is permitted a direct access to any of the devices. The access to a device is routed through a device driver program, which finally accesses the device. There is a standard way in which an application can communicate with the device driver. It is device driver's responsibility to ensure that multiple requests coming from different applications are handled without causing any conflict. This standard way of communication is discussed in detail in Chapter 17.

DOS Programming Model

Typical 16-bit environments like DOS use a sequential programming model. In this model programs are executed from top to bottom in an orderly fashion. The path along which the control flows from start to finish may vary during each execution depending on the input that the program receives or the conditions under which it is run. However, the path remains fairly predictable. C programs written in this model begin execution with **main( )** (often called entry point) and then call other functions present in the program. If you assume some input data you can easily walk through the program from beginning to end. In this programming model it is the program and not the operating system that determines which function gets called and when. The operating system simply loads and executes the program and then waits for it to finish. If the program wishes it can take help of the OS to carry

nCmdShow: This is an integer value that is passed to the function. This integer tells the program whether the window that it creates should appear minimized, as an icon, normal, or maximized when it is displayed for the first time.

– The **MessageBox( )** function pops up a message box whose title is ‘Title’ and which contains a message ‘Hello!’.

– Returning **0** from **WinMain( )** indicates success, whereas, returning a nonzero value indicates failure.

– Instead of printing ‘Hello!’ in the message box we can print the command line arguments that the user may supply while executing the program. The command line arguments can be supplied to the program by executing it from Start | Run as shown in Figure 16.7.

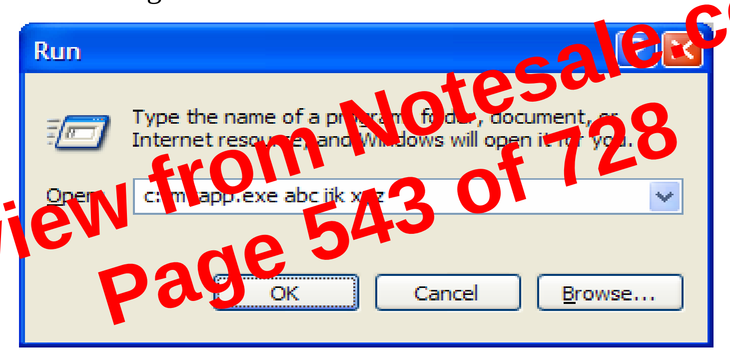

Figure 16.7

Note from Figure 16.7 that ‘myapp.exe’ is the name of our application, whereas, ‘abc ijk xyz’ represents command line arguments. The parameter **lpszCmdline** points to the string “abc ijk xyz”. This string can be printed using the following statement:

```
MessageBox ( 0, lpszCmdline, "Title", 0 ) ;
```

If the entire command line including the filename is to be retrieved we can use the **GetCommandLine( )** function.

(f) Windows does not permit direct access to memory or hardware devices.
(g) Windows uses a Demand-based Virtual Memory Model to manage memory.
(h) Under Windows there is two-way communication between the program and the OS.
(i) Windows maintains a system message queue common for all applications.
(j) Windows maintains an application message queue per running application.
(k) Calling convention decides the order in which the parameters are passed to a function and whether the calling function or the called function clears the stack.
(l) Commonly used calling conventions are **__cdecl** and **__stdcall**.
(m) Hungarian notation though good its usage is not recommended any more.

Exercise

[A] State True or False:

(a) MS-DOS uses a procedural programming model.
(b) A Windows program can directly call a device driver program for a device.
(c) API functions under Windows do not have names.
(d) DOS functions are called using an interrupt mechanism.
(e) Windows uses a 4 GB virtual memory space.
(f) Size of a pointer under Windows depends upon whether it is **near** or **far**.
(g) Under Windows the address stored in a pointer is a virtual address and not a physical address.
(h) One of the parameters of **WinMain( )** called **hPrevInstance** is no longer relevant.

```
int _stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdLine, int nCmdShow )
{
    HWND h ;

    h = CreateWindow ( "BUTTON", "Hit Me", WS_OVERLAPPEDWINDOW,
                        10, 10, 150, 100, 0, 0, i, 0 ) ;
    ShowWindow ( h, nCmdShow ) ;
    MessageBox ( 0, "Hi!", "Waiting", MB_OK ) ;
    return 0 ;
}
```

Here is the output of the program…

Figure 17.2

Let us now understand the program. Every window enjoys certain properties—background color, shape of cursor, shape of icon, etc. All these properties taken together are known as 'window class'. The meaning of 'class' here is 'type'. Windows insists that a window class should be registered with it before we attempt to create windows of that type. Once a window class is registered we can create several windows of that type. Each of these windows would enjoy the same properties that have been registered through the window class. There are several predefined window classes. Some of these are BUTTON, EDIT, LISTBOX, etc. Our program has created one such window using the predefined BUTTON class.

```
            h[x] = CreateWindow ( "BUTTON", "Press Me",
                                  WS_OVERLAPPEDWINDOW, x * 20,
                                  x * 20, 150, 100, 0, 0, i, 0 ) ;
            ShowWindow ( h[x], l ) ;
      }

      MessageBox ( 0, "Hi!", "Waiting", 0 ) ;
      return 0 ;
}
```

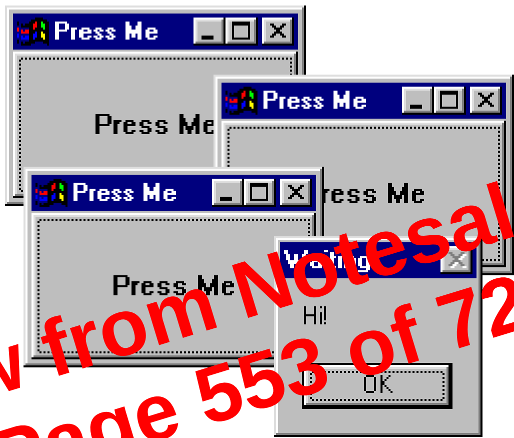

Figure 17.3

Note that each window created in this program is assigned a different handle. You may experiment a bit by changing the name of the window class to EDIT and see the result.

A Real-World Window

Suppose we wish to create a window and draw a few shapes in it. For creating such a window there is no standard window class available. Hence we would have to create our own window class, register it with Windows OS and then create a window on the basis of it. Instead of straightway jumping to a program that draws shapes in a window let us first write a program that creates a window using our window class and lets us interact with it. Here is the program…

```
#include <windows.h>
#include "helper.h"

void OnDestroy ( HWND ) ;

int __stdcall WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdline, int nCmdShow )
{
    MSG m ;

    /* perform application initialization */
    InitInstance  ( hInstance, nCmdShow, "title" ) ;

    /* message loop */
    while ( GetMessage ( &m, 0, 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 ;
        default :
            return DefWindowProc ( hWnd, message, wParam, lParam ) ;
    }
    return 0 ;
}
```

```
void OnDestroy ( HWND hWnd )
{
     PostQuitMessage ( 0 ) ;
}
```

On execution of this program the window shown in Figure 17.4 appears on the screen. We can use minimize and the maximize button it its title bar to minimize and maximize the window. We can stretch its size by dragging its boundaries. Finally, we can close the window by clicking on the close window button in the title bar.

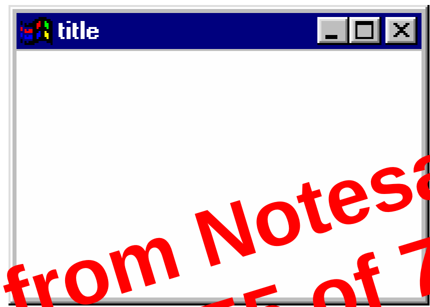

Figure 17.4

Let us now try to understand this program step by step.

Creation and Displaying of Window

Creating and displaying a window on the screen is a 4-step process. These steps are:

(a) Creation of a window class.
(b) Registering the window class with the OS.
(c) Creation of a window based on the registered class.
(d) Displaying the window on the screen.

Creation of a window class involves setting up of elements of a structure called **WNDCLASSEX**. This structure contains several mouse cursor and the status of mouse buttons. Since it is difficult to memorize the message ids they have been suitably **#define**d in 'windows.h'. The message id and the additional information are stored in a structure called MSG.

In **WinMain( )** this MSG structure is retrieved from the message queue by calling the API function **GetMessage( )**. The first parameter passed to this function is the address of the **MSG** structure variable. **GetMessage( )** would pick the message info from the message queue and place it in the structure variable passed to it. Don't bother about the other parameters right now.

After picking up the message from the message queue we need to process it. This is done by calling the **DispatchMessage( )** API function. This function does several activities. These are as follows:

(a) From the **MSG** structure that we pass to it, **DisplayMessage( )** extracts the handle of the window for which this message is meant for.
(b) From the handle it figures out the window class based on which the window has been created.
(c) From the window class structure it obtains the address of a function called **WndProc( )** (short for window procedure). Well I didn't tell you earlier that in **InitInstance( )** while filling the **WNDCLASSEX** structure one of the elements has been set up with the address of a user-defined function called **WndProc( )**.
(d) Using this address it calls the function **WndProc( )**.

Since several messages get posted into the message queue picking of the message and processing it should be done repeatedly. Hence calls to **GetMesage( )** and **DispatchMessage( )** have been made in a **while** loop in **WinMain( )**. When **GetMesage( )** encounters a message with id **WM_QUIT** it returns a **0**. Now the control comes out of the loop and **WinMain( )** comes to an end.

Program Instances

Windows allows you to run more than one copy of a program at a time. This is handy for cutting and pasting between two copies of Notepad or when running more than one terminal session with a terminal emulator program. Each running copy of a program is called a 'program instance'.

Windows performs an interesting memory optimization trick. It shares a single copy of the program's code between all running instances. For example, if you get three instances of Notepad running, there will only be one copy of Notepad's code in memory. All three instances share the same code, but will have separate memory areas to hold the text data being edited. The difference between handling of the code and the data is logical, as each instance of Notepad might edit a different file, so the data must be unique to each instance. The program logic to edit the files is the same for every instance, so there is no reason why a single copy of Notepad's code cannot be shared.

Summary

(a) A message box can be displayed by calling the **MessageBox( )** API function.
(b) Message boxes are often used to ascertain the flow of a program.
(c) Appearance of a message box can be customized.
(d) The **CreateWindow( )** API function creates the window in memory.
(e) The window that is created in memory is displayed using the **ShowWindow( )** API **f**unction.
(f) A 'window class' specifies various properties of the window that we are creating.
(g) The header file 'Windows.h' contains declaration of several macros used in Windows programming.

WndProc( ) function and the message handlers that perform this task are given below

```
int x1, y1, x2, y2 ;

LRESULT CALLBACK WndProc ( HWND hWnd, UINT message,
                                   WPARAM wParam, LPARAM lParam )
{
    switch ( message )
    {
        case WM_DESTROY :
            OnDestroy ( hWnd ) ;
            break ;

        case WM_LBUTTONDOWN :
            OnLButtonDown ( hWnd, LOWORD ( lParam ),
                                   HIWORD ( lParam ) ) ;
            break ;

        case WM_LBUTTONUP :
            OnLButtonUp ( ) ;
            break ;

        case WM_MOUSEMOVE :
            OnMouseMove ( hWnd, wParam, LOWORD ( lParam ),
                                   HIWORD ( lParam ) ) ;
            break ;

        default:
            return DefWindowProc ( hWnd, message, wParam, lParam ) ;
  }
  return 0 ;
}

void OnLButtonDown ( HWND hWnd, int x, int y )
{
    SetCapture ( hWnd ) ;
    x1 = x ;
```

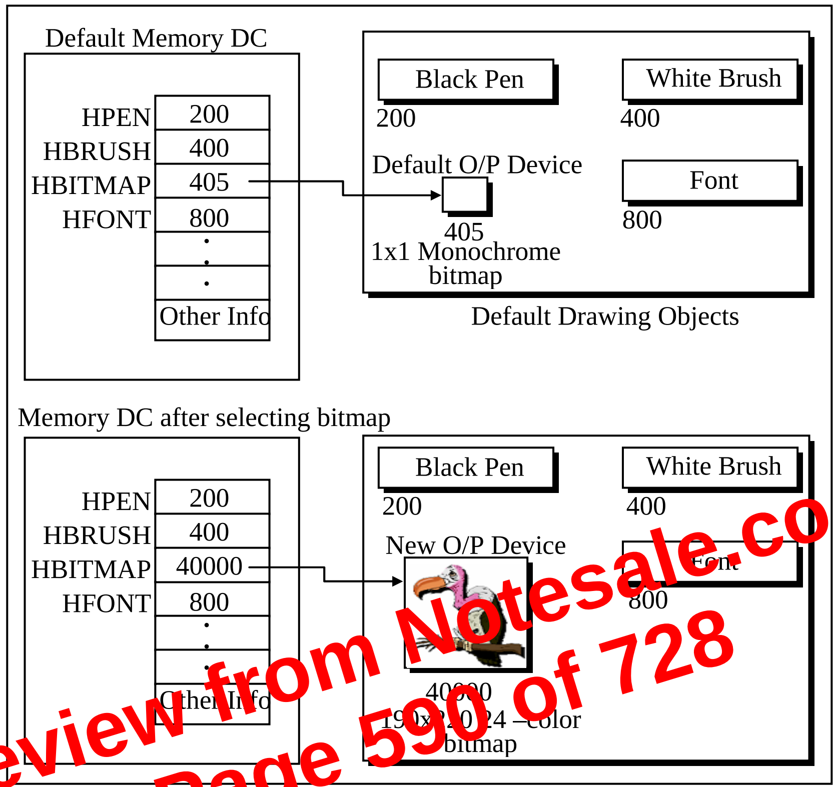

Figure 18.7

What purpose would just increasing the bitmap size/color would serve? Whatever we draw here would get drawn on the bitmap but would still not be visible. We can make it visible by simply copying the bitmap image (including what has been drawn on it) to the screen DC by using the API function **BitBlt( )**.

Before transferring the image to the screen DC we need to make the memory DC compatible with the screen DC. Here making compatible means making certain adjustments in the contents of the memory DC structure. Looking at these values the screen device driver would suitably adjust the colors when the pixels in

```
    hmemdc = CreateCompatibleDC ( hdc ) ;
    holdbmp = SelectObject ( hmemdc, hbmp ) ;

    ReleaseDC ( hWnd, hdc ) ;

    srand ( time ( NULL ) ) ;

    GetClientRect ( hWnd, &r ) ;

    x = rand( ) % r.right - 22 ;
    y = rand( ) % r.bottom - 22 ;

    SetTimer ( hWnd, 1, 50, NULL ) ;
}

void OnDestroy ( HWND hWnd )
{
    KillTimer ( hWnd, 1 ) ;
    SelectObject ( hmemdc, holdbmp ) ;
    DeleteDC ( hmemdc ) ;
    DeleteObject ( hbmp ) ;
    PostQuitMessage ( 0 ) ;
}

void OnTimer ( HWND hWnd )
{
    HDC hdc ;
    RECT r ;
    const int wd = 22, ht = 22 ;
    static int dx = 10, dy = 10 ;

    hdc = GetDC ( hWnd ) ;
    BitBlt ( hdc, x, y, wd, ht, hmemdc, 0, 0, WHITENESS ) ;
    GetClientRect ( hWnd, &r ) ;

    x += dx ;
    if ( x < 0 )
    {
```

(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.
(c) The API function EndPaint( ) is used to release the DC.
(d) The default pen in the DC is a solid pen of white color.
(e) The pen thickness for the pen style other than PS_SOLID has to be 1 pixel.
(f) BeginPaint( ) and GetDC( ) can be used interchangeably.
(g) If we drag the mouse from (10, 10) to (110, 100), 100 WM_MOUSEMOVE messages would be posted into the message queue.
(h) WM_PAINT message is raised when the window contents are scrolled.
(i) With each DC a default monochrome bitmap of size 1 pixel x 1 pixel is associated.
(j) The WM_CREATE message arrives whenever a window is displayed.

[B] Answer the following:

(a) What is meant by Device Independent Drawing and how it is achieved?

Windows does not permit an application program to directly access any of the devices. Instead it provides several API functions to carry out the interaction. These functions have names so calling them is much easier than calling DOS/BIOS functions. When we call an API function to interact with a device, it in turn accesses the device driver program for the device. It is the device driver program that finally accesses the device. There is a standard way in which an application can communicate with the device driver. It is device driver's responsibility to ensure that multiple requests coming from different applications are handled without causing any conflict. In the sections to follow we would see how to communicate with the device driver to be able to interact with the hardware.

One last question—won't the API change if a new device comes into existence? No it won't. That is the beauty of the Windows architecture. All that would change is the device driver program for the new device. The API functions that we would need to interact with this new device driver would remain same. This is shown in Figure 19.4.

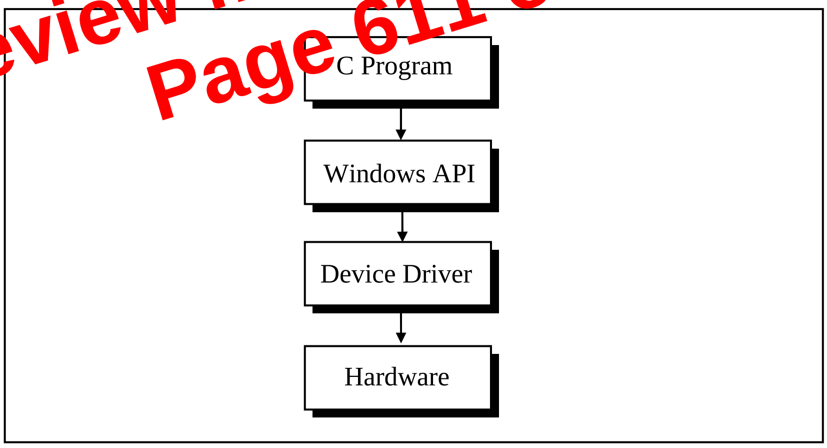

Figure 19.4

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) If we are to read from the second floppy drive we should replace **A:** with **B:** while calling **ReadSector( )**.

(b) To read from storage devices like hard disk drive or CD-ROM or ZIP drive, etc. use the string with appropriate drive letter. The string can be in the range **.\C:** to **.\Z:**.

opposite to their order of installation. This means the last hook procedure installed is the first one to get called.

If the **nCode** parameter contains a value **HC_ACTION** it means that the message that was just removed form the system message queue was a keyboard message. If it is so, then we have checked the previous state of the key before the message was sent. If the state of the key was ‘depressed’ (30th bit of **lParam** is 1) then we have obtained the state of the CapsLock key by calling the **GetKeyState( )** API function. If it is off (0th bit of **state** variable is 0) then we have turned on the CapsLock by simulating a keypress. For this simulation we have called the function **keybd_event( )** twice—first call is for pressing the CapsLock and second is for releasing it. Note that **keybd_event( )** creates a keyboard message from the parameters that we pass to it and posts it into the system message queue. The parameter VK_CAPITAL represents the code for the CapsLock key.

A word of caution! When we use **keybd_event( )** to post keyboard message for a simulated CapsLock keypress, once again our hook procedure would be called when these messages are retrieved from the system message queue. But this time the CapsLock would be on. So we would end up passing control to the next hook procedure through a call to **CallNextHookEx( )**.

When we close the application window as usual the **OnDestroy( )** would be called. In this handler we have obtained the address of the **removehook( )** exported function and called it. In the **removehook( )** function we have unregistered our hook procedure by calling the **UnhookWindowsHookEx( )** API function. Note that to this function we have passed the handle to our hook. As a result our hook procedure is now removed from the hook chain. Hereafter the CapsLock would behave normally. Having unhooked our hook procedure the control would return to **OnDestroy( )** handler where we have promptly unload the DLL from memory by calling the **FreeLibrary( )** API function.

but may contain different application programs, libraries, frameworks, installation scripts, utilities, etc. Which one is better than the other is only a matter of taste.

Linux was first developed for x86-based PCs (386 or higher). These days it also runs on Compaq Alpha AXP, Sun SPARC, Motorola 68000 machines (like Atari ST and Amiga), MIPS, PowerPC, ARM, Intel Itanium, SuperH, etc. Thus Linux works on literally every conceivable microprocessor architecture.

Under Linux one is faced with simply too many choices of Linux distributions, graphical shells and managers, editors, compilers, linkers, debuggers, etc. For simplicity (in my opinion) I have chosen the following combination:

Linux Distribution - Red Hat Linux 9.0
Console Shell - BASH
Graphical Shell - KDE 3.1-10
Editor - KWrite
Compiler - GNU C and C++ compiler (gcc)

We would be using and discussing these in the sections to follow.

C Programming Under Linux

How is C under Linux any different than C under DOS or C under Windows? Well, it is same as well as different. It is same to the extent of using language elements like data types, control instructions and the overall syntax. The usage of standard library functions is also same even though the implementation of each 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.

```
    }
    else
    {
        printf ( "In parent process\n" ) ;
        /* code to copy file */
    }
}
```

As we know, **fork( )** creates a child process and duplicates the code of the parent process in the child process. There onwards the execution of the **fork( )** function continues in both the processes. Thus the duplication code inside **fork( )** is executed once, whereas the remaining code inside it is executed in both the parent as well as the child process. Hence control would come back from **fork( )** twice, even though it is actually called only once. When control returns from **fork( )** of the parent process it returns the PID of the child process, whereas when control returns from **fork( )** of the child process it always returns a 0. This can be exploited by our program to segregate the code that we want to execute in the parent process from the code that we want to execute in the child process. We have done this in our program using an **if** statement. In the parent process the 'else block' would get executed, whereas in the child process the 'if block' would get executed.

Let us now write one more program. This program would use the **fork( )** call to create a child process. In the child process we would print the PID of child and its parent, whereas in the parent process we would print the PID of the parent and its child. Here is the program…

```
# include <sys/types.h>
int main( )
{
    int pid ;
    pid = fork( ) ;

    if ( pid == 0 )
```

After forking a child process we have called the **execl( )** function. This function accepts variable number of arguments. The first parameter to **execl( )** is the absolute path of the program to be executed. The remaining parameters describe the command line arguments for the program to be executed. The last parameter is an end of argument marker which must always be **NULL**. Thus in our case the we have called upon the **execl( )** function to execute the **ls** program as shown below

```
ls -al /etc
```

As a result, all the contents of the **/etc** directory are listed on the screen. Note that the **printf( )** below the call to **execl( )** function is not executed. This is because the **exec** family functions overwrite the image of the calling process with the code and data of the program that is to be executed. In our case the child process' memory was overwritten by the code and data of the **ls** program. Hence the call to **printf( )** did not materialize.

It would make little sense in calling **execl( )** before **fork( )**. This is because a child would not get created and **execl( )** would simply overwrite the main process itself. As a result, no statement beyond the call to **execl( )** would ever get executed. Hence **fork( )** and **execl( )** usually go hand in hand.

Zombies and Orphans

We know that the **ps –A** command lists all the running processes. But from where does the **ps** program get this information? Well, Linux maintains a table containing information about all the processes. This table is called 'Process Table'. Apart from other information the process table contains an entry of 'exit code' of the process. This integer value indicates the reason why the process was terminated. Even though the process comes to an end its entry would remain in the process table until such time that the parent of the terminated process queries the exit code. This act of querying

Communication is the essence of all progress. This is true in real life as well as in programming. In today's world a program that runs in isolation is of little use. A worthwhile program has to communicate with the outside world in general and with the OS in particular. In Chapters 16 and 17 we saw how a Windows based program communicates with Windows. In this chapter let us explore how this communication happens under Linux.

Communication using Signals

In the last chapter we used **fork( )** and **exec( )** library function to create a child process and to execute a new program respectively. These library functions got the job done by communication with the Linux OS. Thus the direction of communication was from the program to the OS. The reverse communication—from the OS to the program—is achieved using a mechanism called 'Signal'. Let us now write a simple program that would help you experience the signal mechanism.


```
int main( )
{
    while ( 1 )
        printf ( "Pogram Running\n" ) ;
    return 0 ;
}
```

The program is fairly straightforward. All that we have done here is we have used an infinite **while** loop to print the message "Program Running" on the screen. When the program is running we can terminate it by pressing the Ctrl + C. When we press Ctrl + C the keyboard device driver informs the Linux kernel about pressing of this special key combination. The kernel reacts to this by sending a signal to our program. Since we have done nothing to handle this signal the default signal handler gets called. In this

```
                printf ( "SIGINT Received\n" ) ;
                break ;

        case SIGTERM :
                printf ( "SIGTERM Received\n" ) ;
                break ;

        case SIGCONT :
                printf ( "SIGCONT Received\n" ) ;
                break ;
        }
}

int main( )
{
        signal ( SIGINT,  sighandler ) ;
        signal ( SIGTERM, sighandler ) ;
        signal ( SIGCONT, sighandler ) ;

        while ( 1 )
            printf ( "\rProgram running" ) ;

        return 0 ;
}
```


In this program during each call to the **signal( )** function we have specified the address of a common signal handler named **sighandler( )**. Thus the same signal handler function would get called when one of the three signals are received. This does not lead to a problem since the **sighandler( )** we can figure out inside the signal ID using the first parameter of the function. In our program we have made use of the **switch-case** construct to print a different message for each of the three signals.

Note that we can easily afford to mix the two methods of registering signals in a program. That is, we can register separate signal handlers for some of the signals and a common handler for

```
            case SIGCONT :
                  printf ( "SIGCONT Received\n" ) ;
                  break ;
      }
}

int main( )
{
      char buffer [ 80 ] = "\0" ;
      sigset_t block ;

      signal ( SIGTERM, sighandler ) ;
      signal ( SIGINT,  sighandler ) ;
      signal ( SIGCONT, sighandler ) ;

      sigemptyset ( &block ) ;
      sigaddset ( &block, SIGTERM ) ;
      sigaddset ( &block, SIGINT ) ;

      sigprocmask ( SIG_BLOCK, &block, NULL ) ;

      while ( strcmp ( buffer, "n" ) != 0 )
      {
            printf ( "Enter a String: " ) ;
            gets ( buffer ) ;
            puts ( buffer ) ;
      }

      sigprocmask ( SIG_UNBLOCK, &block, NULL ) ;

      while ( 1 )
            printf ( "\rProgram Running" ) ;

      return 0 ;
}
```


In this program we have registered a common handler for the **SIGINT**, **SIGTERM** and **SIGCONT** signals. Next we want to

We need to compile this program as follows:

```
gcc  mywindow.c  `pkg-config gtk+-2.0 - -cflags - -libs`
```

Here we are compiling the program 'mywindow.c' and then linking it with the necessary libraries from GTK toolkit. Note the quotes that we have used in the command.

Here is the output of the program…

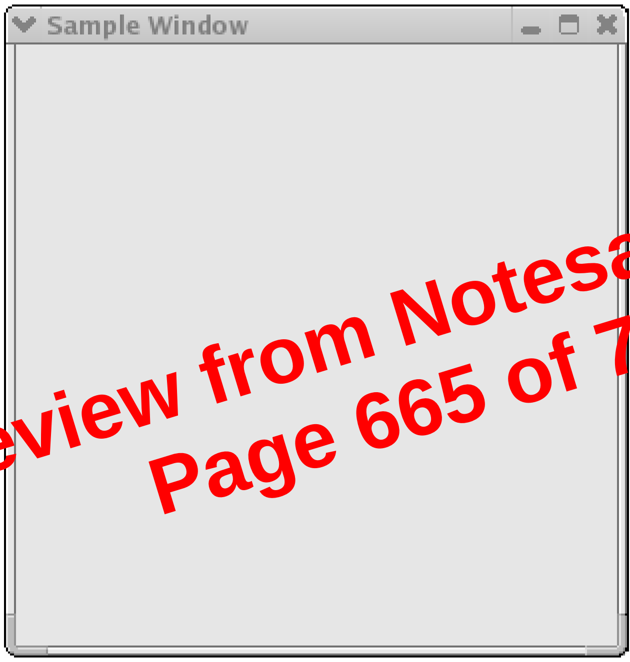

Figure 21.1

The GTK library provides a large number of functions that makes it very easy for us to create GUI programs. Every window under GTK is known as a widget. To create a simple window we have to carry out the following steps:

(g) A process can block a signal or a set of signals using the **sigprocmask( )** function.
(h) Blocked signals are delivered to the process when the signals are unblocked.
(i) A **SIGSTOP** signal is generated when we press Ctrl + Z.
(j) A **SIGSTOP** signal is un-catchable signal.
(k) A suspended process can be resumed using the **fg** command.
(l) A process receives the **SIGCONT** signal when it resumes execution.
(m) In GTK, the **g_signal_connect( )** function can be used to connect a function with an event.

Exercise

[A] State True or False:

(a) All signals registered signals must have a separate signal handler.
(b) Blocked signals are ignored by a process.
(c) Only one signal can be blocked at a time.
(d) Blocked signals are ignored once the signals are unblocked.
(e) If our signal handler gets called the default signal handler automatically gets called.
(f) **gtk_main( )** function makes uses of a loop to prevent the termination of the program.
(g) Multiple signals can be registered at a time using a single call to **signal( )** function.
(h) The **sigprocmask( )** function can block as well as unblock signals.

[B] Answer the following:

(a) How does the Linux OS know if we have registered a signal or not?

(b) What happens when we register a handler for a signal?

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

File Handling Functions

Function	Use
remove	Deletes file
rename	Renames file
unlink	Deletes file

Directory Control Functions

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

Buffer Manipulation Functions

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

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

Graphics Functions

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

Time Related Functions

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

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

Miscellaneous Functions

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

DOS Interface Functions

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

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

a pointer to the character string “a” is assigned to **ch**.

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

```
char  ch ;
```

whereas in the second case it would be,

```
char  *ch ;
```

[13] Forgetting the bounds of an array.

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

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


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

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

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

C Creating Libraries

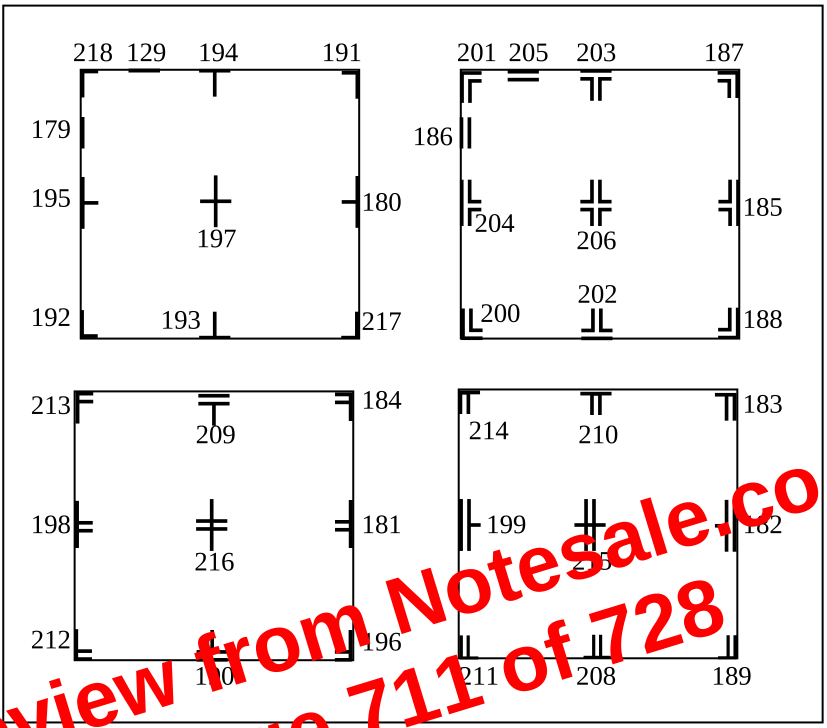
218
129
194
191
179
195
197
180
192
193
217
201
205
203
187
186
204
206
185
202
200
188
213
209
184
198
216
181
212
196
214
210
183
199
182
211
208
189

Figure E.2

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

HINSTANCE hInst ; // current instance

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

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

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

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

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

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

Figure G.1

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