```
    struct point p1, p2;
    printf("This program determines the distance\n");
    printf("between two points in the X-Y plane.\n");
    printf("Enter X and Y co-ordinates of first point: ");
    scanf("%f %f", &p1.x, &p1.y);
    printf("Enter X and Y co-ordinates of second point: ");
    scanf("%f %f", &p2.x, &p2.y);
    printf("Distance between (%f, %f) and (%f, %f) is: %f\n",
p1.x, p1.y, p2.x, p2.y, distance(&p1, &p2));
    return (0);
}
float distance(struct point *p1, struct point *p2)
{
    return (sqrt((p1->x - p2->x) * (p1->x - p2->x) +
(p1->y - p2->y) * (p1->y - p2->y)));
}
```

Here are some exercises for you.

E8) Write a C language program to print a file backwards.

E9) Write a C language program to maintain a personal telephone directory. Your program should support the following functions: 1) add a new number to the directory; 2) locate and display a number, given the owner's name; 3) change a number; 4) delete a number; and 5) exit from the program gracefully.

In the next section, we will discuss another variable type called **union** which can hold variables of different types.

10.7 UNIONS

A **union** is a C variable typed by the keyword **union** that can accommodate at any time just one of its listed members, which may be objects of different types, i.e. of different widths:

```
union hold_all
{
  char alpha;
  int beta;
  float gamma;
  double delta  [3];
  int * ptr;
}
```

This instances `box_1, box_2` and `box_3` of the **union** `hold_all` can hold variables of each of its listed types. Obviously they will be wide enough to hold the biggest of its members, in this case **double** `delta [3]`. Each member of a **union** is stored at the same base address. Similarly to structures, the dot and arrow operators are used to access any of the **union**'s members, via the mechanisms:

`union_name.member_name;`

as in:

```
box_1.alpha = 'x';
box_2.beta = 3;
box_3.delta [0] = 3.14;
```

or `union_pointer->member`, as in:

`x->gamma = 2.71`

where x is a pointer to a **union** variable, say box_1.

A **union** may be initialised by a value of the type of its first member. If a **union** has been assigned a value of a certain type, then it is not correct to attempt to retrieve from it a value of a different type. Program 10.16 and its output illustrate some properties of **union**s. x is a pointer of type **union** hold_all, and is a parameter of the function func() whose two other parameters are an **int** and a **double**. The **union** hold_all is just wide enough to accommodate the three **double**s of the array delta []. The unions box_1, box_2 and box_3 are assigned a **char**, and **int** and a **double** value respectively. func() is invoked with the following arguments: a pointer to box_1, and the current values of box_2.beta and box_3.delta [0]. Within func() itself the expression P->alpha reference the member alpha of box_1, which has the value 'x'. This is changed to 'y'.

```
/* Program 10.16 */
#include <stdio.h>
union hold_all
    {
        char alpha;
        int beta;
        float gamma;
        double delta [3];
        int * ptr;
    }   box_1, box_2, box_3, * x;
void func (union hold_all *p, int q, double r);
int main (void)
{
    printf("The size of the union hold_all is: %d\n",
 sizeof (union hold_all));
    printf("The size of an instance of it, box_1 is: %d\n",
 sizeof box_1);
    box_1.alpha = 'x';
    box_2.beta = 3;
    box_3.delta [0] = 3.14;
    printf("Can func () change box_1.alpha, \
which noe is: %c?\n", box_1.alpha);
    printf("Let\'s see...sending \
control of func ()...\n");
    func(&box_1, box_2.beta, box_3.delta[0]);
    printf("Yes it can: box_1.alpha after func()\
 is: %c\n", box_1.alpha);
    return (0);
}
void func (union hold_all *P, int Q, double R)
{
    P->alpha += R/Q;
}
```

```
/* Program 10.16:  Output */
The size of the union hold_all is:  24
the size of an instance of it, box_1 is:  24
Can func () change box_1.alpha, which now is:  x?
Let's see...  sending control to func()...
Yes it can:  box_1.alpha after func () is:  y
```

In the next section, we will discuss how to manipulate the individual bits of values integer variables.

10.8 BIT FIELDS: THE BITWISE OPERATORS

No doubt it has occurred to you that Booleans should be representable by individual bits in memory, rather than by entire bytes or words. When several flags (i.e. Boolean values) must be stored, the savings made possible by using bits within a word rather than whole words can be considerable. This idea directly motivates the concept of packed fields of bits, and operations on individual bits.

In C there are two ways in which the individual bits within a word can be manipulated. The first depends upon storing a value as an **int** and then to use the bitwise operators, explained below, to mask or set specific bits of the word. The second uses the bit field approach, in which the syntax of definition and the access method are based on structures.

C has six bitwise operators, of which five are binary and one unary. They may be applied to any `singed` or **unsigned** integer operands, **char**, **short**, **int** or **long**. The unary operator ~, a tilde sign, one's complements its operand, i.e. it changes the 1-bits to 0s, and vice versa. It associates from right to left. Suppose the **int** `x` stores the value-1 on a two's complement machine. On such a machine this **int** value is represented by all bits set to 1s. Then ~x will have all its bits set to 0s. Of the binary bitwise operators, three are logical connectives: in order of decreasing priority they are:

& the bitwise AND operator

| the bitwise OR operator, and
^ the bitwise EXCLUSIVE OR operator
The bitwise AND operator produces a result every bit of which is the result of ANDing the corresponding bits of its two operands. The bitwise OR operator produces the result of ORing the bits of its operands. Similarly the bitwise exclusive OR yields the value resulting from XORing its operands. (The difference between the OR and the exclusive OR is that in the latter case if both operands are `true`, the result is `false`. For example, if I'm going to Jaipur from Delhi by bus or by train, then it's false that I'm simultaneously travelling by both forms of transport: I can be either on the bus, or on the train, but not on both.)

The two shift operators << and >> push their left operand to the left or to the right, respectively, a number of bit positions given by their right operand.

To illustrate how these operators are used, suppose a 16-bit **int** `x` has the octal value 012345, i.e. 0001010011100101 binary, and another 16-bit **int** `y` has the octal value 023456, i.e. 0010011100101110 binary, then `x & y` is produced by ANDing the corresponding bits of `x` and `y`, as follows:

x	0001010011100101 octal 12345
y	0010011100101110 octal 23456
x & y	0000010000100100, i.e. octal 2044

Similarly, ORing the bits of `x` and `y` yields:

x	0001010011100101
y	0010011100101110
x | y	0011011111101111, i.e. octal 33757

The exclusive OR of the bits of `x` and `y` yields:

x	0001010011100101
y	0010011100101110
x^y	0011001111001011, i.e. octal 31713

x << 2 pushes out the 2 leftmost bits of x, and replaces the vacated bits on the right by zeroes; this is equivalent to multiplying x by 4. Similarly y >> 2 pushes out to the right the 2 rightmost bits of y. If y is an **unsigned** quantity, as it is in the present instance, the vacated bits will be stuffed with 0s; if y is a signed value, then an arithmetic shift will replace the vacated bits by 1s, but a logical shift will replace them by 0s. Your compiler's manual should tell you which of the two shifts is implemented on your system; if not, a simple program can help you find out. Program 10.17 illustrates the bit fiddling operators:

```
/* Program 10.17; file name: unit10-prog17.c */
#include <stdio.h>
int main (void)
{
    int w = -1;
    int x = 012345, y = 023456, z;
    printf("w = %d, one's complement = ~w = %d\n",
 w, ~w);
    printf("x =\t\t%o\ny =\t\t%o\nz = x & y = %o\n",
 x, y, x & y);
    printf("x =\t\t%o\ny =\t\t%o\nz = x | y = %o\n",
 x, y, x | y);
    printf("x =\t\t%o\ny =\t\t%o\nz = x ^ y = %o\n",
 x, y, x ^ y);
    printf("w = %d, w << 2 = %d, w >> 2 = %d\n",
 w, w << 2, w >> 2);
    return (0);
}
```

```
/* Program 10.17:  Output:  */
w = -1, one's complement = w̃ = 0
x = 12345
y = 23456
z = x & y = 2044
x = 12345
y = 23456
z = x |y = 33757
x = 12345
y = 23456
z = x ŷ = 31713
w = -1, w « 2 = -4, w » 2 = -1
```

The second way to manipulate bits within a word is by creating a field structure. Groups of contiguous bits within a word are called a field. A field is assigned both a name as well as a width, this being the number of bits in the field. The value stored within a field must be an **int**-like quantity:

```
struct bitfields
{
unsigned x : 2;
unsigned y : 2;
unsigned z : 3;
unsigned z : 1;
} psw;
```

This defines a variable called psw which contains four unsigned bit fields.

The number following the colon is the field width. Field variables may be assigned values; be careful however that the value assigned to a field is not greater than the maximum storable inside it. Individual fields are referenced as though they were structure members. The assignments:

psw.z = 5;

set the bits in the field z as 101. The **signed** or **unsigned** specification makes for portability; this is important because bit fields are extremely implementation dependent. For example, C does not specify whether fields must be stored left to right within a word, or vice versa. Some compilers may not allow fields to cross a word boundary. Unnamed fields may be used as fillers. In declaring the structure pc as follows we have forced a 3 bit gap between the fields x and y:

```
struct
{
unsigned x  : 2;
            : 3;
unsigned y  : 4;
} pc;
```

An unnamed field of width 0 will cause the next field to begin in the following word instead of at the boundary of the last field. The fields within a word do not have addresses. It is incorrect to use the & (address of) operator with them.

E10) Write a C language program to count the number of bits in your computer's word.

E11) Write a C program to implement right or left rotation of bits within a word. n bits (where n is a positive integer less than the word size) are pushed out to the right or left, caught as they fall, and pushed in from the left or right respectively.

We now end our discussion by recalling what we have discussed so far in the next section.

10.9 SUMMARY

In this Unit we have studied how disk files may be read and written by C programs. File I/O functions are for the most part similar to the keyboard/monitor I/O functions that we have already seen in the preceding Units; they are therefore particularly easy to use.

fopen()

`fopen(`*`string_pointer`*`,`*`options`*`)`
where the `string_pointer` is either the file name itself as a string(like `"password.dat"` for example) or a pointer to a string containing the file name. The options are given in Table. 1 on page 121.

fprintf()

`fprintf(`*`file pointer`*`,`*`control string`*`,`*`expressions`*`)`
This is similar to `printf()` but for the extra argument, the file pointer.

fscanf()

`fscanf(`*`file pointer`*`,`*`control string`*`,`*`expressions`*`)`
This is similar to `scanf()` but for the extra argument, the file pointer.

sprintf() and sscanf()

`sprintf(output_buffer, control_string, list);`
`sscanf(input_buffer, control_string, list);`
are for reading strings writing output to strings.

fgets() and fputs()

`fgets(pointer to a char array, N chars,file pointer)`
is for reading an entire line from a file pointed to by the *file pointer*. If the number of characters in the line is more than $N-1$ it reads the N – 1 characters; otherwise it reads the entire line.
`fputs(pointer to a char array, file pointer)`
writes the string to the file.

getc() and putc()

`getc(file pointer)`
reads a character from the file that *file pointer* points to. It returns EOF if it reaches the end of the file or there is an error.
`putc(Character to be written.,file pointer)`
writes the *Character to be written* to the file pointed to by the *file pointer*

fread() and fwrite()

Both functions have four arguments. The first of these is the address of a memory buffer from which `fwrite()` writes into a file, (or into which `fread()`read from a file). The second argument of each functions is the size of the items to be written or read; the third, the number of items; and the fourth, the file pointer.

rewind()

The function `rewind(file pointer)` makes the *file pointer* to point to the beginning of the file.

feop()

The function `feop(file pointer)` can be used to check if the file pointer is pointing to the end of the file.

fseek()

`fseek(`file pointer`,offset,origin)`
allows us to access any portion of the file. The first argument is the *file pointer*. The third is the position in the file, 0 for the beginning, 1 for the current position and 2 for the end of the file. The second is the offset from the origin specified in argument 3.

Structs are a C facility that allow the storage of heterogeneous but related information. A struct variable has members which can be accessed by the dot and arrow operations.

A **union** is a type of **struct** that can hold at any time just one of its members, which may be of various types. It is similar to Pascal's variant records.

It is possible to manipulate the bits within an int-like word. There are two ways by which this may be accomplished. The first method depends upon the six bitwise operators of C: The second uses the approach of bit fields, which are groups of

Table 2: 6 bit operations.

Operator	Action
x & y	Finds the AND of x and y. In the result a bit is 1 if and only if both the corresponding bits in x and y are 1; otherwise it is zero.
x |y	Finds the OR of x and y. In the result a bit is 0 if and only if both the corresponding bits in x and y are 0; otherwise it is 1.
x ^y	Finds the exlusive OR of x and y. In the the result a bit is zero if and only if both the corresponding bits in x and y are the same.
x «n	Pushes out the n left most bits and replaces the right most n bits vacated bits by 1.
x »	Pushes out the n right most bits. If x is an unsigned int, it replaces the vacated bits by 0s. If x is a signed int, the result is compiler dependent; the vacated bets are replaced by 0s or by 1s depending on whether the compiler implements **logical** or **arithmetical** shifts.

contiguous bits within a word, and which can be assigned small integer values. Bit fields are accessed like structure members.

10.10 SOLUTIONS/ANSWERS

E1) See the program in Listing 1. This is just a modified version of the program in page 20 of Kernighan and Ritchie's book, second edition. Create a small file to test the program and work through the program to find what happens when your file is processed by the programme.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define IN 1 /*inside a word.*/
#define OUT 0 /*outside a word.*/
int main(int argc, char *argv[])
{
    FILE *file;
    char name[20];
    int c, nl, nw, nc, state;
    nl=nw=nc=0;
    state = OUT;
    strcpy(name,argv[1]);
    printf("File name is %s.\n",name);
    if((file=fopen(name,"r")) == NULL){
        printf("Unable to open file.");
        exit(1);
    }
    else{
        while((c=getc(file)) != EOF){
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if(state == OUT){
            state = IN;
```

```
                ++nw;
            }
            }
        }
        printf("%d %d %d\n", nl, nw, nc);
        return (0);
}
```

Listing 1: Solution to exercise 1.

E8) The program is given in Listing 2. As in the program to count the number of lines, words and characters in a file, create a small example file and work through it.

```
/*Answer to exercise 8; file name:unit10-ans-ex-8.c*/
#include<stdio.h>
#define MAXSIZE 100
int main(void)
{
    char fname[MAXSIZE];
    FILE *fp;
    fprintf(stderr, "\nInput a file name:");
    scanf("%s",fname);
    fp = fopen(fname,"r");
    fseek(fp,0,2);/*Go to the end of the file.*/
    while(fseek(fp,-1,1) == 0){
/*move back by a character if possible.*/
        putchar(getc(fp));
/*Move forward one character and print it.*/
fseek(fp, -1, 1);/*Move back on character.*/
    }
    printf("\n");
    return (0);
}
```

Listing 2: Solution to exercise 8.