UNIT 10: FILES AND STRUCTS, UNIONS AND BITMFIELDS

Structure

10.0 INTRODUCTION

Files are the most visible and the most valuable feature of any computer system to its users because they provide a mechanism for the long term storage of data; they remain inviolate even if a computer's power is switched off. Computers would be no use at all if they did not provide a mechanism for the permanent storage of information.

Files are stored in the concentric circular tracks of a floppy or a "hard" disk. A hard disk, un- like a floppy, consists of several platters rigidly mounted on a common spindle; each surface of each platter may have several hundred tracks. These tracks contain a 'n even distribution of particles of a magnetic oxide, each of which can be permanently magnetised in one of two ways, to store a 0 or a 1. So the bi-stable circuits of a computer's RAM find an analogue in the dual states of magnetisation of particles of iron oxide on the disk's surface. As a disk spins in its "drive" (this is a motor driven platform on which the disk is fixed), a "read-write head can be made to home in to any of its tracks. In a hard (or even a floppy) disk drive, there's a head for every surface. These heads are joined to a common actuator arm; ffic arm (and so the heads) can move radially along the disk's surface, towards or away from its centre. Any head can be positioned on any track on the surface it serves. When a file is read, the contents of the track spinning past the head (the individual little magnets) induce cor- responding electric signals inside a coil in the read-write head; these signals flow. in to RAM, where they are stored as bits. Thus, reading a file means picking up the particulate magnefisa- tions on a disk's tracks and storing them as electric signals of 0 or 1 in the computer's RAM. Conversely, when a file is written the contents of RAM . electric signals of low or high volt- age - are made to travel as currents to the coil in the read-write head; these currents mag- netise the particles flowing by

under the head, while the disk spins, creating in them an image precisely similar to the bits from which the signals emanated. Thus are the contents of RAM captured on disk, and files are "written". Disk files are permanent to the extent that the magnetism of the particles in which they are stored is permanent.

The tracks on a disk's surface are divided into a number of sectors of equal size. Just as each word of RAM has an address, so also has each sector of a disk. These addresses are known to the computer's operating system. The operating system also knows which file is written on which sector(s) _ these need not be contiguous - and it also keeps a list of the free sectors on the disk. When a file is to be read, the operating system moves the read-write head to the track which contains the sector on which the file begins; the time that this head movement takes is called the "seek time". Having rcached its e;%signated track, the head waits there until the desired sector has appeared under it: this waiting period is called the "latency time". Finally the data from disk can be transferred to dw RAM; this third delay is called the "trans- fer times'. Large filv@s may occupy several wtors. which need not be contiguous. The read- write head may have to be. directed to sectors of different tracks Gn several surfaces to read large friiginented files. You can see that because mechanical motions are involved, a disk read or write must by CPU standards be a very slow operation; it is in fact hundreds of thousands of times slower than a CPU cycle. SO it,s just as well that in a read operation an en- tire sector is read at a time. Ite contents of a sector arc Placed in a block of RAM called the input buffer. Intelligent programmers organise their data in disk files so that when further data items are needed by their program. they may already be in the input buffer from the last read operation.

In a read operation, precisely how the rcad-write head is directed to a disk's track from which it must pick the chosen file from the thousands that May be resident on the disk, and how it performs the read while the magnetised particles rush past it (sometimes as fast as two hundred km. per hour!) is a matter of enormous and quite awe-inspiring complexity. Fortunately for us, as users of the computer we have nothing whatsoever to do with this complexity. How C finds where exactly our riles are on a given disk. and how it rinds free tracks therein to store the new files that we may create, is a matter that it must negotiate with the computer's operating system. That's not our concern. What is nice for us is that C provides a us with a very convenient interface to deal with files. Its convenience lies in the fact that the functions for I/O that we have so far used . printf 0, scanf 0, puts (, gets 0, putchar 0 and getchar 0 for monitor and keyboard I/O . have quite similar counterparts for disk I/O. There is a special reason for this: the functions for keyboard input and monitor output that we are familiar with, are themselves implemented via files. These functions depend upon three files that work in the background, generally invisible to us. The files, declared in < stdio.h , are the standard input, the standard output and the standard error: we shall have more to say about them later.'Me good news is, if you know how to use scanf 0 and printf 0, you know how to read from and write to disk files! Designers of operating systems call this uniformity of interface "device independence".

Let us now turn our attention to the organisation of information within files. Suppose that you want to store data about the passbook entries for a savings bank accounl For each trans- action you would want to store a variety of items of different types; to make our example more concrete. suppose that you?ve received a cheque, and you'd like to record the following details about it: by whom given, of what amount, on which bank, whether local or outstation, and date and number of cheque. How will this information be organised? It would be con- venient if there was a single data receptacle that could hold all this information in one place.

Clearly that receptacle couldn't be an array. An array can only hold items of just one type, ints or chars or whatever, the information that you need to store happens to involve a variety of types: char arrays, ints and a float. What you're looking for is a data structure that in C parlance is called the struct. A struct. similar to Pascal's record, can hold several kinds of items, and enables you to maintain related information of different types in a single objecl Structs are introduced in this Unit.

A **union** is a C device that allows you to view a single storage area in more than one way; a variable declared as a union can store different kinds of data at different times.

C provides a .variety of operators to manipulate the contents of a packed field of bits. In sys- tems programs there is very often a need to set up a number of "flags" . values that must be 0 or 1: sensing these tells about the status of the system. and helps decide on a suitable course of action. Now, since a single bit

is enough to represent a status, it makes little sense to use a byte or word for the purpose. In C it's possible to "pack" bits into a "field". where they may be set and tested. Bit operations are the final topic covered in this Unit.

10.1 OBJECTIVES

After reading this Unit you should be able to:

- declare files and perform file I/O
- declare. Create and operate on instances of structs
- declare and operate on unions
. use the bit operators on packed fields

10.2 FILES AND FILE I/O

Before a file can be used for reading or writing. it must be opened. This is done. by a can to the the fopen (function. fopen (takes two string arguments. therefore enclosed in double quotes. The first of these is the file's name; the second is an option (r, w, a, etc.) that tells C what we want to do with the file: read it, or write to it@ or append to it@ etc. File handling functions are prototyped in <stdio.h, which also includes other needed declarations. Naturally this header must be #included in all your programs that work with files. Table 10.1 lists the options available with fopen 0.

Table 10.1.

File Opening Options

Option	**Action**
r	Open existing file for reading.,
w	Open (create if need be) file for writing; discard previous contents;
a	Open (create if need be) file for appending;
r+	Open existing file for reading and writing;
w+	Create and open file for reading and writing; discard Previous contents;
a+	Open (crew if need be) file for reading and appending.

A call to fopen ( ) to read the tile **passbook.dat** looks like.

fopen (passbookdat". "e)

If fopen (is successful in its mission of opening die file for reading, it returns a new type of pointer, a pointer of type FILE; FILE is a typedef defined in <stdio.h. If fopen 0 cannot successfully open the file, it returns instead the NULL pointer.

To connect your program to the file that fopen (opens, your program must also declare a pointer of type FILE: that declaration is easy:

FILE * rile_ptr.

Now you can assign to fileptr the pointer that fopen (returned:

file-ptr = fopen Cpassbook.dat", "r');

In this way ftle_ptr provides you read access to the file passbook.dal You need never again refer to the file itself by name: file_ptr will do all the required book-keeping for you. such as holding your place in the file, etc.

In typical concise fashion C programmers combine the file opening operation with a check to see if the pointer returned by fopen (was NULL:

```
if ((file_ptr = fopen C'passbook.daC, "r")) != NULL)
  printf ("Had no difficulty in opening passbook.dat\n');
else
  printf e'Regrets ... couldn't open passbookdat");
```

Even inore concisely, one writes:

```
if (file_ptr = fopen ("passbook.dat', "r')) etc.
```

A call to hose 0 closes a previously opened file, flushing as it does so any associated buffer; the single argument of fciose (is the relevant file pointer. Let's now look at Programs 10.1 - 10.5.

10.2.1 fprintf ( )

Program 10. 1 illustrates how the duee parts of the argument list of fprintf 0, the function to write to a file opened by fopen 0, am written: the first is the file pointer, the second is the control string. and the third is the list of variables or expressions to be written:

```
fprintf (filc_l, "%s", "Writing to a disk file is\n");
```

```
/* Program 10.1 */
#include <stdio.h
void main (void)
        {
          FILE *file_1;
          if ((file_1 = fopen ('newfile.dat", "w')) != NULL)
                  {
                    fprintf (filip-1, "%s". "Writing to a disk file is\n");
                    fprintf (file-1, "%s", "quite coy, as no doubt you\n');
                    fprintf (filip-1, "%s", "will readily agree.\n');
                    fclose (file-1);
                   }
              else
                printf ('Unable to open newfile.dat for writing");
           }
```

Executing Program 10. 1 creates the 77 byte file. named newrile.dal Here are its contents:

```
Writing to a disk file is
quite easy. as no doubt you
will readily agree.
```

Program 10.2 reads 5 int values from the keyboard, and stores them in die file numbers.dat:

```
/* Program 10.2 */
#include <stdio.h
void main (void)
```

```
{
  FILE * ptr, int numbers [51; int i;
  if ((ptr = fopen C'numbers.dat". "w")) != NULL)
        {
          printf e'Enter 5 numbers, to be stored in numbcrs.dat...");
          for (i = 0-, i < 5; i ++)
                {
                  scanf C'%d", &numbers [i]);
                  fprintf (ptr, "%ft", nwnbers [i]);
                }
             fprintf (Ptr. 4"1ng);
             fclose (ptr);
           }
     else
       printf C'Unable to open numbers.dat for writing...'\n"); '
   }
```

Here is the result of executing Program 10.2:

Enter 5 numbers, to be stored in numbers.dat ... 1 2 3 4 5

The file numbers.dat was created:

1 2 3 4 5

10.2.2 fscanf ( )

Similarly fscanf 0 reads files. Its argwnents are written as shown below:

fscanf (file-ptr, "control string", list of pointers);

Program 10.3 reads the file numbers.dat, and sums its entries.

```
/* Program 10.3 */
#include <stdio.h
void main (void)
        {
          FILE * ptr; int val, i, sum = 0;
          if ((ptr = fopen C'numbers.dat', "r")) NULL)
                {
                  for (i =0;i < S;i++)
                        {
                          fscanf (ptr. 44%d'sq & val);    sum +-. val; }
                     printf C'The sum of the entries in numbers.dat is %d\n", sum);
                     fclose (ptr);
                   }
              else
                printf C'Unable to open numbers.dat for writing ... \n");
            }
                                        /* Program 10.3: Output: */
```

The sum of the entries in numbers.dat is 15

10.2.3 stdin, stdout and stderr

Now you might think that calls to fprin.f ( )and fseanf ( ) differ significantly from calls to printf ( ) and scant' ( ): these latter functions didn't seem to require file pointers. As a matter of fact they do. The file pointer associated with printf ( ) is a constant pointer named stdout. defined in . Similarly scanf ( )has an associated constant pointer named stdin. scanf ( )reads from stdin and printf ( )writes to stdout, as you may verify by executing Pro- grain 10.4 below.

```
/* Program 10.4 */
#include < stdio.h
void main (void)
                {
                  int first, second;
                  fprintf (stdout, "Eriter two ints in this line: ");
                  fscanf (stdin, "%d %d". &first. &second);
                  fprintf (stdout, "Their sum is: %d.\n", first + second);
                }
```

There is a third constant file pointer defined in named stderr. This 's associated with the standard error file. stderr has the following use: in some system$ such as PC-DOS and Unix, you can redirect the output of your programs to foes, by using the redirection operator, . In DOS, for example, if fl.exe is an executable rile that writes to the monitor. then you can redirect its output to a disk file fl.o by the command:

fl fl.o<CR>

Output that would normally appear on the monitor can thus be sent to a rile. On die other hand, if you were redirecting output, you wouldn't want any error messages such as:

"Unable to open newfile.dat for writing"

to be redirected; you'd want them to appear on the screen. Writing error messages to stderr:

fprintf (stdeff, "Unable to open newfile.dat for writing");

ensures that normal output will be redirected, but error messages will still appear on the screen.

10.2.4 sprintf ( ) and sscanf ( )

printf () and fprintf () are therefore close cousins, as are scanf () and fscanf (); but there are two other useful members of this family, namely sprintf () and sscanf (). sprintf () writes its output in a string, whose address is its first argument; as before, its second argument is the format control string, and this is followed by a list of variables or expressions:

sprintf (output_buffer, control_string, list);

Likewise, sscanf () can read a string and store its contents into the pointers constituting list

sscanf (input_buffer, control_string, list);

10.2.5 fgets ()

Recall from Unit 7 that scanf () is not the most convenient function to read strings with - it regards white space as a string delimiter_and fscanf () is not different in this regard. But the fgets () function, like the gets

(), is easy to use if you wish to read a whole line from a file at a time. It's called with three arguments: a pointer to a char array that will store the string read, an integer specifying the maximum number of characters to be read, say n, and a file pointer specifying the file to be read. Program 10.5 uses an array of 50 chars. named input buffer [], in which fgets () deposits the contents of newfile.dat, a line at a time. Fgets () rears characters until a now line occurs in the input., which is stored. or until it has read n - 1 characters. It appends the null character after the last character read. As a function, fgets returns the value of its first argument (the pointer input buffer) if a read was successful, otherwise the NULL pointer. This property is exploited in the while () loop of Program 10.5:

```
/* Program 10.5 */
#include <stdio.h
void main (void)
                {
                  FILE *file_1;
                  char input_buffer [50];
                  if ((file_1 = fopen ("newfile.dat", "r")) != NULL)
                          {
                            while ((fgets (input_buffer, sizeof (input_buffer), file-1)) ! = NULL)
                                    printf (input_buffer);
                                fclose (file_1);
                              }
                       else
                         printf ("Sorry, unable to Open newfile.dat\n);
                     }
```

10.2.6 fputs ( )

Similarly, fputs (), used in Program 10.6, writes a string in a file. Its two arguments are the memory address of an input string and the file pointer. The input string is written upto (but not including) its null character. Program 10.6 accepts the names of two files in its command line. It copies the first file into the second. line by line.

```
/* Program 10.6 */
#include <stdio.h
#include, <stdlib.h
void main (int argc, char * argv ∩)
                {
                  FILE *file_1 *file_2;
                  char input_buffer [50];
          /* contd. */

               if ((file_1 = open (argv [1], "r")) = = NULL )
                          printf ("Unable to open %s for reading \n", argv [1]);
                    else if ((file_2 = fopen (argv [2], "w")) == NULL)
                          printf ("Unable to open %s for writing \n", argv [2]);
                    else
                          {
                                  while ((fgets (input_buffer, sizeof (input_buffer), file_1)) ! = NULL)
                                          fputs (input_buffer, file_2);
                                          fclose (file_1);
                                          fclose (file_2);
                              }
                     }
```

10.2.7 getc ( ) and putc ( )

getc ( ) and putc ( ) are two character I/O functions for files. very similar to getchar ( ) and putchar ( ) respectively. The single argument of gete ( ) is an input file pointer. it fetches the next character from the file, and returns EOF at end of file, or if an error occurs. pute ( ) has two arguments: the character to be written, and a pointer to the output file. Why has the variable next.char in Program 10.7 been declared as an int?

```
/* Program 10.7 */
#include <stdio.h
void main (void)
                        {
                          FILE *file-1;
                          int next_char;
                          if (file_1 = fopen C4newfile. &Clt "e))
                                  {
                                    while ((next_char = getc(file_1))!=EOF)
                                            putc (next_char, stdout);
                                       fclose (file_1);
                                    }
                              else
                                 printf ("Sorry, unable to open newfile.dat\n");
                             }
```

Realise from Program 10.7 that putchar (x) is equivalent to putc (xg stdout). and getchar (is equivalent to getc (stdin).

10.2.8 fread ( ), fwrite ( ), rewind ( ) and feof ( )

Program 10.8 uses two ANSI C functions for direct I/O. The fread ( )and pyrite ( )functions do not interpret ( )i.e. format( ) what they read or write. You will therefore not in general be able to display, i.e. type or cat to the screen a file created by fwrite ( ); instead, fread ( ) must be used to read it, as Program 10.8 illustrates. The fread ( )and fwrite ( )functions have four arguments., The first of these is die address of a memory buffer from which fwrite ( )writes into a file, ( )or into which fread ( )reads from a file( ). The second argument of each function is the size of the items to be written or read; the third, the number of items; and the fourth, the file pointer. The call:

```
fwrite (numbers, sizeof (double), NUM EL, f I)*
```

writes NUM_EL doubles from a buffer numbers into a file associated with the pointer f 1.

The call:

```
fread (copy, size of (double), NUM-EL, f1);
```

reads NUM EL doubles from a file associated with the pointer fl into a buffer named copy.

Program 10.8 opens (creates) a file doubles.dat for both writing and reading: observe the "w+" option in the call to fopen (). numbers is the name (i.e. address) of a buffer of dimension NUM EL. The doubles stored in numbers are written into doubles.dat; but this rile cannot be Displayed on the screen because the objects written in it are in the format in which they were stored in memory. To look at the contents of the file, you'll have to access it via an fread (). However, the file pointer after the last write is stationed at the bottom of the file; before you can begin to read it you must bring its pointer back to the top. The call:

```
rewind (fl);
```

brings the read pointer back to the beginning of the file.

The call to fread ( ) in Program 10.8 reads the file into the buffer copy. THe for **(;;)** loop prints the contents of copy [ ].

```
/* Program 10.8 */
#include <stdio.h
#define NUM-EL 100
void main (void)
                {
                  FILE *f1;
                  static double numbers [NUM_EL] =
                                                                    {
                                                                      1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
                                                                    };
                     static double copy [NUM_EL];
                     int i;
                     if ((fi = fopen ("doubles.dat", "w+")) != NULL)
                       fwrite (numbers, sizeof (double). NUM-EL, fl);
                      else
                        printf ("Error writing in file doubles.dat...\n");
                      rewind (n);
                      fread (copy, sizcof (double), NUM_EL, fl);
                      for (i = 0-, copy [i] 0.9 i ++)
                        printf C"%.of\n", copy [i]);
                      fclose (f 1);
                    }
```

The end-of-file condition can be tested by a call to feof (); this function, whose single argument is a file pointer, returns a non-zero value if an attempt is made to read beyond the end of a file:

```
if (feof (fp))

        printf ("Error: cannot read beyond EOF");
```

Check Your Progress 1

Question 1: Write a C program to count the number of characters, words, and in lines in a text file whose name is supplied in the command line.

Question 2: Write a C program to implement a computer based memo-pad, in which the user may write appointment schedules. notes for meetings, memoranda, etc. (You may use the standard header < time.h to record and compute with dates and times.) The pad should be created the first time that the program is used; each subsequent execution should append to it. At the end of the year the pad should be closed, and a new pad should automatically be created.

10.3 STRUCTS

Arrays. as we have seen in the previous units. provide a means to bundle homogeneous data items into a single named unit. But more often than not "real world" applications require the grouping together of

heterogeneous data items. Consider once more the information that one would want to store about depositing a cheque in one's savings account:

i. by whom issued;
ii. on what bank;
iii. whether cheque was local or outstation;
iv. cheque number
v. date of issue
vi. date of deposit
vii. amount of cheque

Of the items in our list, the first four are stings; the fifth and sixth, the dates of issue and deposit. consist of triplets of ints; and the seventh is a float. If we considered using arrays to store the records of several similar transactions, then we would need eleven in all: four two- dimensional arrays for the strings; three int arrays each for the two dates; and one of floats to hold the amounts of the cheques. Every transaction would require us to access these eleven arrays: nor could we know the appropriate number of elements to specify for them: altogether an approach to be avoided.

Structs are a C facility that permit several heterogeneous but interrelated items to be treated as a single unit. The keyword struct introduces such a unit:

```
struct cheque-deposited
        {
        char by_whom_given [30];
        char bank [30];
        char local_or_not [2];
        char cheque_number [10];
        int dd_issued;
        int mm_issued;
        int yy_issued;
        int dd_deposited.,
        int mm_deposited;
        int yy_deposited;
        float amount;
        };
```

This struct declaration names eleven members; (arrays have elements; structs have members). But it reserves no storage. It creates a template that can be used to define objects of type struct cheque_deposited:

```
struct cheque_deposited nobel_prize, copley_medal, fields_medal;<
```

Here nobel _prize, copley_medal and fields_medal are structure identifiers. Each of these structures can store the details of a cheque, just as much as an int can store an integer value. The scientist who wins a Nobel Prize, a Copley Medal and a Fields Medal can store the details of the cheques he receives in these structures. The name cheque _ deposited is a structure tag, and is optional: structs may be defined without a tag, as for example the following structs to store the dates of our national holidays:

```
struct
        {
        int dd;
        int mm;
        int yy;
        } gandhi_jayanti, independence_day. republic_day;
```

Here gandhi_jayanti,. Independence_day, republic _day are three structures dart can store dates. Because the struct delaration is not tagged. a type has not been created to store any other dates.

The advantage of a tag is that further instances of structures can subsequently be created by simple definitions using the tag as a type. Consider a struct date declared externally to main ():

```
struct date    (
                    int dd;
                    ifit mm;
                    int yy;
                }
```

Then other structures of type date may be defined in the functions that follow:

```
struct date gandhi-jayanti, independence_.day republic_day.
   struct date Abhishek-birthday. Aparajita_birthday;
```

Arrays of structures may also be defined:

```
struct cheque_deposited bunch 10001;
```

bunch is an array of 1000 structs of type cheque_deposited.

A structure tag may therefore be thought of as a user defined type, like oat or double. The sizeof operator can be used to determine the size of a struct type, or of its instances:

```
printf ("%d\n". sizcof (struct date));
printf ("%d\n", sizeof nobel_prize);
```

Remember, in this last usage the sizeof operator will not return the value of the Nobel Award in dollars_only the size of die structure nobel_prize in bytes!

Check Your Progress 2

Question 1: Show how you will declare a struct to hold employee payroll data: name, department, date of birth, date of joining, basic salary, dearness, house-rent and other allowances, provident fund, life insurance and income tax deducations, and contributions towards CGHS, ESI etc.

Question 2: Show how you will declare a struct to hold the fee-bill of student in a school. The structure must be able to record the following information: student name, date of birth,, class and section, father's name address and telephone number, whether student uses the school bus, distance she travels on the bus (this information is used to compute the buss bill), tution fee, and fees chargeable for extracurricular activities, such as swimming, skating, etc.

Question 3: Show how you will declare a struct to record information about students enrolled for a degree program at a University. The University offers B.A., B. Sc., M.A., M.Sc., and Ph. D. degrees in several subjects. Each student is identified by a unique enrollment number.

10.4 THE DOT OPERATOR

In order to gain,access to a member of a structure, say float amount in the cheque for the Nobel Prize, we use the dot operator, "."

```
structure,_name.member_name
```

as in the **printf** ( ) below:

printf ("Congratulations on your award! Amount deposited = %,2f"s

nobel.prize.amount);

Though the dot operator may appear to you as a binary operator in the above expression, ii has in fact a priority as high as the parentheses or the subscript operators, and like them it groups from left to right

The members of a structure can be initialised to constant values by enclosing the values to be assigned in braces after the structure's definition (non ANSI C compilers may require the static keyword in the lead):

```
/* static */ struct date Gandhiji_birthday = (
                                                2,10,1869
                                              }
                                    Nehruji_birthday =   (
                                                14,11,1889
                                              }

/* static */ struct cheque_dcposited delhi_lottery =     {
                                                    "Delhi Administration",
                                                    "State Bank of India,
                                                    "Y"
                                                    "381654729"
                                                    2, 5, 1994, 3, 5, 1994,
                                                    50000000.00
                                                };
```

If there are fewer values listed in an initialisation than the number of members present, then the remaining members are initialised to zero by default.

To print the approximate difference in the ages of Gandhiji and Nehruji we can write:

printf ("Age difference = %d\n",

Nehruji _birthday.yy - Gandhiji_birthday.yy);

Program 10.9 illustrates how the dot operator is used to extract the members of a structure:

```
/* Program 10.9 */
#include <stdio.h
struct cheque_deposited
                {
                  char by_whom_given [30];
                  char bank [30];
                  char local_or_not [2];
                  char cheque_number [10];
                  int dd_issued;
                  int mm_issued;
                  int yy_issued;
                  int dd_deposited;
                  int mm_deposited;
                  int yy_deposited;
```

```
    float amount;
};

void main (void)
{
   /* static */ struct cheque_deposited delhi_lottery =
                                                    {
                                "Delhi Administration",
                                "State Bank of India".
                                "Y"
                                "381654729"
                                2, 5, 1994, 3, 5, 1994,
                                50000000.00
                                                    };
    printf ("Cheque date: %d-%d-%d\n".
          delhi_lottery.dd_issued,
          delhi-lottery.mm-issued,
          delhi lottery.yy_issued);
    /* contd. */


  printf ("Cheque number: %s\n", dclhi_Iottery.cheque_number);
  printf ("Drawn on: %s\n", dolhi_lottery..bank);
  printf ("Amount Rs. %.2f, delhi_Iottery.amount);
}
                              /* Program 10.9: Output: */

Cheque date: 2-5-1994
Cheque number: 381654729
Drawn on: State Bank of India

Amount Rs. 50000000.00
```

Finally, structures may be nested. Given the prior declaration of struct date. struct cheque_deposited may be declared:

```
struct cheaue_deposited
{
  char by_whom_given [30];
  char bank [30];
  char local_or_not [2];
  char cheque_number [10];
  struct date issued*.
  struct date deposited;
  float amount;
};
```

Programs 10.10- 10.11 illustrate struet declarations and the use of the dot operator. Program 10.10, which declares struct complex. implements the addition of complex numbers. Its members store the real and imaginary parts of three complex numbers. val_1. val_2 and their complex sum, named result.

```
/* Program 10.10 */
#include <stdio.h
struct complex
{
  float real;
```

```
                float imaginary;
              };
main ( )
              {
                struct complex val_1, val_2, result;
                printf ("This program adds two complex numbers.\n");
                printf ("Enter real and imaginary parts of first number:");
                scanf ("%f %f". &val_1. real, & val_1.imaginary);
                printf ("Enter real and imaginary parts of second number: ");
                scanf ("%f %f", &val_2.real, &val_2.imaginary);
                regult.real = val_1.real + val_2.real;
                result.imaginary = val_1.imaginary, + val_2.imaginary;
                printf ("The sum is %f + i * %\n", result.real, result.imaginary);
              }
```

Program 10.1 1 uses nested structures: struct date is used in the declaration of struct cheque deposited. Because the dot operator associates from left to right, in order to access die month parameter of struct date of structure nobel_prize, we write:

```
                                                nobel_prize.issued.mm;
```

```
/* Program 10.11 */
#include <stdio.h
struct date
              {
                int dd;
          /* contd. */


                int mm;
                int yy,
              }

struct cheque_deposited
              {
               char by_whom_given [30];
               char bank [30];
               char locai_or_.not [2];
               char cheque_number [10];
               struct date issued;
               struct date deposited;
               float amounts;
              }

nobel_prize;
void main          (void)
                   {
                     printf ("Size in bytes of struct cheque_deposited is %d\n".
                                        sizeof (struct cheque_.deposited));
                     printf ("'Size in bytes of struct date is %d\n", sizeof (struct date));
                     printf ("Sizeof month parameter in struct date of");
                     printf ("structure nobel_prize is: %d\n', size nobel_prizc,.issued.mm);
                   }
```

Check Your Progress 3

Question 1: Write a C language program to create an array of structures to record and print employee payroll data. (Refer to Question 1 above).

Question 2: Write a C language program to create an array of structures to record and print student fee payment data. (Refer to Question 2 above.) Bus fees will depend upon actual distance between home and school, which must be input to the program in each case. Fees may be paid in cash or by cheque, If fees are paid by cheque, your program should record relevant details

Question 3: Write a C program to record and print data about students enrolled in the various courses at a University. (,Refer to Question 3 above.)

10.5 STRUCTS AND FILES: fseek ( )

Structs are customarily associated with files and databases. Our program to record the transactions listed in a passbook must store the information in riles if it's to be any use at all. Naturally, because details about cheque issues and deposits must be different from those for cash withdrawals and deposits, it may be simplest to have separate files for each of these activities. Program 10.12 below is a "quick and dirty" way of writing the information about deposited cheques to a file named cheqdpst.dat, and fetching the data about any cheque already deposited. It's quick and dirty because it uses the same char array, namely value-holder [], to read the number of records to be entered. the amount on each cheque deposited, and the number of the record to be fetched; nor does the program store the number of transactions currently held in cheqdpst.dat. Moreover, it has no error checking, no facility for data correction after entry, no input for date variables, etc. But these features are easily built in. The new, and extremely useful feature that this program introduces, is the C capability to seek out an arbitrarily chosen record from the hundreds or thousands that may exist in cheqdpst.dat. The fseek () function allows a file to be treated as though it were an array of bytes: we can position ourselves anywhere inside that we want, and read any number of bytes forwards or backwards from it!

The "a+" option in the call to fopen (opens the rile cheqdpst.dat for appending and reading. If the file does not exist, this call to fopen () creates it.

```
/* Program 10. 12 */
#include <stdio.h.
#include <stdlib.h
#define NUM_CBEQUES 1 00
struct cheque_deposited
                {
                  char by_whom_given [30];
                  char bank [30];
                  char local_or_not [2];
                  char cheque_number [10];
                  float amount;
                };

void main (void)
                {
                 FILE *fp;
                 struct cheque_dcposited cheques [NUM_CHEQUES], cheque_holder;
                 int number, rerd_nmb, i;
                 char value_holder [30];
                 float amt_deposited = 0.0;
                 if ((fp = fopcn("cheqdpst.dat", "a+"))!= NULL)
```

```
                    {
                      printf ("How many cheques to be deposited?")
                      gets (value_holder);.
                      number = atoi (value_holder);
                      for (i = 0; i <> number; i ++)
                              {
                                printf ("\t\t\t.... Enter details for cheque %d.....\n", i + 1);
                                printf ("Cheque issued by: ");
                                gets ((cheques [i] ).by_whom_given);
                                printf ("'Bank name: ");
                                gets ((cheques [i] ).bank);
                                printf ("Local bank (Y/N): ");
                                gets ((cheques [i] ).local_or_not);
                                printf ("Cheque number: ");
                                gets ((cheques [i] ).cheque_number);
                                printf ("Cheque amount: ");
                                gets (value_holder);
                                cheques [i].amount = atof (value_holder);
                                amt_deposited + = cheques [i] .amount;
                              }
                          fwrite (cheques, sizeof (struct cheque_deposited), number, fp);
                          printf ("Amount deposited was: %.2n", amt_deposited);
                          printf ("Which record number to be retrieved... ?");
                          gets (value_holder);
                          rcrd_nmb = atoi (value_holder);
                          rewind (fp);
                          if ((fseek (fp, (long) ((rcrd_nmb - 1) * sizeof (struct cheque_deposited), 1, fp );
                              {
                                fread (&cheque_holder, sizeof (struct cheque_deposited), 1, fp);
                                printf ("Cheque details are as under.\n");
                                printf ("Cheque issued by: ");
                                puts (cheque_holder.by whom_given);
                                printf ("Bank name: ");
                                puts (cheque_holder.bank);
                                printf ("Local bank (Y/N): ");
                                puts (cheque._holder.local_or_not);
                                printf ("Cheque number: ");
                                puts (cheque_holder.cheque_number);
                                printf ("Cheque amount: ");
                                printf ("%.2f\n",cheque_holder.amount);
                              }
                          else
                            printf ("Sony, unable to locate record %d\n", rerd_nmb);
                          fclose (fp);
                        }
/* contd */

                                  else
                        printf ("Error opening passbook.dat for appending records ... \n");
              }
```

10.5.1 face ( )

fseek (needs three arguments: the first is a file pointer; the second, a long offset value, which is die number of bytes the position pointer Must move in order to home in to the chosen byte: the third argument of fseek

(specifies. the origin from which offset is measured: this can be , 1 or 2. 0 sets the file beginning as the origin of the search, 1 sets the current position in the file as the search origin, and 2 sets the end of the file as the search origin. fseek () returns a non-zero value on error

```
if ((fseek (fp, (long)
     ((rcrd_nmb - 1) * sizoof (struct cheque_deposited)), 0)) == 0)
etc.
```

10.6 STRUCTS AND FUNCTIONS: THE ARROW OPERATOR

Error checking is extremely important to any program that writes data to riles. Program, 10.13 validates a date read in as a struct via the function int valid date (). Each member of the structure any_date is passed to valid_date ( )as a separate parameter. The function cannot alter the values of the arguments sent to it. The function Zeller ( )is invoked only after valid_date () confirms that the three date parameters sent to it are good. As usual, it reports the day corresponding to the given date. Note that it does so without the switch statement.

```
/* Program 10.13 */
#include <stdio.h
#define LEAP_YEAR(Year) (Year % 4 == 0 && Year % 100!= 0)\

                                    || Year % 400 == 0
struct date
          {
            int dd
            int mm;
            int yy;
          };

int valid_date (int D, int M, int Y);
int Zeller (int D, int M, int Y);
char * days [] = {
                    "Sunday", "Monday", "Tuesday", "Wednesday",
                    "Thursday", "Friday", "Saturday"
                  };

main ( )
               {
                 struct, date any_date;
                 printf ("This program reads a date in the format dd-mm- yyyy\n");
                 printf ("and determines if its valid. If it is, it returns the\n");.
                 printf ("corresponding day. Enter a date: ");
                 scanf ("%d-%d-%d", &any_date.dd, &any_date.mm, &any_date.yy);
                 if (valid_date (any_date.dd. any_date.mm. any_date.yy))
                   printf ("That was a %s",
                   days [Zeller(any_date.dd. any_date.mm, any_date.yy)]);
                 else
                   printf ("Not a valid date.\n");
               }

int valid_date (int dd, int mm, int yy)
                         {
                           if (yy < 0)
```

```
        return 0;
      if (yy < 100)
        yy += 1900;
  /* contd. */


     if (yy 4902)
        return 0;
      if (!(LEAP_YEAR(yy)) && (mm == 2) && (dd 28))
        return 0;
      if ((LEAP_YEAR(yy)) && (mm == 2) && (dd 29))
        return 0;
      if (mm 12)
        return 0;
      if (dd 31)
        return 0;
      if ((dd 30) && (mm == 4 | | mm == 6 | | mm == 9 | | mm
        return 0;
      return 1;
    }

int Zeller (int DD, int MM, int YY)

    {
      int ZM, ZY;
      if (MM < 3)
        ZM =MM+ 10;
      else
        ZM=MM-2;
      if (ZM 10)
        ZY = YY - 1;
      else
        ZY = YY;
      return (((int) ((13 * ZM - 1) / 5) + DD + ZY % 100 +
      (int) ((ZY % 100) /4) - 2 * (int) (ZY / 100) +
      (int) (ZY /400) + 77) % 7);
    }
```

Instead of passing the members of a structure separately to a function (as was done in the program above), one can pass a structure to a function in toto, provided that the struct declaration is known to the function. As usual, if the values of the members are changed in the function, then those changes are restricted to the function. Program 10.14 uses a function to implement complex multiplication. The function has two structures for its parameters, each of type struct complex, and returns a value of the same type.

```
/* Program 10.14 */
#include <stdio.h
struct complex
    {
      float real;
      float imaginary;
    };

struct complex result (struct complex val_1, struct complex val_2);
void main (void)
        {
          struct complex num_l, num_2, product;
```

```
    printf ("This program multiplies two complex numbers.\n");
    printf ("Enter real and imaginary parts of First number:")
    scanf ("%f %f", &num_l.real, & num_l.imaginary);
    printf ("Enter real and imaginary parts of second number:");
    scanf ("%f %f", &num_2.real, &num_2.imaginary);
    product = result (num_1, num_2);
    printf ("Result: %f + i * %f\n", product.real, product.imaginary);
}

struct complex result (struct complex v 1, struct complex v2)

{
  struct complex answer,
  answer.real = vl.real * v2.real - v 1.imaginary * v2.imaginary;
  answer.imaginary = v1.real * v2.imaginary + v2.real * v 1.imaginary;
  return answer;
}
```

Pointers to structures are declared in the same way as pointers to any other variables:

```
struct complex * pc;
```

pc is now a pointer to a structure of type complex. The address of a structure of this type may be assigned to it:

pc = &num_l; (For ANSI C compliant compilers pc would now point to the first member of num_l; for pre-ANSI C compilers this may not be true.) Structure pointers may be passed to functions. This is often more convenient than passing (i.e. copying) an entire structure into the function, but there is an inherent risk of unintended changes to the structure. Within the function the real and imaginary parts of the complex number referenced by pc are given by:

(*pc).real and (*pc).imaginary.

The parentheses are necessary. Though the "contents of" operator binds very tightly to its operand, the membership operator has a higher priority. If die parentheses were absent the order of evaluation would have been:

*(pc.real) and *(pc.imaginary)

But pc is not a struct, it's a pointer. It doesn't have members; pc.real and pc.imaginary are therefore illegal constructs. On the other hand, the notation (*pc).real is cumbersome: so,C has as an alternative__the arrow operator - (which consists of a minus sign followed by the greater than sign) for the member reference operator. The member reference:

pc-real has the same effect as:
(*pc).real

Program 1 0. 1 5 shows how the arrow operator is used. The program determines the distance between two points (xl, yl) and (x2, y2). Each point is represented as a structure. and the addresses of these structures are passed to the function distance (). The arrow operator is used to extract the value of each of the members__the x and y co-ordinates__of the structures from their pointers. These values are used in Pythagoras's formula for the distance between (x1, y1) and (x2, y2).

```
/* Program 10.15 */
#include <stdio.h
```

```
#include <math.h
struct point
        {
          float x;
          float y;
        };

float distance (struct point x1 y1, struct point x2y2);
void main (void)
                {
                  struct point X1Y1, X2Y2;
                  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", &X1YI.x, &X1Y1.y);
                  printf ("Enter X and Y co-ordinates of second point: ");
                  scanf ("%f %f", &X2Y2.x. &X2Y2.y);
                  printf ("Distance between (%f, %f) and (%f, %f) is: %f.\n",
                  X1Y1.x, X1Y1.y, X2Y2.x. X2Y2.y. distance (&X1Y1, &X2Y2));
                }

float distance (struct point * x1y1, struct point * x2y2)

        {
                return (sqrt ((x1y1->x - x2y2 ->x) * (x1y1->x - x2y2->x) +
                        (x1y1-> y - x2y2-> y) * (x1y1-> y - x2y2->y))) ;
        }
```

Check Your Progress 4

Question 1: Write a C language program to print a file backwards.

Question 2: Write a C language program to maintain a personal telephone directory. Your program should support the following function: 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.

Question 3: Write a C language program to record all, and seek any, saving account transactions that can be listed in a passbook. Your program should be able to state the current balance, inclusive of the interest due in the previous accounting periods, and should display the transactions that took place between two specified dates.

Question 4: Write a C language program to implement an information management system to serve the billing functions of a school. Your program should be fully menu driven, supporting the following activities:

1. Add a new student: this activity should assign the student a unique ID number and should record all the requisite information in the master file of student records

2. Locate a student in the master file: given a student's ID, your program should display all relevant data about the student.

3. Delete a student from the master file: if the student leaves the school, her record should be archived in an archive file and deleted from the master file.

10.7 UNIONS

A union is a C variable typed by ft keyword union dud 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 [31;
                     int * ptr.
                    }

box_l,box_2,box_3;
```

The 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 number of a union is stored at the same base address. So a union is comparable to Pascal's variant records. 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_l.

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 unions. 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 accomodate the three doubles of the array delta []. The unions box_1, box_2 and box_3 are assigned a char, an 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 references 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 * par;
          } box_l,box_2,box_3,*x;

void func (union hold_all *p, int q, double r);
void main (void)
```

```
{
 printf ("The size of the union hold_afl is: %d\n", sizeof (union hold_all));
 printf ("The size of an instance of it, box_1 is: %d\n", sizeof box_1);
 box_l.alpha = 'x';
 box_2.beta = 3;
 box_3.delta [0] = 3.14;
 printf ("Can func () change box_l.alpha, which now is: %c?\n", box_1.alpha);
 printf ("Let\'s see ... sending control to func 0...\n");
 func (&box_1 box_2.beta. box_3.delta[0]);
 printf ("Yes it can: box_l.alpha after func () is: %c\n". box_1.alpha);
}

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
```

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 structure.

C has six bitwise operators, of which five are binary and one unary. They may be applied to any signed 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 Is. Then %x will have. all its bits set to Os. 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 < 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. 00100111001011 10 binary. then x & y is produced by ANDing the corresponding bits of x and y, as follows:

```
x       001010011100101 octal 12345
y       0010011100101110octal 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   001 1001 1 1 100101 1. i.e. octal 31713
```

X< < 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 Is, 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 */
#include <stdio.h
void main (void)
        {
          int W = -1;
    /*contd */

                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\n", w, w << 2, w 2);
        }
                        /* Program 10.17: Output: */
```

```
w = -1, one's complement = ~w = 0
x = 12345
y = 23456
z = x & y = 2044
x = 12345
y = 23456
z = x | y = 33757
x = 12345
```

y = 23456
z = x ^ y = 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 w : 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 assignment:

```
psw.z = 5;
```

sets 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 betwen the fields x and y:

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

An unnamed field of width () 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.

Check Your Progress 5

Question 1: Write a C language program to count the number of bits in your computer's word.

Question 2: Write a C program to implement right or left rotation of bits within a word a bits (where n is a positive integer less than the wordsize) are pushed out to the right or left, caught as they fall. and pushed in froin the left or riglit respectively.

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.

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 operators.

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 bit wise operators of C: the second uses the approach of bit fields. which are groups of contiguous bits within a word, and which can be assigned small integer values. Bit fields are accessed like structure members.