UNIT-1

C PROGRAMMING FUNDAMENTALS

CONTROL STATEMENTS

In C language the control statements are classified into two types. They are:

1. Conditional Statements
2. Unconditional Statements

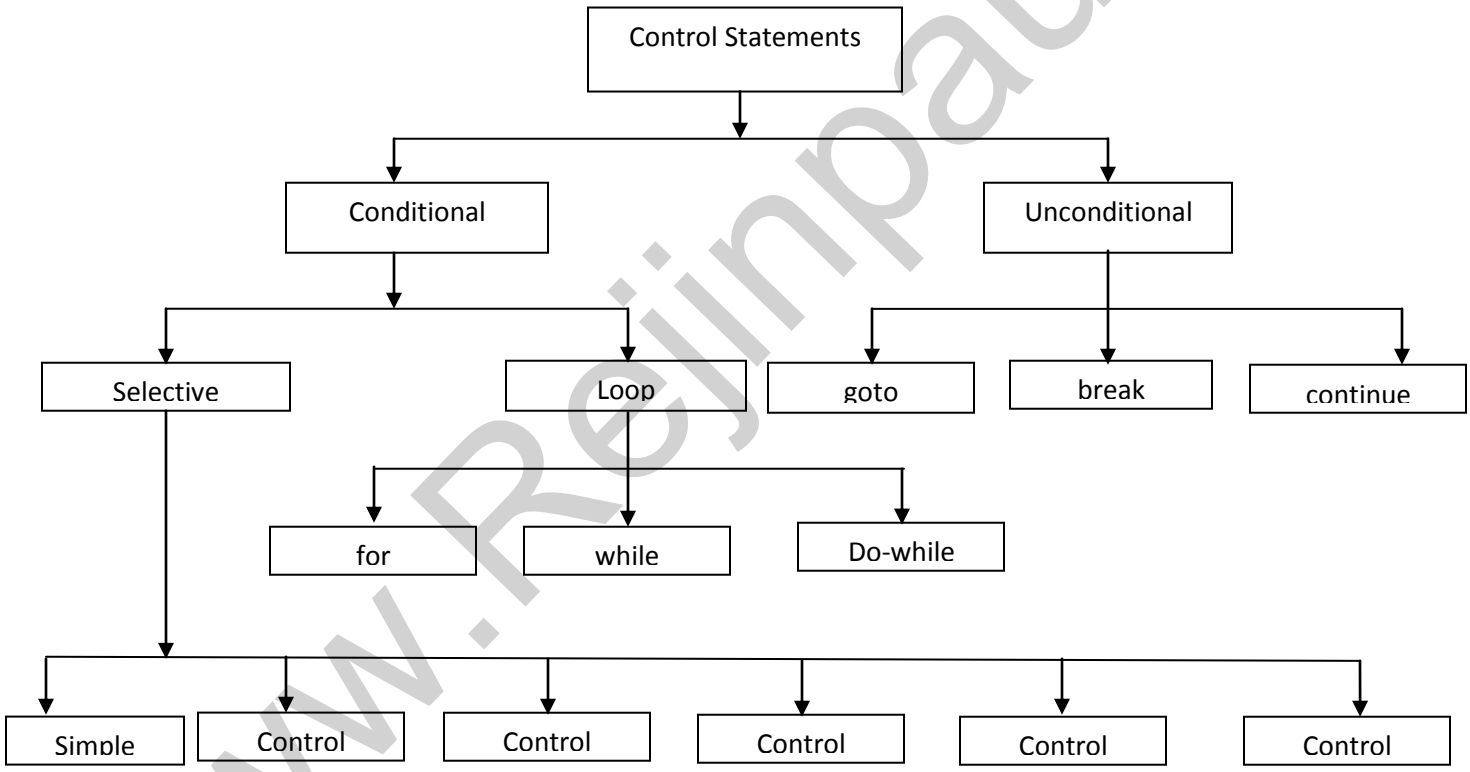


Figure: Control Statements Classification

In conditional statements, the flow of execution may be transferred from one part to another part based on the output of the conditional test carried out.

The Conditional statements are further classified in to selective and loop constructs.

The Unconditional construct is further classified into goto, break and continue statements.

Conditional Statements

Simple if statement:

Simple if statement is used to execute some statements for a particular conditions the general form of if statement is

```
if (condition)
{
statement –1;
}
statement-2;
```

Where condition is a relational, or logical expressions. the conditions must be placed in parenthesis statement-1 can be either simple or compound statement the value of condition is evaluated first the value may be true or false if the condition is true the statement –1 is executed and then the control is transferred to statement2 If the condition is false the control is transferred directly to the statement2 without executing the statement1.

Example

```
if (category==sports)
{
marks=marks+bonus marks;
}
printf(“%d”,marks);
```

if else statement

if...else statement is used to execute one group of statements if the conditions is true the other group of statements are executed when the condition is false

The general form is:

```
if (condition)
{
statement-1;
}
else
{
statement-2;
}
statement3;
```

If the condition is true the statement –1 is executed and then the control is transferred to statement3 if the condition is false the control is transferred directly to the statement2 and then statement3.

Eg:

```
If ( a>b)
{
printf("A is Big");
}
else
{
Printf(" B is Big");
}
```

Nested if -else statement:

A nested if statement is an if statement which is within another if block or else block. If an if..else is contained completely within another construct, then it is called nesting of if's.

Syntax:

```
if(condition-1)
{
  if(condition-2)
    {
      statement-1;
    }
  else
    {
      statement-2;
    }
else
    {
      ststement-3;
    }
}
 statement-4;
```

if the condition –1 is false, the statement-3 will be executed. Otherwise it continues to test condition-2. if the condition-2 is true the statement-1 will be executed; otherwise the statement-2 will be executed and then the control is transferred to statement-4.

Example

```
if(sex is female)
{
  if (balance>5000)
    {
      bonus=0.05*balance;
    }
 else
    {
      bonus=0.02*balance;
    }
else
    {
       bonus=0.02*balance;
    }
```

The elseif ladder:

There is another way of putting ifs together when multipath decisions are involved. The conditions are evaluated from top to bottom.

The general format of elseif ladder is

```
if(condition-1)
    statement-1;
elseif(condition-2)
      statement-2;
  .
```

```
.
elseif(condition-n}
    statement-n;
else
    default statement;
```

When a true condition is evaluated, the statement associated with it is executed and the rest of the ladder is omitted. if none of the condition is true then default statement is executed. if the default statement is not present, no action takes place when all conditions are false

Example:

```
.....
if(marks>74)
    printf("first class with distinction");
elseif(marks>59)
    printf("first class");
elseif(marks>39)
    printf("second class");
else
    printf("fail");
```

Switch statement:

Switch statement is a multi-branch decision statement. Switch statement is used to execute a group of statements. These groups of statements are selected from many groups. The selection is based upon the value of the expression. This expression is included within the switch statement.

The general form of the switch statement is:

```
switch(expression)
 {
```

```
        case constant1;
            statement-1;
            break;
         case constant-2;
            statement-2;
            break;
                    :
                    :
                    case constant-N:
                    statement-N;
                    break;
          default;
              default statement;
                    break;
       }
  statement-x;
```

Where switch, case and default are the keywords. The expression following switch must be enclosed in parenthesis and the body of the switch must be enclosed within curly braces. Expression may be variable or integer expression. Case labels can be only integer or character constants. Case labels do not contain any variables names. Case labels must all be different. Case labels end with a semicolon.

Rules for forming switch statement:

(1) The order of the case may be in any order. it is not necessary that order may be in as ascending or descending.

(2) If multiple statements are to be executed in each case there is no need to enclose within a pair of braces

(3) If default statement is not present, then the program continues with the next instruction.

Example:

```
main()
{
  char ch;
   int a,b,c=0;
   scanf("%d%d",&a,&b);
   scanf("%c",&ch); or ch=getchar();
   switch ch()
   {
      case '+':c=a+b;
                break;
       case '-':c=a-b;
                 break;
        case '*':c=a*b;
               break;
       case'/':c=a/b;
               break;
        default:printf("the operator entered is invalid")
       }
  printf("the result is %d",c);
}
```

LOOPING STATEMENTS

Loops are used to execute a same set of instructions for much times.the number of times a loop executed is given by some condition. The loop structures available in c are

1. The while loop

2. The do..while

3. The for loop

While Loop

It is one type of looping statement. this is entry control statement ,that mean first check the condition then execute the statement is called entry control statement

The general format of the while loop is

```
while (condition)
{
    body of the loop
}
```

Where body of the loop is either a single statement or a block of statements. The condition may be any expression. The loop repeats while the condition is true. When the condition is false the control will be transferred out of the loop.

example:

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

```
while (i<=100)
 {
   printf("%d",i);
   i++;
 }
}
```

do..while loop:

This statement is exit control statement. Therefore the body of the loop is always executed at least once; evenif the condition is false initially. The general form of the do..while loop is

```
do
{
   body of the loop
}
while(condition);
```

The do..while loop repeats until the condition becomes false. In the do..while the body of the loop is executed first, then the condition is checked . When the condition becomes false, control is transferred to the statement after the loop.

for loop:

for loop is used to repeat a statement or a block of statements for a known number of times. The general format of the for loops

```
for (initialization; condition; increment)
 {
      body of the loop
 }
```

Initialization is an assignment statement, that is used to set the loop control variable. The condition is a relational or logical expression which determines when to end the loop. The increment is a unary or an assignment expression.

Execution of the for loop:

* Initialization of control variable is done first.

* The value of the control variable is tested using condition test. If the value is true, the body of the loop will be executed; otherwise the loop determinate.

Example:

```
main()
{
   int i;
   for (i=1;i<=10;i++)
    printf("%d",i);
}
```

Jumps in Loop:

Loops perform a set of operations repeatedly until the control variables fails to satisfy the test conditions. The number of times the loop is repeated is decided in advance and the test conditions is returned to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain conditions occurs.

Syntax:

```
While( condition-1)
{
statements1;
if (condition-2)
break;
```

```
statements2;
} ;
```

When the condition2 is true means, control will out of the loop.

Example:

```
Int t=1;
While ( t <= 10)
  {
  sum = sum + t;
    if ( sum = = 5)
     break;
     t=t+1;
    }
```

FUNCTION

DEFINITION OF FUNCTION

A function in C can perform a particular task, and supports the concept of modular programming design techniques. We have already been exposed to functions. The main body of a C program, identified by the keyword *main* and enclosed by the left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.

FUNCTION DECLARATION

Functions have a basic structure. Their format is

```
Returntype functionname( argument lists);
```

Where return type refers to the datatype of the data returned by the functions, and the function name refers the user defined function name.

Eg: Int add(int x, int y);

Category of functions

A functions , depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions with no arguments and return values.

Functions with no arguments and no return values

When functions has no arguments , it does not receive any data from the calling function. Similarly , when it does not return a value, the calling function does not receive any data from the called function. In effect , there is no data transfer between the calling function and the called function.

Eg:

```
Void main()
{
void add();
add()
}
void add()
{ printf( " no arg and no value\n");
}
```

Functions with arguments and no return values

Calling function send arguments to called functions, but called function does not return any value to calling program.

Eg:

```
Void main()
{
```

```
void  add(a,b);
int a,b;
scanf("%d%d",&a,&b);
add(a,b)
}
void add(int x, int y)
{
int c;
c= x+y;
printf( " %d",c);
}
```

Functions with arguments and one return values

Calling function send arguments to called functions, as well as called function return a value to calling program.

```
Eg:   Void  main()
      {
      int  add(a,b);
      int a,b,c;
      scanf("%d%d",&a,&b);
      c=add(a,b);
      printf("%d",c);
      }
      int add(int x, int y)
      {
      int c;
      c= x+y;
      return(c);
```

```
}
```

Functions with no arguments but return values

Calling function does not send any arguments to called functions, but called function return a value to calling program.

```
Eg:   Void  main()
      {
      void  add();
      int  c;
      c=add();
      printf("%d",c);
      }
      int add()
      {
      int a,b,c;
      scanf("%d%d",&a,&b);
      c= x+y;
      return(c);
      }
```

Functions That Return Multiple Values

Called function consists of more than one return statements but execute any one return statement is possible.

```
Void main()
{
int   add (int a, int b, int c);
int a,b,c,d;
Scanf("%d%d%d",&a,&b,&c);
```

```
 D=add(a,b,c);
Printf("%d",d);
}
int add (int x, int y, int z)
  {
    if ((x>y)&&(x>z))
      return(x);
 else if (y > z)
       return(y);
   else
      return(z);
}
```

Nesting of Functions

C permits nesting of functions freely. Main can call function1, which calls functions2, which calls function3 etc…. and so on. Consider the following program.

```
Void main()
{
void add();
add()
}
void add ()
{
printf(" PROGRAMMING");
add1();
}
void add1()
{
```

printf(“LANGUAGE”);

}

PASSING ARGUMENT TO FUNCTION :

1. In C Programming we have different ways of parameter passing schemes such as Call by Value and Call by Reference.
2. Function is good programming style in which we can write reusable code that can be called whenever require.
3. Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called **argument**.

Two Ways of Passing Argument to Function in C Language:

1. Call by value
2. Call by Reference

3) CALL BY VALUE:

```
#include<stdio.h>

void interchange(int number1,int number2)
{
   int temp;
   temp = number1;
   number1 = number2;
   number2 = temp;
}

int main() {

   int num1=50,num2=70;
   interchange(num1,num2);

   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);

   return(0);
```

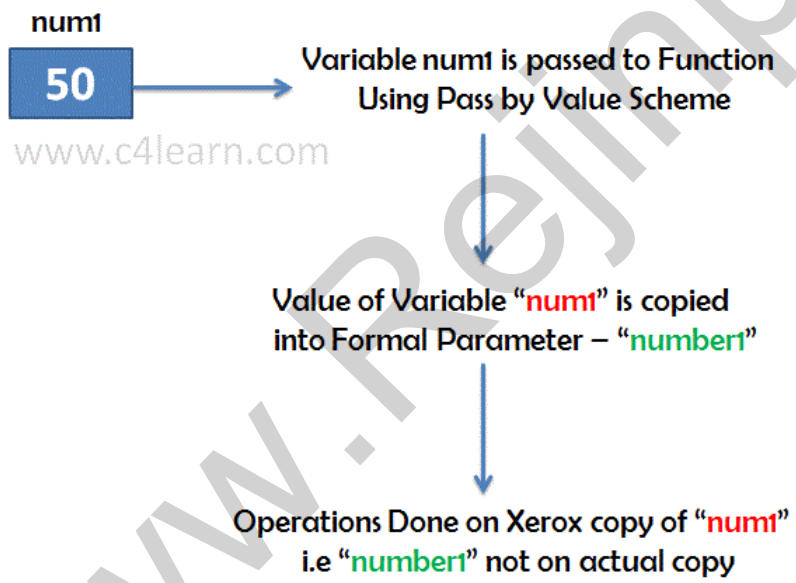
```
}
```

Output :

```
Number 1 : 50
Number 2 : 70
```

Explanation: Call by Value

1. While Passing Parameters using call by value, **Xerox copy of original parameter is created** and passed to the called function.
2. Any update made inside method will not affect the **original value of variable in calling function**.
3. In the above example num1 and num2 are the original values and Xerox copy of these values is passed to the function and these values are copied into number1, number2 variable of sum function respectively.
4. As their scope is limited to only function so they **cannot alter the values inside main function**.



4) CALL BY REFERENCE/POINTER/ADDRESS:

```
#include<stdio.h>

void interchange(int *num1,int *num2)
```

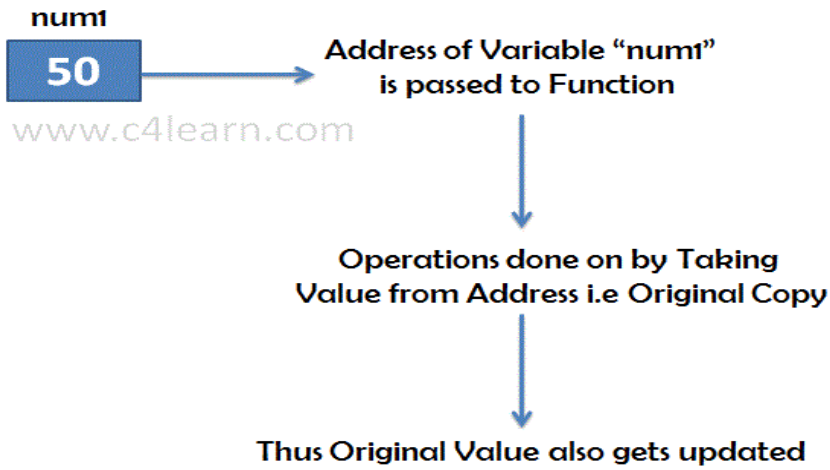
```
{
   int temp;
   temp  = *num1;
   *num1 = *num2;
   *num2 = *num1;
}

int main() {

   int num1=50,num2=70;
   interchange(&num1,&num2);

   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);

   return(0);
}
```

Output :

```
Number 1 : 70
Number 2 : 50
```

Explanation: Call by Address

1. While passing parameter using call by address scheme, we are **passing the actual address of the variable** to the called function.
2. Any updates made inside the called function **will modify the original copy** since we are directly modifying the content of the exact memory location.



Summary of Call By Value and Call By Reference

Point	Call by Value	Call by Reference
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function

ARRAY

Arrays are collection of similar items (i.e. ints, floats, chars) whose memory is allocated in a contiguous block of memory.

- Pointers and arrays have a special relationship. To reference memory locations, arrays use pointers. Therefore, most of the times, array and pointer references can be used interchangeably.

INITIALIZATION OF ARRAYS:

We can initialize the element of arrays in the same. The same Way as the ordinary variables when they are declared. The general form of initialization of arrays is

Type arrayname [size] = {list of values};

The values in the list are separated by commas for example the statement.

int member[5]={1,2,3,4,5};

In the example, the size of the array is 5 and the values 1,2,3,4,5 will be assigned to the variable number[0],number[1],number[2],number[3] and number[4] respectively.

DECLARING ARRAYS

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

Type arrayName [arraySize];

This is called a single-dimensional array. The array Size must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

Now balance is a variable array which is sufficient to hold up to 10 double numbers.

ONE DIMENSIONAL ARRAY

The array which is used to represent and store data in a linear form is called as 'single or one dimensional array.'

Syntax:

<data-type> <array_name> [size];

Example:

```
int a[3] = {2, 3, 5};
char ch[20] = "TechnoExam" ;
float stax[3] = {5003.23, 1940.32, 123.20} ;
```

Total Size (in Bytes):

total size = length of array * size of data type

In above example, a is an array of type integer which has storage size of 3 elements. The total size would be 3 * 2 = 6 bytes.

*** MEMORY ALLOCATION :**

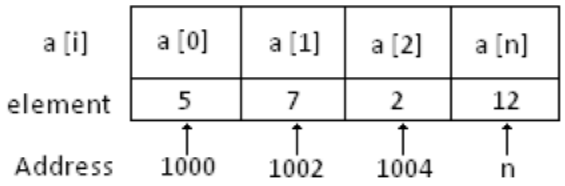


Fig : Memory allocation for one dimensional array

Program:

```
/* Program to demonstrate one dimensional array.

#include <stdio.h>
#include <conio.h>
void main()
{
        int a[3], i;;
        clrscr();
        printf("\n\t Enter three numbers : ");
        for(i=0; i<3; i++)
        {
                scanf("%d", &a[i]);  // read array
        }
        printf("\n\n\t Numbers are : ");
        for(i=0; i<3; i++)
        {
                printf("\t %d", a[i]);  // print array
        }
        getch();
```

}

Output:

```
Enter three numbers : 9 4 6

Numbers are :    9       4       6_
```

Features:

- Array size should be positive number only.
- String array always terminates with null character ('\0').
- Array elements are countered from 0 to n-1.
- Useful for multiple reading of elements (numbers).

Disadvantages:

- There is no easy method to initialize large number of array elements.
- It is difficult to initialize selected elements.

Two Dimensional Array:

The array which is used to represent and store data in a tabular form is called as 'two dimensional array.' Such type of array specially used to represent data in a matrix form.
The following syntax is used to represent two dimensional array.

Syntax:

<data-type> <array_nm> [row_subscript][column-subscript];

Example:

```
int a[3][3];
```

In above example, a is an array of type integer which has storage size of 3 * 3 matrix. The total size would be 3 * 3 * 2 = 18 bytes.
It is also called as 'multidimensional array.'

*** MEMORY ALLOCATION:**

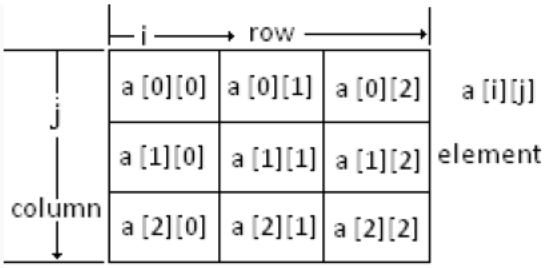


Fig : Memory allocation for two dimensional array

Program:

```
/* Program to demonstrate two dimensional array.

#include <stdio.h>
#include <conio.h>
void main()
{
        int a[3][3], i, j;
        clrscr();
        printf("\n\t Enter matrix of 3*3 : ");
        for(i=0; i<3; i++)
        {
                for(j=0; j<3; j++)
                {
                scanf("%d",&a[i][j]);  //read 3*3 array
                }
        }
        printf("\n\t Matrix is : \n");
        for(i=0; i<3; i++)
        {
                for(j=0; j<3; j++)
                {
                printf("\t %d",a[i][j]);  //print 3*3 array
                }
          printf("\n");
        }
        getch();
```

}

Output :

```
Enter matrix of 3*3 : 3 4 5 6 7 2 1 2 3

Matrix is :
3       4       5
6       7       2
1       2       3_
```

Limitations of two dimensional array :

- We cannot delete any element from an array.
- If we don't know that how many elements have to be stored in a memory in advance, then there will be memory wastage if large array size is specified.

PREPROCESSOR DIRECTIVES

Preprocessor is a macro processor program that processes the code before it passes through the compiler. It operates under the control of preprocessor command lines and directives. Preprocessor directives are placed in the source program before the main line before the source code passes through the compiler it is examined by the preprocessor for any preprocessor directives. If there is any appropriate actions are taken then the source program is handed over to the compiler.

It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

Preprocessor directives follow the special syntax rules and begin with the symbol #bin column1 and do not require any semicolon at the end. A set of commonly used preprocessor directives

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro

#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

The preprocessor directives can be divided into three categories:

1. Macro substitution division
2. File inclusion division
3. Compiler control division

POINTERS

Introduction:

A pointer is a variable that is used to store the address of another variable. Since the pointer is also another variable its value is also stored in the memory in another location.

Advantages of pointers:

- Pointers are used to increase the speed of the execution.
- Pointers are used to reduce the length and complexity of program.
- A pointer is used to access a variable that is defined outside the function.
- Storage space is saved when using pointer array to character strings.

Understanding pointers:

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number; this location will have its own address number.

Int quantity = 177

Quantity ------- variable
179 ------ value
5000 ------- address

During execution of the program, the system always associates the name quantity with the address 5000. We may have access to the value 179 by using either the name quantity or the address 5000.

Accessing the Address of Pointer Variable:

The content of any pointer variable can be accessed with the help of "*" operator. if "p" is an pointer variable then *p is used to access the content of the pointer variable "p".

for example the statement t=*p is used to assign the content of the pointer variable p.

```
int  * p;
int a=30;
p=&a;
```

Declaring a pointer variable:

For defining a pointer variable '*' symbol is used. a pointer variable must be declared with * preceding the variable name. The general structure for declaring a pointer variable is

```
data_type*ptr_name;
```

for example the statement int*p declares p as a pointer variable that points to an integer data type.

Example:

```
main()
{
  int*p;
```

```
int i=30;
p=&i;
}
```

p 30

INITIALIZATION OF POINTER VARIABLES:

Pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

Data type *pt_name

1. The asterisk (*) tells that the variable pt_name .
2. pt_name needs a memory location.
3. pt-name pointes to a variable of type data type.

Example:

```
Int *p;
```

Example 2:

```
float a,b;
int  x, *p;
p= &a;
b= *p;
```

Accessing a variable through its pointer:

A pointer is a variable that contains an address. Remember, this address is a location of another variable in memory. The value of pointers "points" to a variable, which may be accessed indirectly with the special pointer operators * and & .

Example:

```
main()
{
  int*p;
  int i=30;
  p=&i;
 }
```

Chain of Pointers:

Any pointer variable points to another pointer variable to access the content of other memory location of variable is called chain of pointers.

Ptr -> Ptr -> memory

Pointer expression:

Pointer variables can also be used in expressions. Assume a and c are pointer variables. The following is an example of pointer expression.

```
c=*a+1;
```

The above expression adds the value 1 to the contents of address stored in 'a' and assign the new value to c. the following are valid expression

```
x=*a**b;
b=b+10;
```

Pointer Increments and scale factors:

We are able to increment the address of the pointer variable. We have seen the pointers can incremented like

```
P1 =  P2 + 2;
P1 =  P1 + 1;
```

And so on. An expression like

```
P1++;
```

Will cause the pointer P1 to point to the next value of its type .

POINTERS ARITHMETIC

C pointer is an address which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

Now after the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location. If **ptr** points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = var;
  for ( i = 0; i < MAX; i++)
  {
```

```
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = &var[MAX-1];
  for ( i = MAX; i > 0; i--)
  {
```

```
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the previous location */
    ptr--;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have address of the first element in pointer */
```

```
   ptr = var;
   i = 0;
   while ( ptr <= &var[MAX - 1] )
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* point to the previous location */
      ptr++;
      i++;
   }
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

POINTERS AND ARRAYS:

Pointers and arrays have got close relationship. In fact array name itself is a pointer. some times pointer2 and array name can be inter changeably used.

The array name always points to the five element of the array for example and array int a[5].

Element	a[0]	a[1]	a[2]	a[3]	a[4]
value	1	2	3	4	5
address	1000	1002	1004	1006	1008

for example

```
int a[5]

int i;

A(a+2)=30.
```

here x(a+2) actually refers to a[2].

Pointers and character string:

All operations on strings were performed by indexing the character array. However , most string operations in C are usually performed by pointers to the array and pointer arithmetic because pointer are faster and easier to use.

```
Void main()

{

    char s[80];

   char *p1;

   do{

   p1 = s;

   gets(s);

   while (*p1) printf("%d",*p++);

/* print the decimal equivalent of each character */

}

   while(! Strcmp(s, "done");

}
```

Array of pointers:

Pointer may be arrayed just like any other data type. The declaration for an int pointer array of size 10 is

```
Int *x[10];
```

To assign the address of an integer variable called var to the third element of the pointer array, you would write

```
Int var;
```

X[2]= & var;

The only values that the array elements may hold are the addresses of integer variables.

Pointers as function arguments:

When we pass addresses to function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variable is known as call by reference. (You know, the process of passing the actual value of variables is known as call by value .) The function which is called by 'reference' can change the value of the variable used in the call. Consider the following code:

```
Main ()
  {
  int x;
  x = 20;
  change(&x);
  printf("%d \n",x);
  }
  change(p);
  int *p;
  {
  *p = *p + 10;
  }
```

Function Returning pointers

Calling function passes address of a variable to a function and returns back the value to main program.

Pointers and functions:

A function also has an address location in memory. it is therefore possible to declare a pointer to a function which can be used as an argument in another function. A pointer to a function is described as follows.

type(* pointer-name)();1

for example, in the statements,

int * sqr(), square();

sqr=square;

sqr is pointer to a function and sqr points to the function square. To call the function square, the pointer sqr is with the list of parenthesis. That is

(*sqr)(a.b)

is equal to

square(a,b)

Pointers and structures

We know that the name of an array stands for the address of its zeros element. The same thing is true of the names of arrays of structure variable. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeros element. Consider the following declaration:

Struct inventory

{

char name[30];

int number;

float price;

}

Product [2],*ptr;

This statement declare **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to dat object of the type **struct inventory.** The assignment

Ptr = product;

Would assign the address of the zeros element of product to ptr. That is, the pointer ptr will now point to product [0]. Its members can be accessed using the following notation.

Ptr -> name

Ptr -> number

UNIT-2

C PROGRAMMING ADVANCED FEATURES

INTRODUCTION TO STRUCTURE IN C

1. As we know that **Array is collection of the elements of same type**, but many time we have to store the elements of the different data types.
2. Suppose Student record is to be stored, then for storing the record we have to group together all the information such as Roll name, Percent which may be of different data types.
3. Ideally Structure is **collection of different variables under single name**.
4. Basically Structure is for storing the complicated data.
5. A structure is a convenient way of **grouping several pieces of related information together**.

Structure Initialization:

The members of the structure can be initialized to constant valued by enclosing the values to be assigned within the braces after the structure definition

```
struct date{
int date;
int month;
int year;
} republic= {26, 1, 1950};
```

Initializes the member variable data, month, year of republic to 26, 1, 1950 respectively.

2) DEFINITION OF STRUCTURE IN C:

Structure is composition of the different variables of different data types, grouped under same name.

```
typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
  } student;
```

Some Important Definitions of Structures:

1. Each member declared in Structure is called **member.**

```
char name[64];
char course[128];
int age;
int year;
```

are some examples of members.

2. Name given to structure is called as **tag**

```
student
```

3. Structure **member** may be of **different data type** including **user defined data-type** also

```
typedef struct {
        char name[64];
        char course[128];
        book b1;
        int year;
  } student;
```

Here book is user defined data type.

3) STRUCTURE DECLARATION

In C we can group some of the user defined or primitive data types together and form another compact way of storing complicated information is called as Structure. Let us see how to declare structure in c programming language -

Syntax Of Structure in C Programming:

```
struct tag
{
  data_type1 member1;
  data_type2 member2;
  data_type3 member3;
};
```

Structure Alternate Syntax :

```
struct <structure_name>
{
   structure_Element1;
```

```
    structure_Element2;
    structure_Element3;
    ...
    ...
};
```

Some Important Points Regarding Structure in C Programming:

1. **Struct** keyword is used to declare structure.
2. **Members of structure** are enclosed within opening and closing braces.
3. **Declaration** of Structure reserves **no space**.
4. It is nothing but the "**Template / Map / Shape"** of the structure.
5. Memory is created, very first time when the **variable is created** /**Instance** is created.

Different Ways of Declaring Structure Variable:

Way 1 : Immediately after Structure Template

```
struct date
{
    int date;
    char month[20];
    int year;
}today;

// 'today' is name of Structure variable
```

Way 2 : Declare Variables using struct Keyword

```
struct date
{
    int date;
    char month[20];
    int year;
};

struct date today;
```

Where "**date**" is name of structure and "**today**" is name of variable.

Way 3: Declaring Multiple Structure Variables

```
struct Book
{
   int pages;
   char name[20];
   int year;

} book1, book2, book3;
```

We can declare multiple variables separated by comma directly after closing curly.

Arrays of Structures:

Arrays of structures are commonly used in a large number of similar records required to be processed together. If there were one hundred employees in an organization , this data can be organized in an array of structures. consider the following example:

```
struct  emp-data

{

char  name[30];

int age;

};
```

Then the array of structure used is

```
struct emp-data  employee[100];
```

the first record is referred as employee[0], second record is referred by employee[1], and so on.

```
employee [0].name  = “raja1”

employee[0].age  = 25

employee[1].name  = “raja3”

employee[1].age  = 35
```

hence array of structures is used to manage large number of records easily.

Array within structures:

A structure can have an array within it.

Char name[30];

Like this is a structure can have no. of arrays as members.

Eg:

```
Struct  student
{
char  name[25];
int  sem;
char  branch[10];
int mark[3];
}
```

In the above example there are four members. Out of these four fields, name, branch and mark are arrays.

4. STRUCTURE WITHIN STRUCTURE: NESTED STRUCTURE

A structure within structure means nesting of structures. a structure can be declared within another structure. Consider the following emp-data structure. In this case the structure 'date' may be placed inside the structure.

```
struct date{
int date,month,year;
};
struct emp-data
{
char name[30];
struct date dob;
};
```

Example

```
main()
{
struct emp-data raja={"raja",{14,8,90}};
printf("%s",raja.name);
printf("%d%d%d",raja.dob.date,raja.dob.month, raja.dob.year);
}
```

Structure and functions

This method is to pass each member of the structure as an actual arguments of the function call. The actual arguments are then treated independently ordinary variables. This is the most elementary methods and becomes inefficient when the structure size is large.

Syntax:

Function name (Structure variable name)

UNION

Unions are quite similar to the structures in C. Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in **union** where keyword used in defining structure was **struct.**

```
union car{
char name[50];
int price;
};
```

Union variables can be created in similar manner as structure variable.

```
union car{
 char name[50];
 int price;
}c1, c2, *c3;
```

OR;

```
union car{
  char name[50];
  int price;
};
-------Inside Function-----------
union car c1, c2, *c3;
```

In both cases, union variables *c1*, *c2* and union pointer variable *c3* of type **`union car`** is created.

Accessing members of an union

The member of unions can be accessed in similar manner as that structure. Suppose, we you want to access price for union variable *c1* in above example, it can be accessed as `c1.price`. If you want to access price for union pointer variable *c3*, it can be accessed as `(*c3).price` or as `c3->price`.

Union:

Union is another compound data type like structure. Union is used to minimize memory utilization. In structure each member has the separate storage location. but in union all the members share the common place of memory. so a union can handle only one member at a time.

Declaration of Union:

The general format for declaring union is,

```
union  tagname
{
datatype  member1;
datatype  member2;
..
...
datatype member N;
};
```

Where union is the keyword. Tag name is any user defined data types.

Example:

```
union  personal
{
char  name[25];
int age;
float height;
};
```

In the above example 'union bio' has 3 members first member is a character array 'name' having to 10 character (10 bytes)second member is an integer ' age' that requires 2 bytes third member is a float as four bytes, all the three members are different data types.

Here all these 3 members are allocated with common memory . They all share the same memory. the compiler allocates a piece of storage that is large enough to hold the largest type in the union.

In case of structure it will be occupies a separate memory of each data types.

Advantages:

1. It is used to minimize memory utilization.
2. it is used to convert data from one type to another type.

Example:

```
union test
{
int  sum;
float average;
};
```

PROGRAMS USING STRUCTURES AND UNIONS

C Program to Store Information of Single Variable

```
#include <stdio.h>
```

```
struct student{
   char name[50];
   int roll;
   float marks;
};
int main(){
   struct student s;
   printf("Enter information of students:\n\n");
   printf("Enter name: ");
   scanf("%s",s.name);
   printf("Enter roll number: ");
   scanf("%d",&s.roll);
   printf("Enter marks: ");
   scanf("%f",&s.marks);
   printf("\nDisplaying Information\n");
   printf("Name: %s\n",s.name);
   printf("Roll: %d\n",s.roll);
   printf("Marks: %.2f\n",s.marks);
   return 0;
}
```

Output

```
Enter information of students:

Enter name: Adele
Enter roll number: 21
Enter marks: 334.5

Displaying Information
Name: Adele
Roll: 21
Marks: 334.50
```

-

Source Code to Store Information of 10 students Using Structure

```
#include <stdio.h>
struct student{
   char name[50];
   int roll;
   float marks;
};
int main(){
   struct student s[10];
   int i;
   printf("Enter information of students:\n");
   for(i=0;i<10;++i)
   {
      s[i].roll=i+1;
      printf("\nFor roll number %d\n",s[i].roll);
      printf("Enter name: ");
      scanf("%s",s[i].name);
      printf("Enter marks: ");
      scanf("%f",&s[i].marks);
      printf("\n");
   }
   printf("Displaying information of students:\n\n");
   for(i=0;i<10;++i)
   {
```

```
    printf("\nInformation for roll number %d:\n",i+1);
    printf("Name: ");
    puts(s[i].name);
    printf("Marks: %.1f",s[i].marks);
  }
  return 0;
}
```

Output

```
Enter information of students:
For roll number 1
Enter name: Tom
Enter marks: 98

For roll number 2
Enter name: Jerry
Enter marks: 89
.
.
.
Displaying information of students:

Information for roll number 1:
Name: Tom
Marks: 98
```

Source Code Demonstrate the Dynamic Memory Allocation for Structure

```
#include <stdio.h>
#include<stdlib.h>
struct name {
  int a;
  char c[30];
};
int main(){
  struct name *ptr;
  int i,n;
  printf("Enter n: ");
  scanf("%d",&n);

/* allocates the memory for n structures with pointer ptr pointing to the base address. */
  ptr=(struct name*)malloc(n*sizeof(struct name));
  for(i=0;i<n;++i){
     printf("Enter string and integer respectively:\n");
     scanf("%s%d",&(ptr+i)->c, &(ptr+i)->a);
  }
  printf("Displaying Infromation:\n");
  for(i=0;i<n;++i)
    printf("%s\t%d\t\n",(ptr+i)->c,(ptr+i)->a);
  return 0;
}
```

Output

Enter n: 2

Enter string and integer respectively:

Programming

22

Enter string, integer and floating number respectively:

Structure

33

Displaying Information:

Programming 22

Structure 33

Program to demonstrate UNION

```
#include <stdio.h>
Union  job {        //defining a union
  char name[32];
  float salary;
  int worker_no;
}u;
Struct  job1 {
  char name[32];
  float salary;
  int worker_no;
}s;
int main(){
```

```
  printf("size of union = %d",sizeof(u));

  printf("\nsize of structure = %d", sizeof(s));

  return 0;

}
```

Output

Size of union = 32

Size of structure = 40

There is difference in memory allocation between union and structure as suggested in above example. The amount of memory required to store a structure variables is the sum of memory size of all members.

Fig: Memory allocation in case of structure

But, the memory required to store a union variable is the memory required for largest element of an union.

Fig: Memory allocation in case of union

File Handling & File Manipulations in C Language

In this section, we will discuss about files which are very important for storing information permanently. We store information in files for many purposes, like data processing by our programs.

What is a File?

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files. These two classes of files will be discussed in the following sections.

ASCII Text files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of

text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signaled the intention to process a text file.

Binary files

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files. They a generally processed using read and writes operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

Creating a file and output some data

In order to create files we have to learn about File I/O i.e. how to write data into a file and how to read data from a file. We will start this section with an example of writing data to a file. We begin as before with the include statement for stdio.h, then define some variables for use in the example including a rather strange looking new type.

```
/* Program to create a file and write some data the file */
#include <stdio.h>
#include <stdio.h>
main( )
{
    FILE *fp;
    char stuff[25];
    int index;
    fp = fopen("TENLINES.TXT","w"); /* open for writing */
    strcpy(stuff,"This is an example line.");
    for (index = 1; index <= 10; index++)
            fprintf(fp,"%s Line number %d\n", stuff, index);
    fclose(fp); /* close the file before ending program */
}
```

The type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the file name is. We do this with the fopen() function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the file name first, followed by the file type.

The file name is any valid DOS file name, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a

file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you don't have a file by this name, that is good because we will create one and put some data into it. You are permitted to include a directory with the file name. The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character â€˜\â€™ must be written twice. For example, if the file is to be stored in the \PROJECTS sub directory then the file name should be entered as "\\PROJECTS\\TENLINES.TXT". The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case.

Reading (r)

When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Here is a small program that reads a file and displays its contents on screen.

```
/* Program to display the contents of a file on screen */
#include <stdio.h>
void main()
{
  FILE *fopen(), *fp;
  int c;
  fp = fopen("prog.c","r");
  c = getc(fp) ;
  while (c!= EOF)
  {
                    putchar(c);
                    c = getc(fp);
```

```
  }
  fclose(fp);
}
```

Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Here is the program to create a file and write some data into the file.

```
#include <stdio.h>
int main()
{
 FILE *fp;
 file = fopen("file.txt","w");
 /*Create a file and add text*/
 fprintf(fp,"%s","This is just an example :)"); /*writes data to the file*/
 fclose(fp); /*done!*/
 return 0;
}
```

Appending (a)

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the indicates that the file is assumed to be a text file.

Here is a program that will add text to a file which already exists and there is some text in the file.

```
#include <stdio.h>
int main()
{
   FILE *fp
   file = fopen("file.txt","a");
   fprintf(fp,"%s","This is just an example :)"); /*append some text*/
   fclose(fp);
   return 0;
}
```

Outputting to the file

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, fprintf replaces our familiar printf function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the printf statement.

Closing a file

To close a file you simply use the function fclose with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. Compile and run this program. When you run it, you will not get any output to the monitor because it doesn't generate any. After running it, look at your directory for a file named TENLINES.TXT and type it; that is where your output will be. Compare

the output with that specified in the program; they should agree! Do not erase the file named TENLINES.TXT yet; some of the other examples in this section.

Reading from a text file

Now for our first program that reads from a file. This program begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an r is used here because we want to read it.

```
#include <stdio.h>
  main( )
  {
    FILE *fp;
    char c;
    funny = fopen("TENLINES.TXT", "r");
    if (fp == NULL)
                        printf("File doesn't exist\n");
    else
{
     do
 {
      c = getc(fp); /* get one character from the file      */
        putchar(c); /* display it on the monitor      */
      }
           while (c != EOF);  /* repeat until EOF (end of file)   */
    }
   fclose(fp);
  }
```

In this program we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is

one do while loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated. At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the getc function is a character, so we can use a char variable for this purpose. There is a problem that could develop here if we happened to use an unsigned char however, because C usually returns a minus one for an EOF – which an unsigned char type variable is not capable of containing. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one in C. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent: always have a char or int type variable for use in returning an EOF. There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT.

C Program to Create File & Store Information

This C Program creates a file & store information. We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files and this program demonstrate file creation and writing data in that.

Here is source code of the C program to create a file & store information. The C program is successfully compiled and run on a Linux system. The program output is also shown below.

```
/*    C program to create a file called emp.rec and store information  about a person, in
terms of his name, age and salary.    */

#include <stdio.h>
 void main()
{
   FILE *fptr;
   char name[20];
   int age;
   float salary;
    /*  open for writing */
   fptr = fopen("emp.rec", "w");
    if (fptr == NULL)
   {
      printf("File does not exists \n");
      return;
   }
   printf("Enter the name \n");
   scanf("%s", name);
   fprintf(fptr, "Name    = %s\n", name);
   printf("Enter the age\n");
   scanf("%d", &age);
   fprintf(fptr, "Age     = %d\n", age);
   printf("Enter the salary\n");
  scanf("%f", &salary);
   fprintf(fptr, "Salary  = %.2f\n", salary);
   fclose(fptr); }
```

OUTPUT:

Enter the name

Raj

Enter the age

40

Enter the salary

4000000

Binary files

Binary files are very similar to arrays of structures, except the structures are in a disk-file rather than an array in memory. Binary files have two features that distinguish them from text files:

1. You can instantly use any structure in the file.
2. You can change the contents of a structure anywhere in the file.
3. After you have opened the binary file, you can read and write a structure or seek a specific position in the file. A file position indicator points to record 0 when the file is opened.
4. A read operation reads the structure where the file position indicator is pointing to. After reading the structure the pointer is moved to point at the next structure.
5. A write operation will write to the currently pointed-to structure. After the write operation the file position indicator is moved to point at the next structure.
6. The fseek function will move the file position indicator to the record that is requested.
7. Remember that you keep track of things, because the file position indicator can not only point at the beginning of a structure, but can also point to any byte in the file.

The fread and fwrite function takes four parameters:

- A memory address
- Number of bytes to read per block
- Number of blocks to read
- A file variable

For example:

fread(&my_record,sizeof(struct rec),1,ptr_myfile);

This fread statement says to read x bytes (size of rec) from the file ptr_myfile into memory address &my_record. Only one block is requested. Changing the one into ten will read in ten blocks of x bytes at once.

Let's look at a write example:

```
#include<stdio.h>
/* Our structure */
struct rec
{
        int x,y,z;
};
int main()
{
        int counter;
        FILE *ptr_myfile;
        struct rec my_record;
```

```
        ptr_myfile=fopen("test.bin","wb");
        if (!ptr_myfile)
        {
                printf("Unable to open file!");
                return 1;
        }
        for ( counter=1; counter <= 10; counter++)
        {
                my_record.x= counter;
                fwrite(&my_record, sizeof(struct rec), 1, ptr_myfile);
        }
        fclose(ptr_myfile);
        return 0;
}
```

In this example we declare a structure rec with the members x,y and z of the type integer. In the main function we open (fopen) a file for writing (w). Then we check if the file is open, if not, an error message is displayed and we exit the program. In the “for loop” we fill the structure member **x** with a number. Then we write the record to the file. We do this ten times, thus creating ten records. After writing the ten records, we will close the file (don’t forget this).

So now we have written to a file, let’s read from the file we have just created. Take a look at the example:

```
#include<stdio.h>
/* Our structure */
struct rec
{
        int x,y,z;
};

int main()
{
        int counter;
        FILE *ptr_myfile;
        struct rec my_record;

        ptr_myfile=fopen("test.bin","rb");
        if (!ptr_myfile)
        {
                printf("Unable to open file!");
                return 1;
        }
        for ( counter=1; counter <= 10; counter++)
        {
                fread(&my_record,sizeof(struct rec),1,ptr_myfile);
                printf("%d\n",my_record.x);
```

```
        }
        fclose(ptr_myfile);
        return 0;
}
```

The only two lines that are changed are the two lines in the “for loop”. With the fread we read-in the records (one by one). After we have read the record we print the member x (of that record).

The only thing we need to explain is the fseek option. The function fseek must be declared like this:

int fseek(FILE * stream, long int offset, int whence);

The fseek function sets the file position indicator for the stream pointed to by the stream. The new position, measured in characters from the beginning of the file, is obtained by adding offset to the position specified by whence. Three macros are declared in stdio.h called: SEEK_SET, SEEK_CUR and SEEK_END.

If the position declared by whence is SEEK_SET, then the position is the beginning of the file. The SEEK_END can be used if you want to go to the end of the file. (Using negative numbers it is possible to move from the end of the file.) If when SEEK_CUR then the position is set, x bytes, from the current position.

Let’s take a look at an example:

```
#include<stdio.h>
struct rec
```

```
{
        int x,y,z;
};
int main()
{
        int counter;
        FILE *ptr_myfile;
        struct rec my_record;
        ptr_myfile=fopen("test.bin","rb");
        if (!ptr_myfile)
        {
                printf("Unable to open file!");
                return 1;
        }
        for ( counter=9; counter >= 0; counter--)
        {
                fseek(ptr_myfile,sizeof(struct rec)*counter,SEEK_SET);
                fread(&my_record,sizeof(struct rec),1,ptr_myfile);
                printf("%d\n",my_record.x);
        }
```

```
        fclose(ptr_myfile);

        return 0;

}
```

In this example we are using fseek to seek the last record in the file. This record we read with fread statement and with the printf statement we print member x of the structure my_record. As you can see the “for loop” also changed. The “for loop” will now countdown to zero. This counter is then used in the fseek statement to set the file pointer at the desired record. The result is that we read-in the records in the reverse order.

A last note: if you set the file position indicator to a position in a file and you want the first position in a file then you can use the function rewind to the first position in the file. The function rewind can be used like this:

```
#include<stdio.h>

/* Our structure */

struct rec

{

        int x,y,z;

};

int main()

{

        int counter;

        FILE *ptr_myfile;

        struct rec my_record;
```

```
        ptr_myfile=fopen("test.bin","rb");
        if (!ptr_myfile)
        {
                printf("Unable to open file!");
                return 1;
        }

        fseek(ptr_myfile, sizeof(struct rec), SEEK_END);
        rewind(ptr_myfile);

        for ( counter=1; counter <= 10; counter++)
        {
                fread(&my_record,sizeof(struct rec),1,ptr_myfile);
                printf("%d\n",my_record.x);
        }
        fclose(ptr_myfile);
        return 0;
}
```

With the fseek statement in this example we go to the end of the file. Then we rewind to first position in the file. Then read-in all records and print the value of member x. Without the rewind you will get garbage.