1. ARRAYS

Many applications require the processing of multiple data items that have common characteristics (e.g., a set of numerical data, represented by $x_1, x_2, x_3, x_4, \ldots, x_n$). In such situations, it is often convenient to palace the data items into an array.

An array is an identifier that refers to a collection of data items that all have the same name. The data items must all be of the same type (e.g., all integers, all characters, all floats, etc.) and the same storage class. The individual data items are represented by their corresponding array element (i.e., the first data item is represented by the first array element, etc.).

The individual array elements are distinguished from one another by the value that is assigned to a *subscript* (sometimes called *index*). An individual data item within an array is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets.

Each subscript must be expressed as a nonnegative number. The value of the subscript can be expressed as an integer constant, an integer variable or a more complex integer expression.

In C, array indices always begin with zero. There are several different ways to recognize arrays (e.g., integer arrays, character arrays, one-dimensional arrays, multi-dimensional arrays)

1.1. DEFINING A ONE-DIMENSIONAL ARRAY

Arrays are defined in the same manner as ordinary variables, except that each array name must be accompanied by a size specification (i.e., the number of elements). For a one-dimensional array, the size is specified by a positive integer expression, enclosed in square brackets. The expression is usually written as positive integer constant.

In general terms, a one-dimensional array definition may be expressed as

storage-class data-type array-name[exp1];

where *storage-class* refers to the storage class of the array, *data-type* is its data type, *array-name* is the name of the array and *exp1* is a positive-valued integer expressions which indicates the number of array elements.

The *storage-class* is optional; the default values are *automatic* for arrays that are defined within a function or a block, and *external* for arrays defined outside of a function.

Several typical one-dimensional array definitions are shown below.

```
int x[100];          /* An 100-element integer array */
char text[80];       /*An 80-element character array */
```



```
static float n[20];    /*a 20-element floating-point static array */
```

It is sometimes convenient to define array size in terms of symbolic constant rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all references to the maximum array size (e.g., within for loop as well as in array definitions) can be altered simply by changing the value of the symbolic constant.

The array definition can include the assignment of initial values, if desired. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas. The general form is

storage-class data-type array-name[exp1]= {value 1, value 2, ...,value n};

where *value 1* refers to the value of the first element, *value 2* refers to the value of the second element, and so on.

Several array definitions that include the assignment of initial values are shown below.

```
int num[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float x[5]={0.1, 0,0.2,0.3,0.1};
char colour[8]={'v', 'i', 'b', 'g', 'y', 'o', 'r'}
```

The results of these initial assignments, in terms of individual array elements are, are as follows.

num[0] = 1	x[0] = 0.1	colour[0] = 'v'
num[1] = 2	x[1] = 0	colour[1] = 'i'
num[2] = 3	x[2] = 0.2	colour[2] = 'b'
num[3] = 4	x[3] = 0.3	colour[3] = 'g'
num[4] = 5	x[4] = 0.1	colour[4] = 'y'
num[5] = 6		colour[5] = 'o'
num[6] = 7		colour[6] = 'r'
num[7] = 8		
num[8] = 9		
num[9] = 10		

The array size need not be specified explicitly when initial values are indicated as a part of array definition. With a numerical array, the array size will automatically be set equal to the number initial values included within the definition.

For example, consider the following array definitions.

```
int num[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float x[]={0.1, 0,0.2,0.3,0.1}
```

Thus, *num* will be a 10-element integer array and *x* will be a 5-element floating-point array. These declarations will result in the same assignment of initial values to array elements, as shown in the previous example.

Strings can be assigned to character type arrays. To do so, the array is usually written without explicit size specification. The proper array size will be assigned automatically. When a string is assigned to a character array, a null character is automatically added at the end of every string.

For example, consider the following character array definition.

```
message[]= "I like C"
```

The elements of this 9-element character array are

message[0]='I'	message[3]='i'	message[6]=' '
message[1]=' '	message[4]='k'	message[7]='C'
message[2]='l	message[5]='e'	message[8]='\0'

1.2. MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets, and so on. In general terms, a multidimensional array definition can be written as

storage-class data-type array-name[exp1] [exp2]... [expn];

where *storage-class* refers to the storage class of the array, *data-type* is the data type, *array-name* is the name of the array, and *exp1*, *exp2*, ..., *expn* are positive valued integer expressions that indicate the number of array elements associated with the each subscript. The *storage-class* is optional; the default values are *automatic* for arrays that are defined within a function or a block, and *external* for arrays defined outside of a function.

Examples of multidimensional array definitions are shown below.

```
/*Floating-point array having 100 rows and 5 columns, hence 500 elements */
float table[100][5];

/*String array having 50 strings of 20 characters each */
char names[50][20];

/* 50 tables each having 100 rows and 5 columns */
int tables[50][100][5];
```

Multidimensional array definitions can include the assignment of initial values, if desired. Initialisation of a two-dimensional array is done by specifying the elements in row major order (i.e., with the elements of the first row in a sequence, followed by those of the second, and so on. An example is given below.

```
int matrix[3][4]=
        {
```



```
        { 1,2,3,4},
        { 5,6,7,8},
        { 9,0,1,2}
      };
```

The first subscript can be omitted from the array definition when initial values are indicated as a part of array definition. For example, the declaration given below, wherein the number of rows is not explicitly specified, is equivalent to the previous one.

```
int matrix[][4] =
        {
          { 1,2,3,4},
          { 5,6,7,8},
          { 9,0,1,2}
        };
```

Also, the inner brackets can be omitted, giving the numbers in one continuous sequence. For example,

```
int matrix[3][4]={1,2,3,4,5,6,7,8,9,0,1,2};
```

has the same effect as the previous example, but is not readable as before.

1.3. PROCESSING AN ARRAY

Single operations, which involve entire arrays, are not permitted in C. Thus, the array processing must be carried out on element-by-element basis. This is usually accomplished within a loop. The number of passes through the loop will therefore equal the number of array elements to be processed.

For example, the following program will read a set of numbers and calculate their average. It then calculates and displays the deviation of each number about the average.

```
/* To display deviation of each number about their average */
#include<stdio.h>
main()
{ int i,n;
  float num[25],sum=0,avg;
  printf("How many numbers ? ");
  scanf("%d",&n);
  for(i=0;i<n;++i)
  { printf("Enter Number - %d : ",i+1);
    scanf("%f",&num[i]);
    sum+=num[i];
  }
  avg=sum/n;
  printf("Number\t\tAverage\t\tDeviation\n");
```

```
    printf("------\t\t--------\t\t---------\n");
    for(i=0;i<n;++i)
    printf("%g\t\t%g\t\t%+g\n", num[i],avg,num[i]-avg);
  }
```

The following program that adds two matrices illustrates the processing of a two-dimensional array.

```
/* Matrix addition */
#include<stdio.h>
main()
{ int a[10][10],b[10][10],c[10][10];
  int i,j,row,col;
  printf("How many rows? ");
  scanf("%d",&row);
  printf("How many columns? ");
  scanf("%d",&col);
  printf("Enter Your First Matrix\n");
  for(i=0;i<row;++i)
    for(j=0;j<col;++j)
    { printf("Element(%d,%d) ? ",i+1,j+1);
      scanf("%d",&a[i][j]);
    }
  printf("Enter Your Second Matrix\n");
  for(i=0;i<row;++i)
    for(j=0;j<col;++j)
    { printf("Element(%d,%d) ? ",i+1,j+1);
      scanf("%d",&b[i][j]);
    }
  for(i=0;i<row;++i)
    for(j=0;j<col;++j)
      c[i][j]=a[i][j]+b[i][j];
  printf("\t\tFirst Matrix\n");
  printf("\t\t----------\n\t\t");
  for(i=0;i<row;++i)
  { for(j=0;j<col;++j)
      printf("%d   ",a[i][j]);
    printf("\n\t\t");
  }
  printf("Second Matrix\n");
  printf("\t\t----------\n\t\t");
  for(i=0;i<row;++i)
  { for(j=0;j<col;++j)
      printf("%d   ",b[i][j]);
    printf("\n\t\t");
  }
  printf("Sum Matrix\n");
```



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

2. CHARACTER ARRAYS & STRINGS

In most applications, a string of characters rather than a single character is necessary. A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and $. String literals or string constants, in C are written in double quotation marks as follows:

"Enter a Number:" (a prompt message)
"Ram Raheem" (a name)
"673639 Areacode" (a street address)
"Kozhikkode" (a city and state)
"(91) 9236745622" (a telephone number)

As an individual character variable can store only one character, we need an array of characters to store strings. Thus, in C, a string is stored in an array of characters ending in the null character ('\0'). Each character in the string occupies one location in an array. The null character '\0' is put after the last character. This is done so that program can tell when the end of a string has been reached. For example, the string "Enter a Number:" is stored as follows

'E'	'n'	't'	'e'	'r'	' '	'a'	' '	'N'	'u'	'm'	'b'	'e'	'r'	':'	'\0'

Since the string has 15 characters (including the space), it requires an array of at least, size 16 to store it.

The terminating null character is important. In fact, a string not terminated by a '\0' is not really a string, but merely a collection of characters.

A string is accessed via a pointer to the first character in the string. The value of a string is the address. Thus, in C, it is appropriate to say that a string is a pointer – in fact, a pointer to the string's first character. In this sense, strings are like arrays, because an array is also a pointer to its first element.

A character array or a variable of type char * can be initialized with a string in a definition.

The definitions

```
char colour[] = "blue";
const char *colourPtr = "blue";
```

each initialize a variable to the string "blue". The first definition creates a 5-element array colour containing the characters 'b', 'l', 'u', 'e' and '\0'. The second definition creates pointer variable *colourPtr* that points to the string "blue" somewhere in memory.

The preceding array definition could also have been written

```
char colour[] = { 'b', 'l', 'u', 'e', '\0' };
```

When defining a character array to contain a string, the array must be large enough to store the string and its terminating null character. The preceding definition automatically determines the size of the array based on the number of initializers in the initializer list.

2.1. READING & WRITING STRINGS

C provides several functions in the standard input/output library (*<stdio.h>*) specifically for reading and writing character and string data.

STRING INPUT

Strings can be entered from the standard input device using the following output functions:

- Formatted input function *scanf*()
- String input function *gets*()
- Single character input function *getchar*()

The formatted input function *scanf* can be used with *%s* format specification to read in a string of characters.

For example,

```
char text[80];
scanf("%s",text);
```

The scanf statement can also include the maximum field width specification for the string data item. The maximum field width for a data item can be specified by placing an unsigned integer indicating the field width between the percent sign (%) and the conversion character. The data item may contain fewer characters than the specified field width. Any characters that extend beyond the specified field width will not be read. Such leftover characters may be incorrectly interpreted as the component of the next data item.

```
char text[80];
scanf("%80s",text);
```

However, the limitation of the *s*-type conversion character is that it applies only to a string that is terminated by a white space character. Therefore, a string that includes

white space characters cannot be entered in this manner.

For example, if the following text message

```
THIS IS A SAMPLE TEXT
```

is entered, only the string "THIS" will be read into the array *text*, since the blank space after the "THIS" will terminate the reading process.

To work with strings that includes white space characters general-purpose format specifier, called *scan set*, can be employed. A scan set defines a set of characters. The sequence of characters is enclosed in square brackets, designated as [...]. White space characters may be included within the brackets, thus accommodating strings that contain such characters.

When the program is executed, successive characters will continue to be read from the keyboard as long as each input character matches one of the characters enclosed within the scan set. The order of the characters within the square brackets need not correspond to the order of the characters being entered. Input characters may be repeated. The string will terminate, however, once an input character is encountered that does not match any of the characters within the brackets. A null character ('\0') will then automatically be added to the end of the string.

The following example illustrates the scanf function to enter a string consisting of uppercase letters and blank spaces. The string will be of undetermined length, but it will be limited to 80 characters, including the null character that is added at the end.

```
char text[80];
scanf(" %[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", text);
```

If the string

```
THIS IS A SAMPLE TEXT
```

is entered from the keyboard when the program is executed, the entire string will be assigned to the array text, since the string is comprised entirely of uppercase letters and blank spaces.

If the string were written as

```
This is a Sample Text
```

however, then only the single character T would be assigned to text, since the first lowercase letter would interpreted as the first character beyond the string. It would, of course, be possible to include both uppercase and lowercase characters within the brackets, but this become cumbersome.

A variation of this feature is which is often more useful is to precede the characters within the square brackets by a circumflex (i.e., ^). This causes the subsequent characters within the brackets to be interpreted in the opposite manner. Thus, when the program is executed, successive characters will continue to be read from the

keyboard as long as each input character does not match one of the characters enclosed within the brackets.

For example, if the characters within the brackets are simply the circumflex followed by a newline character, as in

```
scanf("%[^\n]", text);
```

the string entered from the keyboard can contain any ASCII characters except the newline character.

The *gets*() function can also be used to read a string from a standard input device. The *gets*() function will be terminated by a newline character. The newline character will not be included as part of the string. The string may include white space characters. The string will be terminated by a null character. The *gets*() function has only a single argument, as in

```
gets(text);
```

The single character input function with syntax

```
charactrvariable =getchar();
```

can also be used in an iterative loop for reading string, as in

```
#include<stdio.h>
main()
{   char text[80]; int i=0;
    printf("Text: ");
    while((text[i++]=getchar())!='\n');
    text[i]='\0';
    printf("Text: %s", text);
}
```

WRITING STRINGS

Strings can be printed onto the standard output device using the following output functions:

- Formatted output function *printf*()
- String output function *puts*()
- Single character output function *putchar*()

The *puts*() function is used to display a string on a standard output device. The *puts()* function automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with *puts()* is on its own line.

For example, the following statements, when executed

```
char text[80];
gets(text);
```



```
puts(text);
```

will read and line of text from the standard input and print the text onto standard output.

The single character input function *getchar*() function with syntax

```
putchar(character variable);
```

can be used to output multiple character strings, by displaying one character at a time within a multi-pass loop.

2.2. CHARACTER-HANDLING LIBRARY

The character-handling library (*<ctype.h>*) includes several functions that perform useful tests and manipulations of character data. Each function receives a character – represented as an *int* as an argument. Since the characters have an associated numeric value, they are often manipulated as integers (a character in C is usually a 1-byte integer). The functions of the character handling library are listed below:

Prototype	Function description
int isdigit(int c)	Returns a *true* value if *c* is a digit and 0 (false) otherwise.
int isalpha(int c)	Returns a *true* value if *c* is a letter and 0 otherwise.
int isalnum(int c)	Returns a *true* value if *c* is a digit or a letter and 0 otherwise.
int isxdigit(int c)	Returns a *true* value if *c* is a hexadecimal digit character and 0 otherwise.
int islower(int c)	Returns a *true* value if *c* is a lowercase letter and 0 otherwise.
int isupper(int c)	Returns a *true* value if *c* is an uppercase letter and 0 otherwise.
int tolower(int c)	If *c* is an uppercase letter, tolower returns *c* as a lowercase letter. Otherwise, tolower returns the argument unchanged.
int toupper(int c)	If *c* is a lowercase letter, toupper returns *c* as an uppercase letter. Otherwise, toupper returns the argument unchanged.
int isspace(int c)	Returns a *true* value if *c* is a white-space character – newline ('\n'), space (' '), form feed ('\f'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v')—and 0 otherwise.
int iscntrl(int c)	Returns a *true* value if *c* is a control character and 0 otherwise.
int ispunct(int c)	Returns a *true* value if *c* is a printing character other than a space, a digit, or a letter and returns 0 otherwise.
int isprint(int c)	Returns a *true* value if *c* is a printing character including a space (' ') and returns 0 otherwise.
int isgraph(int c)	Returns a *true* value if *c* is a printing character other than a space (' ') and returns 0 otherwise.

FUNCTIONS *isdigit, isalpha, isalnum & isxdigit*

The following program demonstrates functions *isdigit*, *isalpha*, *isalnum* and *isxdigit*. Function *isdigit* determines whether its argument is a digit (0–9). Function *isalpha* determines whether its argument is an uppercase (*A–Z*) or lowercase letter (*a–z*). Function *isalnum* determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function *isxdigit* determines whether its argument is a hexadecimal digit (*A–F*, *a–f*, 0–9).

```
/* Using functions isdigit, isalpha, isalnum, and isxdigit */
#include <stdio.h>
#include <ctype.h>
main()
{   printf( "%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
      isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
      isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isalpha:",
      isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
      isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
      isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
      isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isalnum:",
      isalnum( 'A' ) ? "A is a " : "A is not a ", "digit or a letter",
      isalnum( '8' ) ? "8 is a " : "8 is not a ", "digit or a letter",
      isalnum( '#' ) ? "# is a " : "# is not a ", "digit or a letter" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
      "According to isxdigit:",
      isxdigit( 'F' ) ? "F is a " : "F is not a ", "hexadecimal digit",
      isxdigit( 'J' ) ? "J is a " : "J is not a ", "hexadecimal digit",
      isxdigit( '7' ) ? "7 is a " : "7 is not a ", "hexadecimal digit",
      isxdigit( '$' ) ? "$ is a " : "$ is not a ", "hexadecimal digit",
      isxdigit( 'f' ) ? "f is a " : "f is not a ", "hexadecimal digit" );
} /* end main */
```

Given below is the output of the program run:

```
According to isdigit:

8 is a digit
# is not a digit

According to isalpha:

A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:

A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:

F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```


FUNCTIONS *islower, isupper, tolower & toupper*

The following program demonstrates functions *islower*, *isupper*, *tolower* and *toupper*. Function *islower* determines whether its argument is a lowercase letter (*a–z*). Function *isupper* determines whether its argument is an uppercase letter (*A–Z*). Function *tolower* converts an uppercase letter to a lowercase letter and returns the lowercase letter. If the argument is not an uppercase letter, tolower returns the argument unchanged. Function *toupper* converts a lowercase letter to an uppercase letter and returns the uppercase letter. If the argument is not a lowercase letter, *toupper* returns the argument unchanged.

```
/* Using functions islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>
int main( void )
{   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to islower:",
      islower( 'p' ) ? "p is a " : "p is not a ", "lowercase letter",
      islower( 'P' ) ? "P is a " : "P is not a ", "lowercase letter",
      islower( '5' ) ? "5 is a " : "5 is not a ", "lowercase letter",
      islower( '!' ) ? "! is a " : "! is not a ", "lowercase letter" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isupper:",
      isupper( 'D' ) ? "D is an " : "D is not an ", "uppercase letter",
      isupper( 'd' ) ? "d is an " : "d is not an ", "uppercase letter",
      isupper( '8' ) ? "8 is an " : "8 is not an ", "uppercase letter",
      isupper( '$' ) ? "$ is an " : "$ is not an ", "uppercase letter" );
```

```
   printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
      "u converted to uppercase is ", toupper( 'u' ),
      "7 converted to uppercase is ", toupper( '7' ),
      "$ converted to uppercase is ", toupper( '$' ),
      "L converted to lowercase is ", tolower( 'L' ) );
   return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
According to islower:

p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:

D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter
u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

FUNCTIONS *isspace, iscntrl, ispunct, isprint & isgraph*

The following program demonstrates functions *isspace*, *iscntrl*, *ispunct*, *isprint* and *isgraph*. Function *isspace* determines if a character is one of the following white-space characters: space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v'). Function iscntrl determines if a character is one of the following control characters: horizontal tab ('\t'), vertical tab ('\v'), form feed ('\f'), alert ('\a'), backspace ('\b'), carriage return ('\r') or newline ('\n'). Function *ispunct* determines if a character is a printing character other than a space, a digit or a letter, such as $, #, (,), [,], {, }, ;, : or %. Function *isprint* determines if a character can be displayed on the screen (including the space character). Function *isgraph* is the same as *isprint*, except that the space character is not included.

```
/* Using functions isspace, iscntrl, ispunct, isprint, isgraph */
#include <stdio.h>
#include <ctype.h>
int main( void )
{   printf( "%s\n%s%s%s\n%s%s%s\n%s%s\n\n",
       "According to isspace:",
       "Newline", isspace( '\n' ) ? " is a " : " is not a ",
```



```
       "whitespace character", "Horizontal tab",
       isspace( '\t' ) ? " is a " : " is not a ",
       "whitespace character",
       isspace( '%' ) ? "% is a " : "% is not a ",
       "whitespace character" );

   printf( "%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
       "Newline", iscntrl( '\n' ) ? " is a " : " is not a ",
       "control character", iscntrl( '$' ) ? "$ is a " :
       "$ is not a ", "control character" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
       "According to ispunct:",
       ispunct( ';' ) ? "; is a " : "; is not a ","punctuation character",
       ispunct( 'Y' ) ? "Y is a " : "Y is not a ","punctuation character",
       ispunct( '#' ) ? "# is a " : "# is not a ","punctuation character" );

   printf( "%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
       isprint( '$' ) ? "$ is a " : "$ is not a ", "printing character",
       "Alert", isprint( '\a' ) ? " is a " : " is not a ", "printing character" );

   printf( "%s\n%s%s\n%s%s%s\n",  "According to isgraph:",
       isgraph( 'Q' ) ? "Q is a " : "Q is not a ", "printing character other than a space",
       "Space", isgraph( ' ' ) ? " is a " : " is not a ", "printing character other than a space"
);
   return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
According to isspace:

Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:

Newline is a control character
$ is not a control character

According to ispunct:

; is a punctuation character
Y is not a punctuation character
# is a punctuation character
According to isprint:
$ is a printing character
Alert is not a printing character
```

According to isgraph:

Q is a printing character other than a space
Space is not a printing character other than a space

2.3. STRING-CONVERSION FUNCTIONS

C include some string-conversion functions in the general utilities library (*<stdlib.h>*). These functions convert strings of digits to integer and floating-point values. The string conversion functions are summarized below:

Function prototype	Function description
*double atof(const char *nPtr)*	Converts the string nPtr to double.
*int atoi(const char *nPtr)*	Converts the string nPtr to int.
*long atol(const char *nPtr)*	Converts the string nPtr to long int.
*double strtod(const char *nPtr, char **endPtr)*	Converts the string nPtr to double.
*long strtol(const char *nPtr, char **endPtr, int base)*	Converts the string nPtr to long.
*unsigned long strtoul(const char *nPtr, char **endPtr, int base)*	Converts the string nPtr to unsigned long.

FUNCTION *atof, atoi & atol*

Function *atof* converts its argument – a string that represents a floating-point number – to a double value. The function returns the double value. If the converted value cannot be represented – for example, if the first character of the string is a letter – the behaviour of function *atof* is undefined.

Function *atoi* converts its argument – a string of digits that represents an integer – to an *int* value. The function returns the *int* value. If the converted value cannot be represented, the behaviour of function *atoi* is undefined.

Function *atol* converts its argument – a string of digits representing a long integer – to a *long* value. The function returns the *long* value. If the converted value cannot be represented, the behaviour of function *atol* is undefined. If *int* and *long* are both stored in 4 bytes, function *atoi* and function *atol* work identically.

The following program illustrates the usage of *atof, atoi* and *atol* functions.

```
/* Using atof, atoi and atol */
#include <stdio.h>
#include <stdlib.h>
int main( void )
{   double d; /* variable to hold converted string */
    int i; /* variable to hold converted string */
    long l; /* variable to hold converted string */
```



```
    d = atof( "99.0" );
    printf( "%s%.3f\n%s%.3f\n",
        "The string \"99.0\" converted to double is ", d,
        "The converted value divided by 2 is ", d / 2.0 );
    i = atoi( "2593" );
    printf( "%s%d\n%s%d\n",
        "The string \"2593\" converted to int is ", i,
        "The converted value minus 593 is ", i - 593 );

    l = atol( "1000000" );
    printf( "%s%ld\n%s%ld\n",
        "The string \"1000000\" converted to long int is ", l,
        "The converted value divided by 2 is ", l / 2 );
    return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

FUNCTION *strtod*

Function *strtod* converts a sequence of characters representing a floating-point value to *double*. The function receives two arguments – a string (*char* *) and a pointer to a string (*char* **). The string contains the character sequence to be converted to double. The pointer is assigned the location of the first character after the converted portion of the string.

For example, in the program given below, the line

d = strtod(string, &stringPtr);

indicates that *d* is assigned the double value converted from *string*, and *stringPtr* is assigned the location of the first character after the converted value in string.

The following program demonstrates the usage of the function *strtod*.

```
/* Using strtod */
#include <stdio.h>
#include <stdlib.h>
int main( void )
{   /* initialize string pointer */
    const char *string = "51.2% are admitted"; /* initialize string */
```

```
   double d; /* variable to hold converted sequence */
   char *stringPtr; /* create char pointer */
   d = strtod( string, &stringPtr );
   printf( "The string \"%s\" is converted to the\n", string );
   printf( "double value %.2f and the string \"%s\"\n", d, stringPtr );
   return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

FUNCTION *strtol* & *strtoul*

Function *strtol* converts to long a sequence of characters representing an integer. The function receives three arguments – a string (*char* *), a pointer to a string and an integer. The string contains the character sequence to be converted. The pointer is assigned the location of the first character after the converted portion of the string. The integer specifies the base of the value being converted.

For example in the following program, the line

x = strtol(string, &remainderPtr, 0);

indicates that *x* is assigned the *long* value converted from *string*. The second argument, *remainderPtr*, is assigned the remainder of string after the conversion. Using *NULL* for the second argument causes the remainder of the string to be ignored. The third argument, 0, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format. The base can be specified as 0 or any value between 2 and 36. Numeric representations of integers from base 11 to base 36 use the characters *A–Z* to represent the values 10 to 35. For example, hexadecimal values can consist of the digits 0–9 and the characters *A–F*. A base-11 integer can consist of the digits 0–9 and the character *A*. A base-24 integer can consist of the digits 0–9 and the characters *A–N*. A base-36 integer can consist of the digits 0–9 and the characters *A–Z*.

The following program demonstrates the usage of the function *strtol*.

```
/* Using strtol */
#include <stdio.h>
#include <stdlib.h>
int main( void )
{   const char *string = "-1234567abc"; /* initialize string pointer */
    char *remainderPtr; /* create char pointer */
    long x; /* variable to hold converted sequence */
    x = strtol( string, &remainderPtr, 0 );
    printf( "%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
```



```
       "The original string is ", string,
       "The converted value is ", x,
       "The remainder of the original string is ",
       remainderPtr,
       "The converted value plus 567 is ", x + 567 );
    return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Function *strtoul* converts to unsigned long a sequence of characters representing an *unsigned long* integer. This function works identically to function *strtol*.

2.4. STRING LIBRARY FUNCTIONS

The string-handling library (*<string.h>*) provides many useful functions for manipulating string data (copying strings and concatenating strings), comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces) and determining the length of strings.

STRING-MANIPULATION FUNCTIONS

The string-manipulation functions of the string-handling library are summarized below. Every function – except for *strncpy* – appends the null character to its result.

Function prototype	Function description
*char *strcpy(char *s1, const char *s2)*	Copies string *s2* into array *s1*. The value of *s1* is returned.
*char *strncpy(char *s1, const char *s2, size_t n)*	Copies at most *n* characters of string *s2* into array *s1*. The value of *s1* is returned.
*char *strcat(char *s1, const char *s2)*	Appends string *s2* to array *s1*. The first character of *s2* overwrites the terminating null character of *s1*. The value of s1 is returned.
*char *strncat(char *s1, const char *s2, size_t n)*	Appends at most n characters of string *s2* to array *s1*. The first character of *s2* overwrites the terminating null character of *s1*. The value of *s1* is returned.

Functions *strncpy* and *strncat* specify a parameter of type *size_t*, which is a type defined by the C standard as the integral type of the value returned by operator *sizeof*.

Function *strcpy* & *strncpy*

Function *strcpy* copies its second argument (a *string*) into its first argument (a character array that must be large enough to store the *string* and its terminating null character, which is also copied). Function *strncpy* is equivalent to *strcpy*, except that *strncpy* specifies the number of characters to be copied from the string into the array. Function *strncpy* does not necessarily copy the terminating null character of its second argument. A terminating null character is written only if the number of characters to be copied is at least one more than the length of the string.

For example, if the string constant "test" is the second argument, a terminating *null* character is written only if the third argument to *strncpy* is at least 5 (four characters in "test" plus a terminating null character). If the third argument is larger than 5, null characters are appended to the array until the total number of characters specified by the third argument are written.

The following program demonstrates the usage of the functions *strcat* and *strncat*.

```
/* Using strcpy and strncpy */
#include <stdio.h>
#include <string.h>
int main( void )
{   char x[] = "Happy Birthday to You"; /* initialize char array x */
    char y[25]; /* create char array y */
    char z[15]; /* create char array z */

    /* copy contents of x into y */
    printf("%s%s\n%s%s\n",
        "The string in array x is: ", x,
        "The string in array y is: ", strcpy( y, x ) );
    /* copy first 14 characters of x into z. Does not copy null character */
    strncpy( z, x, 14 );
    z[ 14 ] = '\0'; /* terminate string in z */
    printf("The string in array z is: %s\n", z );
  return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Functions *strcat* & *strncat*

Function *strcat* appends its second argument (a *string*) to its first argument (a character array containing a string). The first character of the second argument replaces the *null* ('\0') that terminates the string in the first argument. You must ensure

that the array used to store the first string is large enough to store the first string, the second string and the terminating null character copied from the second string. Function *strncat* appends a specified number of characters from the second string to the first string. A terminating null character is automatically appended to the result.

The following program demonstrates functions *strcat* and *strncat*.

```
/* Using strcat and strncat */
#include <stdio.h>
#include <string.h>
int main(void)
{   char s1[] = "Happy "; /* initialize char array s1 */
    char s2[] = "New Year "; /* initialize char array s2 */
    char s3[40] = ""; /* initialize char array s3 to empty */
    printf("s1 = %s\ns2 = %s\n", s1, s2 );
    /* concatenate s2 to s1 */
    printf("strcat( s1, s2 ) = %s\n", strcat(s1, s2));
    /* concatenate first 6 characters of s1 to s3. Place '\0'
    after last character */
    printf("strncat( s3, s1, 6) = %s\n", strncat( s3, s1, 6 ) );
    /* concatenate s1 to s3 */
    printf("strcat( s3, s1) = %s\n", strcat( s3, s1));
    return 0; /* indicates successful termination */
} /* end main */
```

Given below is the output of the program run:

```
s1 = Happy
s2 = New Year
strcat( s1, s2 ) = Happy New Year
strncat( s3, s1, 6 ) = Happy
strcat( s3, s1 ) = Happy Happy New Year
```

COMPARISON FUNCTIONS OF THE STRING-HANDLING LIBRARY

The string-handling library's string-comparison functions prototypes and a brief description of each function, is given below.

Prototype	Function Description
strcmp(s1,s2)	Compares *s1* and *s2* lexicographically. Returns a negative value if *s1*<*s2*; 0 if *s1* and *s2* are identical; and positive value if *s1*>*s2*.
strcmpi(s1,s2)	Compares *s1* and *s2* lexicographically, without regard to case. Returns a negative value if *s1*<*s2*; 0 if *s1* and *s2* are identical; and positive value if *s1*>*s2*.
stricmp(s1,s2)	Equivalent to *strcmpi*.

Given below is the output of the program run:

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

3. FUNCTIONS

A *function* is a self-contained program segment that carries out some specific, well-defined task. A function will carry out its intended action whenever it is accessed (i.e., whenever the function is called). The same function can be accessed from several different places within a program. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed. C supports the use of two types of functions

- Library functions, which are used to carry out a number of commonly used operations or calculations, and
- Programmer defined functions, which are defined by the programmer for carrying out various individual tasks.

The use of programmer-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which can individually be solved. This is known as modularisation. The advantages of modularisation are:

- Avoids the need for redundant coding
- If a particular group of instructions to be accessed repeatedly, from several different places within a program, it can be placed within a function, which can then be accessed whenever it is needed. Thus the use function avoids the need for redundant programming of the same instructions.
- Increase the logical clarity
- Decomposition of lengthy, complicated program into several concise functions will improve the logical clarity of the program. Such programs are easier to write, read and debug.
- Enables the programmer to build a customized library of frequently used routines. Each routine can be programmed as a separate function and stored within a special library file. If a program requires a particular routine, the corresponding library function can be accessed and attached to the program during the compilation process.

In order to make use of a user-defined function, the following components must be included in a program

1. Function definition
2. Function call and
3. Function Declaration using function prototype

3.1. DEFINING A FUNCTION

A function definition has two principal components: the function header and the body of the function. The function header contains the type specification of the value returned by the function, followed by the function name, and (optionally) a set of *arguments*, separated by commas and enclosed in parentheses. Each argument is preceded by its associated type declaration. An empty pair of parentheses must follow the function name if the function definition does not include any arguments.

In general terms, the function header statement can be written as

data-type name(type1 arg1,type2 arg2,...,typen argn)

where *data-type* represents the data type of the item that is returned by the function, *name* represents the function name, and *type1*, *type2*, ..., *typen* represents the data types of the arguments *arg1*, *arg2*, ..., *argn*. The data types are assumed to be of type integer if they are not shown explicitly.

The arguments are called *formal arguments* or *formal parameters*, because they represent the names of the data item that are transferred into the function from the calling portion of the program. The corresponding arguments in the function reference or function call are called *actual arguments* or *actual parameters*, since they define the data items that are actually transferred.

The body of the function is a compound statement that defines the action to be taken by the function. This compound statement can contain expression statements, other compound statements, control statements, and so on.

The body of the function should include one or more *return* statements, in order to return a value to the calling portion of the program. Information is returned from the function to the calling portion of the program via the *return* statement. The *return* statement also causes the program logic to return to the point from which the function was accessed.

In general terms the *return* statement is written as

return(expression);

The value of the *expression* is returned to the calling portion of the program.

For example, the following C program determines the largest of three integer quantities. This program makes use of the function that determines the larger of two integer quantities.

```
/* The main function */
```

```
#include <stdio.h>
int max(int a, int b);
main()
{ int x,y,z;
  printf("First No  : ");
  scanf("%d",&x);
  printf("Second No : ");
  scanf("%d",&y);
  printf("Third No  : ");
  scanf("%d",&z);
  printf("Largest among %d , %d and %d is %d" , x,y,z,max(max(x,y),z
}
/* Function that returns the larger of two integer quantities*/
int max(int a, int b)                    /* Function header */
{ int k;                                 /* Body of the function */
  K=(a>b)?a:b;
  return(k);
}                                        /* end of the function */
```

The first line of the function contains the function

```
int x,y;  /* Global variables */
main()
{
  printf("First No  : ");
  scanf("%d",&x);
  printf("Second No : ");
  scanf("%d",&y);
  max();
}
void max(void)
{  int k;
   K=(x>y)?x:y;
   printf("Largest among %d and %d is %d",x,y,k);
   return;
}
```

3.2. ACCESSING A FUNCTION

A function can be accessed (i.e., called) by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function. The function call may be a part of simple expression, or it may be one of the operands within a more complex expression.

The arguments appearing in the function call are referred to as *actual parameters* and the arguments appearing in the header line of the function definition is known as *formal parameters*. In a normal function call, there will be one actual argument for each formal argument. The actual arguments may be expressed as constants, single variables, or more complex expressions. Each actual argument must be of the same data type as its corresponding formal arguments.

For example, consider the following C program which reads in two integer variable and displays the larger of the two.

```
/* The main function */
#include <stdio.h>
int max(int a, int b);
main()
{ int x,y;
  printf("First No  : ");
  scanf("%d",&x);
  printf("Second No : ");
  scanf("%d",&y);
  printf("Largest of  %d and %d is %d" , x,y,max(x,y));
  getchar();
}
/* Function that returns the larger of two integer quantities*/
```



```
int max(int a, int b)
{ int k;
  k=(a>b)?a:b;
  return(k);
}
```

Within the program, *main* contain only one call to the programmer defined function *max*. The call is the part of the *printf* function. The function call contains two actual arguments, the integer type variables *x* and *y*. When the function is accessed, the value of *x* and *y* are transferred to the function. These values are represented by *a* and *b* within the function. The larger of two data is determined and returned to the calling portion of the program, where it is displayed on the standard output device.

There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. For example, the following program illustrates the repeated use of the function *max* to determine the largest among a set of numbers.

```
/* The main function */
#include <stdio.h>
float max(float a, float b);
main()
{ float num,large;
  int n,i=1;
  printf("How many numbers ? ");
  scanf("%d",&n);
  printf("Enter number - %d : ",i);
  scanf("%f",&num);
  large = num;
  for(;i<n;++i)
  { printf("Enter number - %d : ",i+1);
    scanf("%f",&num);
    large = max(large,num);
  }
  printf("Largest is %g",large);
  getchar();
}
/* Function that returns the larger of two integer quantities*/
float max(float a, float b)
{ float k;
  k=(a>b)?a:b;
  return(k);
}
```

The function *max* will be accessed '*n*' times within the *for* loop.

C permits nesting of function references, in which *main* calls *fn1*, that in turn calls *fn2*, that in turn calls *fn3*, and so on. In fact, call to the same function can also be

nested. For example, a nested call to *max* function defined above to find largest among five numbers looks like

```
large = max(a,max(b,max(c,max(d,e))));
```

3.3. FUNCTION PROTOTYPES

If the function call precedes the function definition, the function must be declared within the calling function. A *function prototype* is used for this purpose. Function prototypes are usually written at the beginning of the program following the include statement. A function prototype specifies the type of value returned by the function, the name of the function and the type(s) of argument(s) expected by the function. This enables the compiler to check that, when the function is used, the correct number and type of arguments are supplied. The general form of the function prototype is

data-type name(type1 arg1,type2 arg2, ...,typen argn);

where *data-type* represents the data type of the item that is returned by the function, *name* represents the function name, and *type1*, *type2*, ..., *typen* represents the data types of the arguments *arg1*, *arg2*, ..., *argn*. Thus, the function prototype resembles the header statement of a function definition, except that the prototype is terminated by a semicolon.

A function prototype statement can be placed either before all the function definition in the beginning of the program following the include statements (*global prototype declaration*) or can be placed inside the calling function along with its definition (*local prototype declaration*).

If the prototype is placed at the beginning of the program following the include statements, the prototype is referred to as *global prototype* and are available to all functions in the program. No separate prototype needs to be included in accessing global functions inside other calling function.

If the function prototype is placed inside the calling function along with its definition, such functions can be accessed only in that function.

Remember that the function prototype statement is not absolutely necessary, if the function is defined before it is accessed. When a function is referenced, compiler will check whether there exist a match between the number and the types of actual and formal arguments and between the return types.

If the function definition precedes the function reference, while compiling the function reference, the compiler will have all the necessary information regarding the referred function.

On the other hand, if a top-down approach for coding is employed, where the top-level functions appears before the lower-level functions, the compiler will not have the necessary information concerning the referred function for checking its validity.

This information concerning the referred function can be provided by including a global or local function prototype declaration of the called function in the calling function.

f a prototype statement is not included in such case, the compiler assumes that the function returns an *int* type and that the types actual parameters matches with the types of the corresponding formals.

If these assumptions went wrong, the linker generates error. Thus, it is considered a good practice to always include function prototype declaration, preferably in global declaration section.

For example, consider the following program that calculates the factorial of a positive integer quantity. The program utilizes the function named *factorial* to determine the factorial of a positive integer. The reference to the function factorial within the main function appears ahead of ifs definition. Therefore a function prototype is included in the beginning of the program following the include statement.

The function prototype indicates that a function called factorial, which accepts an integer quantity and returns a double precision quantity, will be defined later in the program.

```
/* The main function */
#include <stdio.h>
double fact(int x);     /* Function prototype */
main()
{ int n;
  printf(" Enter a positive number : ");
  scanf("%d",&n);
  if(n<0)
    printf("Invalid Input");
  else
    printf("factorial of %d is %g",n,fact(n));
}
/* Function that calculates and return the factorial of an integer */
double fact(int x)
{ int i;
  double f = 1.0;
    for(i=2;i<=x;++i)
      f*=i;
  return f;
}
```

3.4. PASSING ARGUMENTS TO A FUNCTION

In C language, arguments can be passed to a function in two different ways:

```
{ int i,j;
  for(i=0;i<r;++i)
  { for(j=0;j<c;++j)
     printf("%6d",x[i][j]);
    printf("\n");
  }
  return;
}
/* Function for matrix multiplication */
void multiplymat(int x[][10],int y[][10],int z[][10],int r,int c, int d)
{ int i,j,k;
  for(i=0;i<r;++i)
    for(j=0;j<c;++j)
    { z[i][j]=0;
      for(k=0;k<d;++k)
        z[i][j]=z[i][j]+x[i][k]*y[k][j];
    }
  return;
}
```

3.6. CATEGORIES OF FUNCTIONS

For better understanding of arguments and return type in functions, user-defined functions can be categorised as:

1. Function with no arguments and no return value
2. Function with no arguments but return value
3. Function with arguments but no return value
4. Function with arguments and return value.
5. Functions that return multiple values.

Let us take an example to find whether a given number is prime or not using above categories of user defined functions.

FUNCTION WITH NO ARGUMENTS & NO RETURN VALUE

A C function without any arguments means you cannot pass data to the called function. Similarly, function with no return type does not pass data back to the calling function. It is one of the simplest types of function in C. This type of function which does not return any value cannot be used in an expression it can be used only as independent statement.

Let us have an example to illustrate this.

```
/* program to check whether a number entered is prime or not
using function with no arguments and no return value*/
```



```
#include <stdio.h>
#include <math.h>
void prime();
int main()
{
   prime(); /*No argument is passed to prime()*/
   return;
}
/* There is no return value to calling function main().
Hence, return type of prime() is void */
void prime()
{  int num,i,flag=1;
   printf("Enter positive integer >1 to check: ");
   scanf("%d",&num);
   if(num<=1)
    {    printf("Invalid Data");
         exit(0);
    }
   for(i=2;i<=sqrt(num);++i)
       if(num%i==0)
       {   flag=0;
           break;
       }
   if (flag)
     printf("%d is prime",num);
   else
     printf("%d is not prime",num);
}
```

Function *prime()* is used for asking user a input, check for whether it is prime of not and display it accordingly. No argument is passed and returned form *prime()* function

FUNCTION WITH NO ARGUMENTS BUT RETURN VALUE

We may need a function which does not take any argument but only returns values to the calling function then this type of function is useful. The best example of this type of function is *getchar()* library function which is declared in the header file *stdio.h*.

We can declare a similar function for our prime number problem. Take a look.

```
/* program to check whether a number entered is prime or not
using function with no arguments but return value*/
#include <stdio.h>
#include <math.h>
int input();
int main()
```

```
{   int num,i,flag = 1;

    num=input(); /* No argument is passed to input() */
    if(num<=1)
    {   printf("Invalid Data");
        exit(0);
    }
    for(i=2; i<=sqrt(num); ++i)
        if(num%i==0)
        {   flag = 0;
            break;
        }
    if(flag)
        printf("%d is prime",num);
    else
        printf("%d is not prime", num);
    return;
}
/* Integer value is returned from input() to calling function */
int input()
{   int n;
    printf("Enter positive integer to check: ");
    scanf("%d",&n);
    return n;
}
```

There is no argument passed to *input()* function. But, the value of *n* is returned from *input()* to *main()* function.

FUNCTION WITH ARGUMENTS BUT NO RETURN VALUE

A C function with arguments can perform much better than previous function type. This type of function can accept data from calling function. In other words, you send data to the called function from calling function but you cannot send result data back to the calling function. Rather, it displays the result on the terminal. But we can control the output of function by providing various values as arguments.

Let us have an example to get it better.

```
/* program to check whether a number entered is prime or not
using function with arguments but no return value*/
#include <stdio.h>
void prime(int);
int main()
{   int num;

    printf("Enter positive integer >1 to check: ");
```



```
    scanf("%d",&num);
    if(num<=1)
    {   printf("Invalid Data");
        exit(0);
    }
    prime(num); /* argument is passed to prime(int)*/
    return;
}
/* There is no return value to calling function main().
Hence, return type of prime(int) is void */
void prime(int x)
{   int i,flag=1;
    for(i=2;i<=sqrt(x);++i)
        if(x%i==0)
        {   flag=0;
            break;
        }
    if (flag)
        printf("%d is prime",x);
    else
        printf("%d is not prime",x);
}
```

Here, *prime()* function is used for check whether a given number is prime or not and display it accordingly. Here, argument is passed to the called function but value is not returned to the calling function.

FUNCTION WITH ARGUMENTS AND RETURN VALUE

This type of function can send arguments from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function. And this type of function is mostly used in programming world because it can do two way communications; it can accept data as arguments as well as can send back data as return value. The data returned by the function can be used later in the calling function for further calculations.

```
/* program to check whether a number entered is prime or not
using function with arguments and return value*/
#include <stdio.h>
int main()
{   int num;
    int prime(int);
    printf("Enter positive integer >1 to check: ");
    scanf("%d",&num);
    if(num<=1)
    {   printf("Invalid Data");
```

```
            exit(0);
        }
        if(prime(num)) /* argument is passed to prime(int)*/
            printf("%d is prime",num);
        else
            printf("%d is not prime",num);
        return;
}
/* int value is returned from prime(int) */
int prime(int x)
{   int i;
    for(i=2;i<=sqrt(x);++i)
        if(x%i==0) return(0);
    return(1);
}
```

Here, *prime()* function is used for checking whether a number is prime or not. In this program, input from user is passed to function *prime()* and an integer value is returned from it. A 1 is returned if the input the number is prime; 0 otherwise.

FUNCTIONS THAT RETURN MULTIPLE

In all the three functions, matrices are not explicitly returned, but they are returned to the calling function either as pointer or as global variable.

For example, though the function *readmat(int [][10], int, int)* returns *void*, the read matrix is returned to the calling function because the actual argument is passed to the called function as reference and hence changes made to the elements are reflected in the calling function. The same is true in the case of *void printmat(int [][10], int, int)* function. Note the keyword void as

In the *void addmat(int, int)* function, however, the matrices are not explicitly passed to the called function nor returned from it. Since the matrices are declared as global variables, their scope will be the entire program and the changes made in any function will be reflected everywhere. This, in effect, equivalent to returning multiple values from the called function.

3.7. STORAGE CLASSES – PERMANENCE & SCOPE

Data type and storage class are the two different ways to characterize variables. Data type refers to the type of information represented by a variable, e.g., integer, floating-point, character, etc.

Storage class refers to the *permanence* and *scope* of a variable within the program. Permanence or *longevity* is the period during which a variable retains its value during the execution of a program and the *scope* is the portion of the program over which the variable is recognized for use.

The permanence and scope together defines the *visibility* of a variable. It is the accessibility of a variable from memory.

There are four storage class specifications in C: Automatic, External, Static and Register. They are identified by the keywords *auto, extern, static* and *register*, respectively.

The storage class associated with a variable can sometimes be established simply by the location of the variable declaration within the program. In other situations, however, the keyword that specifies a particular storage class must be placed at the beginning of the variable declaration.

AUTOMATIC (LOCAL) VARIABLES

Automatic variable are always declared within a function and are *local* to the function in which they are declared, i.e., their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.

Any variable declared within a function is interpreted as automatic variable unless a different storage class specification is shown within the declaration. This includes

formal argument declarations. A variable can be explicitly defined as automatic variable by placing the keyword *auto* at the beginning of the variable declaration. Since the location of the variable declarations within the program determines the automatic storage class, the keyword *auto* is not required at the beginning of each variable declaration.

For example, consider a program for checking whether a given number is prime or not. Within *main()*, *num* is an automatic variable. Within the function *prime*, *flag* and *i* are automatic variables.

```
#include<stdio.h>
#include<math.h>
main()
{ auto int num;
  int prime(int x);
  printf("Enter a Number : ");
  scanf("%d",&num);
  if(num<0)
    printf("Invalid Input");
  else
    if(prime(num))
      printf("%d is a Prime number",num);
    else
      printf("%d is not a Prime number",num);
}
int prime(int x)
{ auto int flag = 0,i;
  for(i=2;i<=sqrt(x);++i)
    if(x%i==0)
      return flag;
  flag = 1;
  return flag;
}
```

Automatic variables can be assigned initial values by including appropriate expressions within the variable declaration or by explicit assignment expressions elsewhere in the function. Such values are assigned each time the function is re-entered. If an automatic variable is not initialised in some manner, its initial value will be unpredictable. An automatic variable does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited.

To illustrate these points, consider the following program that displays the factorial of numbers from 1 to 25. Note that both the functions *main* and *fact* contain one automatic variable named *i*. The automatic variable *i* has different meanings within each function and are independent of one another.

```
/* To display the factorial of number from 1 to 25*/
#include <stdio.h>
main()
{ auto int i;
  double fact(int n);
  printf("Number\t\tFactorial\n");
  for(i=1;i<=25;++i)
    printf("%d\t\t%g\n",i,fact(i));
}
/* Function that calculates and return the factorial of an integer */
double fact(int n)
{ auto int i;
  double f = 1.0;
    for(i=2;i<=n;++i)
      f*=i;
  return f;
}
```

EXTERNAL (GLOBAL) VARIABLES

External variables are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. Since the external variables are recognized globally, they can be accessed from any function that falls within their scope. They retain their assigned values within this scope. Therefore, an external variable can be assigned a value within one function, and this value can be used (by accessing the external variable) within another function.

The use of external variables provides a convenient mechanism for transferring information back and forth between functions. In particular, we can transfer information into a function without using arguments. This is convenient when a function requires numerous input data items. We can also transfer multiple data items out of a function (return statement can return only one data item).

When working with external variables, we must distinguish between *external variable definition* and *external variable declaration*. An external variable definition is written the same manner as an ordinary variable declaration. It must appear outside of, and usually before, the functions that accesses the external variables. An external variable definition will automatically allocate the required storage space for the external variable within the computer's memory. External variables can be assigned initial values as a part of variable definitions. If the initial value is not included in the definition of an external variable, the variable will be automatically assigned a value of zero. The storage-class specifier *extern* is not required in an external variable definition, since the external variables will be identified by the location of their definition within the program.

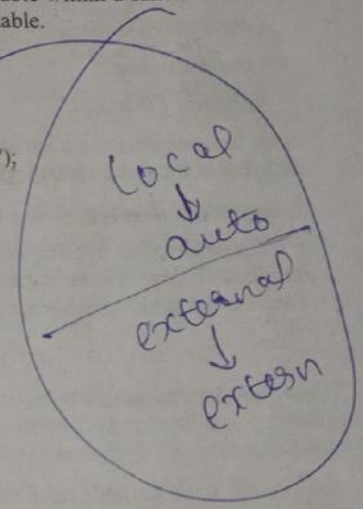
If a function requires an external variable that has been defined earlier in the program,

then the function may access the external variable freely, without any special declaration within the program. On the other hand, if the function definition precedes the external variable definition, then the function must include a declaration for that external variable. An external function declaration must begin with the storage-class specifier *extern*. The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside the function. Storage space will not be allocated to external variables as a result of external variable declaration. Moreover, an external variable declaration cannot include the assignment of initial values. The following sample program illustrates the definition, declaration and use of global variables. The output of this sample program illustrates that the any alteration to the value of an external variable within a function will be recognized within the entire scope of the external variable.

```
/* Illustrating the use of global variables */
#include <stdio.h>
void modify(void)
{ extern int x,y; /* Global variable declaration */
  printf("Before modifying within the modify function\n");
  printf("\t\tx = %d and y = %d\n",x,y);
  x++;
  ++y;
  printf("After modifying within function :\n");
  printf("\t\tx = %d and y = %d\n",x,y);
  return;
}

int x=1,y=2; /* Global variable definition */
void modify(void); /* Function prototype */

main()
{
  printf("Before modifying within main()\n");
  printf("\t\tx = %d and y= %d\n",x,y);
  x++;
  ++y;
  printf("After modifying within main()\n");
  printf("\t\tx = %d and y = %d\n",x,y);
  modify();
  printf("After returning from the modify function to the main()\n");
  printf("\t\tx = %d and y = %d\n",x,y);
}
```



This sample program will generate the following output.

```
Before modifying within main()
        x = 1 and y= 2
After modifying within main()
```

```
        x = 2 and y = 3
Before modifying within the modify function
        x = 2 and y = 3
After modifying within function :
        x = 3 and y = 4
After returning from the modify function to the main()
        x = 3 and y = 4
```

STATIC VARIABLES

A variable is declared to be static by prefixing its normal declaration with the keyword *static*, as in

static int Fibonacci;

We could also use

int static Fibonacci;

since the properties of a variable may be stated in any order. Initial values can be included in the static variable declaration as in

int static Fibonacci = 1;

In the absence of an explicit initialisation, C guarantees that static variables will be initialised to zero. A static variable can be either internal or external. The declaration of an internal static variable appears inside a function, as in

```
fun()
{ ...
  static int Fibonacci = 1;
  ...
}
```

An internal static variable have the same scope as automatic variables, i.e., they are local to the functions in which they are defined. Unlike automatic variables, however, the static variables retain their values throughout the life of the program. Thus, if a function is exited and then re-entered at a later time, the static variables defined within the function will retain their former values.

This is illustrated in following program that calculates the successive Fibonacci numbers. Note that *f1* and *f2* are static variables that are each assigned initial values. These initial values are assigned only once, at the beginning of the program execution. The subsequent values are retained between successive function calls, as they are assigned.

```
#include<stdio.h>
main()
{ int n,i;
  double fib(int n);
```



```
  printf("How many Fibonacci Number ? ");
  scanf("%d",&n);
  printf("The first %d Fibonacci Numbers are :\n",n);
  for(i=1;i<=n;++i)
    printf("%g\t",fib(i));
}
double fib(int n)
{ static double f1=1.0,f2=0.0;
  double f;
  f=f1+f2;
  f1=f2;
  f2=f;
  return f;
}
```

To appreciate the usefulness of the static variable, check the output of the above program after deleting the *static* keyword from the local variable declaration in the *fib* function (then the output will looks like 1, 1, 1, 1, 1, 1, etc., instead of the Fibonacci series).

The declaration of an external static variable appears outside of any function, as in

```
static int Fibonacci = 1;
fun()
{ ...
  ...
}
```

then the variable is known in the remainder of the file containing the declaration. The difference between the external variable and the external static variable is that the latter is unknown outside of the file in which it is declared. Thus, the scope of external static variable is limited to one file only.

The word static denotes permanence. Whether internal or external, a static variable is allocated storage only once, which is retained for the duration of the program. A static variable also affords a degree of 'privacy'. If it is internal, it is known only in the function in which it is declared. If it is external, it is known only in the file in which it is declared.

REGISTER VARIABLES

Registers are the special storage areas within the computer's central processing unit. The actual arithmetic and logical operations that comprise a program are carried out within these registers. Normally, these operations are carried out by transferring information from the computer's memory to these registers, carrying out the indicated operations, and then transferring the results back to the computer's memory. This general procedure is repeated many times during the course of a

program's execution.

The execution time of a program can be reduced considerably if certain values can be stored within these registers rather than in computer's memory. Such programs may also be somewhat smaller in size (i.e., they may require fewer instructions), since fewer data transfers will be required.

In C, the values of the register variables are stored within the registers of the central processing unit. A variable can be assigned this storage class simply by preceding the type declaration with the keyword *register*. This storage class is usually applied to a variable which will be heavily used in the program. In any case, only int, char and pointer variables may be declared as register variables. Furthermore, register may be applied only to automatic variables and to the formal parameters of functions.

The register and automatic storage classes are closely related. In particular, their scope is the same. Thus, the register variables, like automatic variables, are local to the function in which they are declared. Furthermore, the rules governing the use of register variables are the same as those for automatic variables, except that the address operator (&) cannot be applied to the register variables. Where possible, a register variable is assigned to a CPU register, rather than a normal memory location. If there are more register declarations than available CPU registers, then the word register is ignored for the excess declarations. If a register declaration is not honoured, the variables will be treated as having the storage class automatic. Unfortunately, there is no way to determine whether a register declaration will be honoured, other than to run a program with and without declaration and compare the result. A program that makes use of register variables should run faster than the corresponding program without register variables.

3.8. RECURSION

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in recursive form, and second, the problem statement must include a stopping condition.

As an example, consider the problem of finding the factorial of a positive integer quantity.

$$n! = 1 \times 2 \times 2 \times \ldots \times n,$$

where n is a positive integer.

This can be expressed as

$$n! = n \times (n\text{-}1)!$$

This is a recursive statement of the problem, in which the desired action n! is expressed in terms of a previous result, i.e., $(n\text{-}1)!$, which is assumed to be known. Also, we know that 0! = 1. This provides a stopping condition for the recursion.

```
/* Factorial using recursion */
#include<stdio.h>
main()
{ int num;
  double fact(int n);
  printf("Enter a Number : ");
  scanf("%d",&num);
  if(num<0)
    printf("Invalid Input");
  else
    printf("%d! is %g ",num,fact(num));
}

double fact(int n)
{ if(n==0)
    return(1);
  else
    return(n*fact(n-1));
}
```

When the program is executed, the function *fact* is accessed repeatedly, once in *main* and $(n\text{-}1)$ times within itself. When a recursive program is executed, the recursive function calls are not executed immediately. Rather, they are placed on a stack until the condition that terminates the recursion is encountered. (A stack is last-in first-out data structure in which successive data items are 'pushed down' upon preceding data items. The items are later removed (i.e., they are popped) from the stack in reverse order). The function calls are then executed in reverse order. Thus, while evaluating a factorial recursively, the function call will proceed in the following order.

$n! = n \times (n\text{-}1)!$
$(n\text{-}1)! = (n\text{-}1) \times (n\text{-}2)!$
$(n\text{-}2)! = (n\text{-}2) \times (n\text{-}3)!$
.....
$2! = 2 \times 1!$
$1! = 1 \times 0!$

The actual values will then be returned in the following reverse order.

$0! = 1$
$1! = 1 \times 0! = 1 \times 1 = 1$
$2! = 2 \times 1! = 2 \times 1 = 2$
$3! = 3 \times 2! = 3 \times 2 = 6$
$4! = 4 \times 3! = 4 \times 6 = 24$

```
Factorial of first 5 whole numbers
1!=1,2!=2,3!=6,4!=24,5!=120
```

4. POINTERS

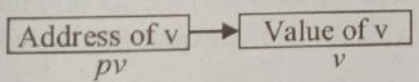
Pointer is a variable that represents the location of a data item, such as a variable or an array element. Within computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent bytes). The number of memory cells required to store a data item depends on the type of the data item. For example, a single character will typically be stored in one byte of memory; an integer usually requires two contiguous bytes; a floating-point number may require four contiguous bytes, and a double-precision quantity may require eight contiguous bytes.

Suppose *v* is variable, that represents some particular data item. The compiler will automatically assign memory cells to this data item. The data item can then be accessed if we know the address of the first memory cell. The address of *v*'s memory location can be determined by the expression *&v*, where & is a unary operator, called the address operator, that evaluates the address of its operand.

Now let us assign the address of *v* to another variable, *pv*. Thus,

```
pv = &v;
```

This new variable is called a pointer to v, since it points to the location where v is stored in the memory. Thus, pv is referred to as a pointer variable. The relationship between pv and v is illustrated below



The data item represented by *v* (i.e., the data item stored in *v*'s memory cells) can be accessed by the expression **pv*, where * is a unary operator, called the indirection operator, that operates only on a pointer variable. Therefore, **pv* and *v* both represent the same data item.

The address operator (&) and the indirection operators (*) are unary operators and they are the members of the same precedence group as the other unary operators. The address operator (&) must act up on operands that associated with unique addresses, such as an ordinary variable or single array element. Thus, the address operators cannot act upon arithmetic expressions. The indirection (*) operator can only act upon operands that are pointers (e.g., pointer variables)

4.1. POINTER DECLARATION

Pointer variables, like all other variables, must be declared before they may be used in a C program. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies that the variable is a pointer. The data

type that appears in the declaration refers to the object of the pointer, i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as

*data-type *ptvar;*

where *ptvar* is the name of the pointer variable, and *data-type* refers to the data-type of the pointer's object.

For example, a C program contains the following declarations.

```
int u,*pu;
float v, *pv;
```

The first line declares *u* to be an integer type variable and *pu* to be a pointer variable whose object is an integer quantity. The second line declares *v* to be a floating-point type variable and *pv* to be a pointer variable whose object is a floating-point quantity.

Within a variable declaration, a pointer variable can be initialised by assigning it the address of another variable. However, the variable whose address is assigned to the pointer variable must have been declared earlier in the program.

For example, a C program contains the following declarations.

```
float v;
float *pv=&v;
```

The first line declares *v* to be floating-point type variable and the second line declares *pv* to be a pointer variable whose object is a floating-point quantity. In addition, the address of *v* is initially assigned to *pv*.

The following program illustrates the use of pointer variables.

```
#include<stdio.h>
main()
{ int v = 5;
   int *pv=&v;
   printf("\n*pv=%d,v=%d,&v=%X and pv=%X",*pv,v,&v,pv);
}
```

When this program is executed, the following output is generated.

*pv=5,v=5,&v=FFD2 and pv=FFD2

The first line declares v to be an integer variable that represents the value 5. The second line declares pv to be a pointer variable whose object is an integer quantity. In addition, the address of v is initially assigned to pv. The address of v is determined automatically by the compiler as FFD2 (hexadecimal). The pointer pv is assigned this value; hence, pv also represents the address FFD2.

4.2. PASSING POINTERS TO A FUNCTION

Pointers can be passed to function as arguments and can also be returned to the calling function. This allows the data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function. . This form of argument passing is referred to passing by reference.

When pointers are used as actual arguments, the corresponding formal arguments must be declared as pointer variable in the formal argument declaration within the function. The usage of pointer arguments is illustrated in the following examples.

```
#include <stdio.h>
void max(int *px, int *py);
main()
{   int x,y;
    printf("First No  : ");
    scanf("%d",&x);
    printf("Second No : ");
    scanf("%d",&y);
    max(&x,&y);
}
void max(int *px, int *pv)
{   int k;
    K=(*px>*py)?*px:*py;
    printf("Largest among %d and %d is %d ",*px,*py,k);
    return;
}
```

4.3. OPERATIONS ON POINTERS

The C language allows arithmetic operations to be performed on pointer variables. It is, however, the responsibility of the programmer to see that the result obtained by performing pointer arithmetic is the address of relevant and meaningful data. The arithmetic operators available for use with pointers can be classified as:

Unary operators: ++ (increment) and □□ (decrement)
Binary operators: + (addition) and – (subtraction)

The C compiler takes the size of the data type being pointed to into account, while performing arithmetic operations on a pointer. For example, if a pointer to an integer is incremented using the ++ operator, then the address contained in the pointer is incremented by four and not one, assuming that an integer occupies four bytes in memory. Similarly, incrementing a pointer to a float causes the initial address contained in the float pointer to be actually incremented by 4 and not 1 (if the size of

```
/* Function for matrix subtraction */
void submat(int (*x)[10],int (*y)[10],int (*z)[10],int r,int c)
{  int i,j;
  for(i=0;i<r;++i)
    for(j=0;j<c;++j)
      *(*(z+i)+j)=*(*(x+i)+j)-*(*(y+i)+j);
  return;
}
```

4.7. ADVANTAGES OF POINTERS

- Pointers can be used to pass information back and forth between a function and its reference point.
- Pointers provide a way to return multiple data items from a function via function arguments
- Pointers permits passing function as arguments to another function.
- Pointers provide an alternate way to access individual array elements.
- Pointers provide a convenient way to represent multi-dimensional arrays.

5. STRUCTURES

A *structure* in C is a heterogeneous data type. A structure may contain different data types. It groups variables into a single entity. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members.

5.1. DEFINING A STRUCTURE

A structure *definition* forms a template that can be used to create structure objects. A structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defied as

```
struct tag
{  member1;
   member2;
   ..........
   membern;
};
```

In this declaration, *struct* is required keyword; *tag* is a name that identifies structures having this composition; and *member1*, *member2*, ..., *membern* are individual member declarations. (There is no formal distinction between a structure definition and a structure declaration; the terms are used interchangeably).

The individual members can be ordinary variables, pointers, arrays, or other structures. The member name within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable that is defined outside the structure. Storage-class cannot be assigned to individual members and individual members cannot be initialised within a structure type declaration.

Once the composition of structure has been defined, individual structure-type variables can be declared as follows:

storage-class struct tag var1, var2, ..., varn;

where *storage-class* is an optional storage class specifier, *struct* is a required keyword, *tag* is the name that appeared in the structure declaration, and *var1*, *var2*, ..., *varn* are structure variables of type *tag*.

A typical structure declaration is shown below.

```
struct account
{   int acct_no
    char acct_type;
    char name[25];
    float balance;
};
```

This structure is named *account* (i.e., the *tag* is *account*). It contains four members: an integer quantity (*acct_no*), a single character (*acct_type*), a 25-element character array (*name[25]*), and a floating-point quantity (*balance*).

We can now declare the structure variables *oldcustomer* and *new customer* as follows.

struct account oldcustomer, new customer;

Thus, the *oldcustomer* and *new customer* are structure variables of type account.

It is possible to combine the declaration of structure composition with that of the structure variables, as shown below.

```
storage-class struct tag
{   member1;
    member2;
    ..........
    membern;
} var1, var2, ..., varn;
```

The *tag* is optional in this situation.

The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct account
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
} oldcustomer, new customer;
```

Thus, the *oldcustomer* and *new customer* are structure variables of type *account*.

Since the variable declarations are now combined with the declaration of the structure type, the tag (i.e., *account*) need not be included. Thus, the above declaration can also be written as

```
struct
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
} oldcustomer, new customer;
```

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure. For example, a C program contains the following structure declaration.

```
struct date
{   int month;
    int day;
    int year;
}
struct
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
    struct date lastpayment;
} oldcustomer, new customer;
```

The second structure *account* now contains another structure *date* as one of its members. Note that the declaration of *date* precedes the declaration of the *account*.

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

storage-class struct tag variable = { val1, val2, ..., valn};

where *val1* refers to the value of the first member, *val2* refers to the value of the second member and so on. The following example illustrates the assignment of initial

values to the members of a structure variable.

```
struct date
{   int month;
    int day;
    int year;

}
struct account
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
    struct date lastpayment;

};
struct account customer = {12345,'r',"james",5687.00,5, 23,2002};
```

Thus, *customer* is structure variable of type *account*, whose members are assigned initial values. The first member (*acct_no*) is assigned the integer value 12345, the second member (*acct_type*) is assigned the character 'R', the third member (*name[25]*) is assigned the string "Jones", and the fourth member (*balance*) is assigned floating-point value 5687.00. The last member is itself a structure that contains three integer members (*month*, *day* and *year*). Therefore, the last member of customer is assigned the integer values 5, 23 and 2002, respectively.

It is also possible to define an array of structures; i.e., an array in which each element is a structure. The procedure is illustrated in the following example.

```
struct date
{   int month;
    int day;
    int year;
}
struct account
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
    struct date lastpayment;
} customer[100];
```

In this declaration customer is a 100-element array of structures. Hence, each element of customer is a separate structure of type account. An array of structures can be assigned initial values just as any other array.

5.2. PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. A structure member can be accessed by writing

variable.member

where variable refers to the name of the structure-type variable, and member refers to the name of the member within the structure. The period (.) operator separates the variable name form the member name. The period operator is a member of the highest precedence group, and its associativity is left to right. For example, consider the following structure declaration.

```
struct date
{   int month;
    int day;
    int year;
}
struct account
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
    struct date lastpayment;
} account;
```

In this example customer is a structure variable of type account. To access the member *acct_no*, we would write

customer.acct_no

Similarly, customer's name and customer's balance can be accessed by writing

customer.name

and

customer.balance

Since the period operator is a member of the highest precedence group, this operator will take precedence over unary operators as well as various arithmetic, logical and assignment operators. Thus, an expression of the form *++variable.member* is equivalent to *++(variable.member)*; i.e., the ++ operator will apply to the structure member, not the entire structure variable.

Similarly, the expression *&variable.member* is equivalent to *&(variable.member)*; thus, the expression accesses the address of the structure member, not the starting address of the structure variable.

More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing

variable.member.submember

where *member* refers to the name of the member within the outer structure, and

submember refers to the name of the member within the embedded structure. Similarly if a structure member is an array, then the individual array element can be accessed

variable.member[exp]

where *exp* is a nonnegative value that indicates the array element.

The use period operator can be extended to array of structures, by writing

array[exp].member

where array refers to the array name, and *array*[*exp*] is an individual array element. Therefore, *array*[*exp*].*member* will refer to a specific member within a particular structure.

The following program illustrates the use of structures for preparing the result of an examination. This program will read the name and marks in 5 subjects for a set of students and will display their result.

```
/* To prepare the result of an examination */
#include<stdio.h>
main()
{ int i,j,n,sum;
  struct{
      char name[20];
      int mark[5];
      int tot;
      float avg;
      } stud[50];
  printf("How many students ? ");
  scanf("%d", &n);
  for(i=0;i<n;++i)
  {
    printf("Enter the details of Student-%d\n\n",i+1);
    printf("Name ? ");
    scanf("   %[^\n]",stud[i].name);
    for(sum=0,j=0;j<5;++j)
    { printf("Mark in Paper-%d ? ",j+1);
      scanf("%d",&stud[i].mark[j]);
      sum+=stud[i].mark[j];
    }
    stud[i].tot=sum;
    stud[i].avg=(float)sum/5.0;
  }
  printf("Name\t\tM1\tM2\tM3\tm4\tM5\tTotal\tAvg\tRes\n\n");
  for(i=0;i<n;++i)
  { printf("%-10s",stud[i].name);
    for(j=0;j<5;++j)
```

```
        printf("%8d",stud[i].mark[j]);
      printf("%8d%8.2f",stud[i].tot,stud[i].avg);
      if(stud[i].avg>35.0)
        printf("  Passed\n");
      else
        printf("  Failed\n");
    }
}
```

5.3. STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of address (&) operator. Thus, if *var* represents a structure-type variable, then *&var* represents the starting address of that variable. We can also declare a pointer variable for a structure by writing

*type *ptrvar;*

where *type* is data type that identifies the composition of the structure, and *ptrvar* represents the name of the pointer variable. The beginning address of structure variable can be assigned to a structure-type pointer variable by writing

ptrvar = &var;

For example, consider the following structure definition.

```
typedef struct
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
} accounlt;
account customer, *pcustomer;
```

In this example, *customer* is a structure variable of type *account*, and *pcustomer* is pointer variable whose object is a structure variable of type *account*. Thus, the beginning address of *customer* can be assigned to the *pcustomer* by writing

pcustomer = &customer;

The variable and pointer declarations can be combined with the structure declaration by writing

```
Struct
{   member1;
    member2;
    ..........
    membern;
} variable, *ptrvar;
```


The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct
{   int acct_no;
    char acct_type;
    char name[25];
    float balance;
} customer, *pcustomer;
```

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

ptrvar–>member

where ptrvar refers to a structure-type pointer variable and –> is comparable to the period (.) operator. Thus the expression

ptrvar–>member

is equivalent to writing

var.member

where *var* is a structure-type variable. The operator –> falls into the highest precedence group. Its associativity is left to right.

The –> operator can be combined with the period operator to access a submember within a structure. Hence, the submember can be accessed by writing

ptrvar–>member.submember

Similarly, the –> operator can be used to access an element of an array that is a member of the structure. This is accomplished by writing

ptrvar–>member[exp]

where *exp* is a nonnegative integer that indicates the array element.

The following program illustrates the use of pointers for processing structures.

```
/* To prepare the result of an examination */
#include<stdio.h>
main()
{
  int i,j,n,sum;
  struct{ char name[20];
          int mark[5];
          int tot;
          float avg;
        } stud[50],*pstud=stud;
  printf("How many students ? ");
```

```
    scanf("%d", &n);
    for(i=0;i<n;++i)
    {
      printf("Enter the details of Student-%d\n\n",i+1);
      printf("Name ? ");
      scanf("  %[^\n]",(pstud+i)->name);
      for(sum=0,j=0;j<5;++j)
      { printf("Mark in Paper-%d ? ",j+1);
        scanf("%d",&(pstud+i)->mark[j]);
        sum+=(pstud+i)->mark[j];
      }
      (pstud+i) ->tot=sum;
      (pstud+i) ->avg=(float)sum/5.0;
    }
    printf("Name\t\tM1\tM2\tM3\tm4\tM5\tTotal\tAvg\tRes\n\n");
    for(i=0;i<n;++i)
    { printf("%-10s",(pstud+i) ->name);
      for(j=0;j<5;++j)
        printf("%8d",(pstud+i) ->mark[j]);
      printf("%8d%8.2f", (pstud+i) ->tot, (pstud+i) ->avg);
      if((pstud+i) ->avg>35.0)
        printf("  Passed\n");
      else
        printf("  Failed\n");
    }
}
```

6. UNIONS

A *union* is a data structure in C that allows the overlay of more than one variable in the same memory area. Unions, like structures, contain members whose individual data types may differ from one another. However, the members within the union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at a time.

In terms of declaration syntax, union is similar to a structure. The only change in declaration is the substitution of the keyword *union* for the keyword *struct*. In general terms, the composition of a union is defined as

```
union tag
{  member1;
   member2;
   .....
   membern;
```



```
};
```

where *union* is required keyword; *tag* is a name that identifies unions having this composition; and *member1*, *member2*, ..., *membern* are individual member declarations.

All members inside a union share storage space. The compiler will allocate sufficient storage for the union members to accommodate the largest member in the union. Other members of the union use the same space. This is how union differs from structure. Individual members in a union occupy the same location in memory. Thus, writing into one will overwrite the other.

Once the composition of union has been defined, individual union-type variables can be declared as follows:

storage-class union tag var1, var2, ..., varn;

where *storage-class* is an optional storage class specifier, *union* is a required keyword, *tag* is the name that appeared in the union declaration, and *var1*, *var2*, ..., *varn* are union variables of type tag.

A typical union declaration is shown below.

```
union id
{   char colour[12];
    int size;
};
```

This union is named *id* (i.e., the tag is *id*). It contains two members: a 12-element character array (*colour[12]*) and an integer quantity (*size*). The 12-element character string will require more storage area within the computer's memory than integer quantity. Therefore, a block of memory large enough for the 12-element character string will be allocated to each union variable. The compiler will automatically distinguish between the 12- element character array and the integer quantity within the given block of memory, as required.

We can now declare the union variables *shirt* and *blouse* as follows.

```
union id shirt, blouse;
```

Thus, the *shirt* and *blouse* are union variables of type *id*. Each variable can represent either a 12-element character string (*colour*) or an integer quantity (*size*) at any one time.

It is possible to combine the declaration of union composition with that of the union variables, as shown below.

```
storage-class union tag
{   member1;
    member2;
    .....
```

```
    membern;
} var1, var2, ..., varn;
```

The *tag* is optional in this situation.

The following single declaration is equivalent to the two declarations presented in the previous example.

```
union id
{   char colour[12];
    int size;
}shirt, blouse;
```

Thus, the *shirt* and *blouse* are union variables of type *id*.

Since the variable declarations are now combined with the declaration of the union type, the *tag* (i.e., *id*) need not be included. Thus, the above declaration can also be written as

```
union
{   char colour[12];
    int size;
}shirt, blouse;
```

A union may be a member of a structure, and a structure may be member of union.

For example, consider the following declarations.

```
union id
{   char colour[12];
    int size;
};
struct clothes
{   char manufacturer[20];
    float cost;
    union id description;
} shirt, blouse;
```

Now shirt and blouse are structures variable of type clothes. Each variable will contain the following members: a string (manufacturer), a floating-point quantity (*cost*), and a union (description). The union may represent either a string (*colour*) or an integer quantity.

An individual union member can be accessed in the same manner as an individual structure member, using the operators . and ->. Thus, if *var* is a union variable, then *var.member* refers to a member of the union. Similarly, if *ptrvar* is pointer variable that points to a union, then *ptrvar->member* refers to a member of that union.

7. BIT FIELDS

In some applications it may be desirable to work with data items that consist of only a few bits (e.g., a single bit flag to indicate a *true/false* condition, a 3-bit integer whose values can range from 0 through 7, or a 7-bit ASCII character.) Several such data items can be packed into an individual word of memory. To do so, the word is subdivided into individual *bitfields*. These bit fields are defined as members of a structure. Each bit field can then be accessed individually, like any other member of a structure.

In general terms, the decomposition of a word into distinct bit fields can be written as

```
struct tag
{   member 1 ;
    member 2;
    . . . . .
    member m;
};
```

where the individual elements have the same meaning as in a structure declaration. Each member declaration must now include a specification indicating the size of the corresponding bit field. To do so, the member name must be followed by a colon and an unsigned integer indicating the field size.

The interpretation of these bit fields may vary from one C compiler to another. For example, some C compilers may order the bit fields from right to left, whereas other C compilers will order them from left to right. We assume right-to-left ordering in the examples shown below.

A C program contains the following declarations.

```
struct sample
{   unsigned a : 1;
    unsigned b : 3;
    unsigned c : 2;
    unsigned d : 1;
};
struct sample v;
```

The first declaration defines a structure which is subdivided into four bit fields, called *a*, *b*, *c* and *d*. These bit fields have widths of 1 bit, 3 bits, 2 bits and 1 bit, respectively. Hence, the bit fields occupy a total of 7 bits within a word of memory. Any additional bits within the word will remain uncommitted.

The following figure illustrates the layout of the bit fields within the word, assuming a 16-bit word with the fields ordered from right to left.

days of the week and colums representing cities. Write a program to access the temperature readings of the cities for the week and process the data to display (*a*) the highest temperature and the corresponding city, (*b*) the lowest temperature and the corresponding city, (*c*) average temperature of all the cities, (*d*) average temperature of the individual cities, (*e*) deviation of from average for each cities and (*f*) deviation of each temperature from the overall average.

123. Find the Nth Fibonacci number using recursion.

124. Solve the problem of Towers of Hanoi using recursion.

125. Perform addition of two *m*×*n* matrices using pointers and functions

126. Perform subtraction of two *m*×*n* matrices using pointers and functions

127. Perform multiplication of two *m*×*n* matrices using pointers and functions

128. Find transpose of a *m*×*n* matrice using pointers and functions

129. Perform addition of two two complex numbers using structure

130. Perform subtraction of two two complex numbers using structure

8.1. ANSWERS TO SELECT PROGRAMMING EXERCISES

5.

```
/* Program to display Prime numbers up to a given number */
#include<stdio.h>
#include<math.h>
int prime(int x);
main()
{ int n,i;
  printf("Up to which number ? ");
  scanf("%d",&n);
  if(n<0)
    printf("Invalid Input");
  else
  { printf("The Prime numbers up to %d are:\n",n);
    for(i=2;i<=n;++i)
      if(prime(i))
        printf("%d\t",i);
  }
}
int prime(int x)
{  int flag = 0,i;
  for(i=2;i<=sqrt(x);++i)
    if(x%i==0)
      return flag;
 flag = 1;
return flag;
```



```
}
```

7.

```
/* First 'n' Prime numbers */
#include<stdio.h>
#include<math.h>
int prime(int x);
main()
{ int n,i;
  printf("Up to which number ? ");
  scanf("%d",&n);
  if(n<0)
    printf("Invalid Input");
  else
  { printf("The first %d Prime numbers are:\n",n);
    for(i=1;i<=n;++i)
        printf("%d\t",prime(i));
  }
}
int prime(int x)
{  int i,j,count=1,flag;
  for(i=2;count<=x;++i)
  { flag = 1;
    for(j=2;j<=sqrt(i);++j)
      if(i%j==0)
        { flag = 0;
          break;
        }
    if(flag)
      count++;
  }
  i--;
  return i;
}
```

11.

```
/* To display the Armstrong numbers within a range */
#include<stdio.h>
#include<math.h>
int Armstrong(int n);
main()
{ int lower,upper,temp;
  printf("Enter a range \n");
  printf("\tLower Bound ? ");
  scanf("%d",&lower);
```

```
    printf("\tUpper Bound ? ");
    scanf("%d",&upper);
    if(lower>upper)
    { temp=lower;
      lower=upper;
      upper=temp;
    }
    printf("Armstrong Numbers between %d and %d are:\n",lower,upper);
    for(++lower;lower<upper;++lower)
      if(Armstrong(lower))
        printf("%d\t",lower);
}
int Armstrong(int n)
{ int num,sum=0,digit;
  num=n;
  while(num>0)
  { digit=num % 10;
    sum+=pow(digit,3);
    num/=10;
  }
  if(sum==n)
    return 1;
  else
    return 0;
}
```

21.

```
/* To display the sum of 1!+2!+3!+....+n! */
#include <stdio.h>
double fact(int n);
main()
{ int n,i;
  double sum=0.0;
  printf("Up to which term ? ");
  scanf("%d",&n);
  for(i=1;i<=n;++i)
    sum+=fact(i);
  printf("Sum of the factorials of the numbers up to %d is %g",n,sum);
}
double fact(int n)
{ int i;
  double f = 1.0;
  for(i=2;i<=n;++i)
    f*=i;
  return f;
}
```


23.

```
/* To Sum the Cosine series */
#include<stdio.h>
#include<math.h>
double fact(int n);
double power(float x, int n);
main()
{ int i;
  float sign,x,tempx;
  double sum=1.0,term=1.0;
  printf("Enter the angle in degree : ");
  scanf("%f",&x);
  tempx=x;
  x=x*3.14/180.0;
  sign = -1;
  printf("The terms are:\n");
  for(i=2;fabs(term)>0.0001;i+=2)
  { printf("%g\t",term);
    term=sign*power(x,i)/fact(i);
    sum+=term;
    sign = -sign;
  }
  printf("\n\nCos(%g) = %.2f",tempx,sum);
}
double fact(int n)
{ int i;
  double f = 1.0;
  for(i=2;i<=n;++i)
    f*=i;
  return f;
}
double power(float x, int n)
{ double p=1,i;
  for(i=1;i<=n;++i)
    p=p*x;
  return p;
}
```

28.

```
/* To display the palindrome numbers */
#include<stdio.h>
int palindrome(int n);
main()
{ int i;
```

```
    printf("Palindrome numbers from 1 to 10000 are\n");
    for(i=1;i<=10000;i++)
      if(palindrome(i))
        printf("%d\t",i);
  }
  int palindrome(int n)
  { int i, rev=0, digit,temp;
    temp = n;
    while(temp>0)
    { digit = temp % 10;
      rev=rev*10+digit;
      temp/=10;
    }
    if(rev==n)
      return 1;
    else
      return 0;
  }
```

38.

```
/* To display Perfect, Abundant and Deficient numbers */
#include<stdio.h>
int test(int n);
main()
{ int i,lower,upper,temp;
  printf("Enter a range \n");
  printf("\tLower Bound ? ");
  scanf("%d",&lower);
  printf("\tUpper Bound ? ");
  scanf("%d",&upper);
  if(lower>upper)
  { temp=lower;
    lower=upper;
    upper=temp;
  }
  printf("Number\tType\n");
  for(++lower;lower<upper;lower++)
    switch(test(lower))
    { case 0: printf("%d\tPerfect\n",lower); break;
      case 1: printf("%d\tAbundant\n",lower); break;
      case 2: printf("%d\tDeficient\n",lower);
    }
}
int test(int n)
{ int i, sum=0;
  for(i=2;i<n;++i)
```



```
    if(n%i==0)
      sum += i;
  printf(" %d ",sum);
  getchar();
  if(sum==;
  else
    if(sum>n)
      return 1;
    else
      return 2;
}
```

40.

```
/* To find and display largest Even and Odd */
#include<stdio.h>
main()
{ int i,n,num[50],oddlarge,evenlarge, evenflag=0, oddflag=0;
  printf("How many numbers ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid data");
    else
    {
      for(i=0;i<n;i++)
      { printf("Enter Number-%d : ", i+1);
        scanf("%d", &num[i]);
      }
      for(i=0;i<n;i++)
      {   if(num[i]%2 == 0)
          {   if(!(evenflag))
              {   evenlarge=num[i];
                  evenflag=1;
              }
              else
                if(evenlarge< num[i])
                    evenlarge = num[i];
          }
          else
          { if(!(oddflag))
              {   oddlarge=num[i];
                  oddflag=1;
              }
              else
                if(oddlarge<num[i])
                    oddlarge=num[i];
          }
```

```
    }
    printf("\nThe numbers you entered are :\n");
    for(i=0;i<n;i++)
    printf("%d\t",num[i]);
    if(evenflag)
      printf("\nThe largest Even number is %d\n",evenlarge);
    else
      printf("\nNo Even numbers in your set of numbers\n");
    if(oddflag)
      printf("The largest Odd number is %d\n",oddlarge);
    else
      printf("No Odd numbers in your set of numbers\n");
  }
}
```

42.

```
/* To calculate and display sum and average of +ve and -ve numbers */
#include<stdio.h>
main()
{ int i,n,poscount=0,negcount=0;
  float num[50],negsum=0.0, possum=0.0,negavg,posavg;
  printf("How many numbers ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid data");
  else
  { for(i=0;i<n;i++)
    { printf("Enter Number-%d : ", i+1);
      scanf("%f", &num[i]);
      if(num[i] < 0)
      { negsum+=num[i];
      negcount++;
      }
      if(num[i] > 0)
      { possum+=num[i];
        poscount++;
      }
  }
  printf("\nThe numbers you entered are :\n");
  for(i=0;i<n;i++)
    printf("%g\t",num[i]);
  if(negcount)
  { negavg = negsum/negcount;
    printf("\nSum of negative numbers is %g",negsum);
    printf("\nAverage of negative numbers is %g",negavg);
  }
```



```
  else
    printf("\nNo negative numbers exist in your set of numbers");
  if(poscount)
  { posavg = possum/poscount;
    printf("\nSum of positive numbers is %g",possum);
    printf("\nAverage of positive numbers is %g",posavg);
  }
  else
    printf("\nNo positive numbers exist in your set of numbers");
  }
}
```

43.

```
/* To calculate and display smallest and largest difference */
#include<stdio.h>
main()
{ int i,j,n;
  float num[50],temp;
  printf("How many numbers ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid data");
  else
  { for(i=0;i<n;i++)
    { printf("Enter Number-%d : ", i+1);
      scanf("%f", &num[i]);
    }
    printf("\nThe numbers you entered are :\n");
    for(i=0;i<n;i++)
      printf("%g \t",num[i]);
    for(i=0;i<n-1;i++)
      for(j=i+1;j<n;++j)
        if(num[i]>num[j])
        { temp = num[i];
          num[i]=num[j];
          num[j]=temp;
        }
    printf("\nThe largest difference is %g",num[n-1]-num[0]);
    printf("\nCorresponding elements are %g and %g",num[n-1],num[0]);
    printf("\nThe Smallest difference is %g",num[0]-num[n-1]);
    printf("\nCorresponding elements are %g and %g",num[0],num[n-1]);
  }
}
```

44.

```
/* To Search a number in a set of numbers */
#include<stdio.h>
main()
{ int i,n,count=0;
  float num[50],search;
  printf("How many numbers ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid data");
  else
  { for(i=0;i<n;i++)
    { printf("Enter Number-%d : ", i+1);
      scanf("%f", &num[i]);
    }
    printf("\n\n\nNumber to search ? ");
    scanf("%f",&search);
    printf("\nThe numbers you entered are :\n");
    for(i=0;i<n;i++)
      printf("%g \t",num[i]);
    printf("\n\n\n");
    for(i=0;i<n;i++)
      if(num[i] == search)
      { printf("Location-%d\n",i);
        count++;
      }
    if(count)
      printf("\nThe number %g exists in %d locations shown above", search,count);
    else
      printf("\nThe number %g does not exists in the set",search);
  }
}
```

47.

```
/* To insert a number into a set of numbers*/
#include<stdio.h>
main()
{ int i,n,loc;
  float num[50],insert,temp1,temp2;
  printf("How many numbers ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid data");
  else
  { for(i=0;i<n;i++)
```



```
    { printf("Enter Number-%d : ", i+1);
      scanf("%f", &num[i]);
    }
    printf("\n\n\nNumber to inserted ? ");
    scanf("%f",&insert);
    printf("\n\n\nLocation ? ");
    scanf("%d",&loc);
    if(location<0 || location >n)
      printf("Invalid location");
    else
    {
      printf("\nThe numbers you entered are :\n");
      for(i=0;i<n;i++)
        printf("%g \t",num[i]);
      temp1=num[loc];
      num[loc]=insert;
      for(i=loc+1;i<=n;i++)
      { temp2=num[i];
        num[i]=temp1;
        temp1=temp2;
      }
      printf("\nThe numbers after insertion :\n");
      for(i=0;i<=n;i++) printf("%g \t",num[i]);
    }
  }
}
```

49.

```
/* To merge two array into a third array */
#include<stdio.h>
void sort(float a[], int n);
void display(float a[], int n);
void readnum(float a[], int n);
main()
{ int i,j,k,n1,n2;
  float num1[50],num2[50],num3[100];
  printf("How many elements in First set ? ");
  scanf("%d",&n1);
  if(n1<=0)
    printf("Invalid data");
  else
  { printf("Enter the elements of the First set :\n");
    readnum(num1,n1);
    printf("How many elements in Second set ? ");
    scanf("%d",&n2);
    if(n2<=0)
```

```
        printf("Invalid data");
      else
      { printf("Enter the elements of the Second set :\n");
        readnum(num2,n2);
        sort(num1,n1);
        sort(num2,n2);
        for(i=0,j=0,k=0;i<(n1+n2);i++)
        { if((j<n1)&&(k<n2))
            if(num1[j]<num2[k])
              num3[i]=num1[j++];
            else
              num3[i]=num2[k++];
          else
            if(k==n2)
              num3[i]=num1[j++];
            else
              if(j==n1)
                num3[i]=num2[k++];
        }
        printf("Your First set is :\n");
        display(num1,n1);
        printf("Your Second set is :\n");
        display(num2,n2);
        printf("The Merged set is :\n");
        display(num3,n1+n2);
      }
    }
  }
  /* Function to sort an array */
  void sort(float a[], int n)
  { float temp;
    int i,j;
    for(i=0;i<n-1;i++)
      for(j=i+1;j<n;++j)
        if(a[i]>a[j])
        { temp=a[i];
          a[i]=a[j];
          a[j]=temp;
        }
    return;
  }
  /* Function to display an array */
  void display(float a[], int n)
  { int i;
    for(i=0;i<n;++i)
      printf("%g\t",a[i]);
```



```
    printf("\n\n\n");
    return;
  }
  /* Function to read an array */
  void readnum(float a[], int n)
  { int i;
    for(i=0;i<n;++i)
    { printf("Number-%d ? " ,i+1);
      scanf("%f",&a[i]);
    }
    return;
  }
```

51.

```
/* To display the positions of the coordinates */
#include<stdio.h>
main()
{ int i,n;
  float x[50],y[50];
  printf("How many Coordinates ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid data");
  else
  { for(i=0;i<n;i++)
    { printf("Enter Set - %d\n",i+1);
      printf("\tX-axis value ? ");
      scanf("%f", &x[i]);
      printf("\tY-axis value ? ");
      scanf("%f", &y[i]);
    }
    printf("\nCoordinates\tPosition");
    printf("\n----------\t---------\n");
    for(i=0;i<n;i++)
    { printf("\n(%g,%g)\t\t",x[i],y[i]);
      if((x[i]==0)&&(y[i]==0))
        printf("Origin");
      else
        if((x[i]==0)&&(y[i]>0))
          printf("+Ve Side of Y-axis");
        else
          if((x[i]==0)&&(y[i]<0))
            printf("-Ve Side of Y-axis");
          else
            if((x[i]<0)&&(y[i]==0))
              printf("-Ve Side of X-axis");
```

```
            else
              if((x[i]>0)&&(y[i]==0))
                printf("+Ve Side of X-axis");
              else
                if((x[i]>0)&&(y[i]>0))
                  printf("I st Quadrant");
                else
                  if((x[i]<0)&&(y[i]>0))
                    printf("II nd Quadrant");
                  else
                    if((x[i]<0)&&(y[i]<0))
                      printf("III rd Quadrant");
                    else
                      printf("IV th Quadrant");
    }
  printf("\n----------\t---------\n");
  }
}
```

```
  return;
}
```

71.

```
/* To find the Transpose of matrix */
#include<stdio.h>
void readmat(int x[][10], int r, int c);
void printmat(int x[][10], int r, int c);
void transpose(int x[][10],int y[][10],int r,int c);
```

```
main()
{ int mat[10][10],tra[10][10],row,col;
  printf("\tEnter the Order of the matrix\n");
  printf("\n\n\t\tNumber of rows ? ");
  scanf("%d",&row);
  printf("\t\tNumber of columns ? ");
  scanf("%d",&col);
  if((row<=0) || (col<=0))
    printf("Invalid Order");
  else
  {
    printf("Enter the elements of the Matrix :\n");
    readmat(mat,row,col);
    printf("\tYour Matrix\n");
    printf("\t------------\n");
    printmat(mat,row,col);
    transpose(mat,tra,row,col);
    printf("\tTranspose of the Matrix\n");
    printf("\t--------------------\n");
    printmat(tra,col,row);
  }
}
/* Function for reading matrix */
void readmat(int x[][10],int r,int c)
{ int i,j;
  for(i=0;i<r;++i)
    for(j=0;j<c;++j)
    { printf("Element (%d,%d) ? ",i+1,j+1);
      scanf("%d",&x[i][j]);
    }
  return;
}
/* Function for displaying matrix */
void printmat(int x[][10],int r,int c)
{ int i,j;
  for(i=0;i<r;++i)
  { for(j=0;j<c;++j)
      printf("%6d",x[i][j]);
    printf("\n");
  }
  return;
}
void transpose(int x[][10],int y[][10],int r,int c)
{ int i,j;
  for(i=0;i<c;++i)
    for(j=0;j<r;++j)
```



```
    y[i][j]=x[j][i];
  return;
}
```

73.

```
/* To generate the Pascal's Triangle */
#include<stdio.h>
void generate(int x[][100], int r);
void print(int x[][100], int r);
main()
{ int tria[100][100],n;
  printf("How many lines ? ");
  scanf("%d",&n);
  if(n<=0)
    printf("Invalid Data");
  else
  {
    generate(tria,n);
    printf("\tFirst %d lines of Pascal's Triangle\n",n);
    print(tria,n);
  }
}
/* Function for generating the Pascal's triangle*/
void generate(int x[][100],int r)
{ int i,j;
  for(i=0;i<r;++i)
  { x[i][0]=1;
    for(j=1;j<=i-1;++j)
      x[i][j]=x[i-1][j]+x[i-1][j-1];
    x[i][j]=1;
  }
  return;
}
/* Function for displaying the Pascal's triangle */
void print(int x[][100],int r)
{ int i,j;
  for(i=0;i<r;++i)
  { for(j=0;j<=i;++j)
      printf("%6d",x[i][j]);
    printf("\n");
  }
  return;
}
```

74.

```
/* To convert the Roman number into equivalent decimal */
#include<stdio.h>
#include<ctype.h>
int value (char dig);
main()
{ char roman[10];
  int val1,val2,decequiv=0,i=0;
  printf("Enter the Roman Number : ");
  scanf("%[^\n]",roman);
  val1 = value(roman[i]);
  for(i=1;roman[i] != '\0';++i)
  {  val2 = value(roman[i]);
    if((val1==0) || (val2 ==0))
      { printf("Invalid Character in Roman Number");
        getchar();
        exit(0);
      }
    if(val1>=val2)
      decequiv+=val1;
    else
      decequiv-=val1;
    val1=val2;
  }
  decequiv+=val1;
  printf("Decimal equivalent of %s is %d",roman,decequiv);
}
/* Function for generating value of roman digit*/
int value(char dig)
{ switch(toupper(dig))
  { case 'M': return(1000);break;
    case 'C': return(100); break;
    case 'L': return(50); break;
    case 'X': return(10); break;
    case 'V': return(5); break;
    case 'I': return(1); break;
    default :return(0);
  }
}
```

81.

```
/*Number conversion */
#include<stdio.h>
#include<ctype.h>
#include<string.h>
```



```
#include<math.h>
int value(char dig);
char letter(int  dig);
main()
{ int oldbase,newbase,len,i,j,d,val;
  long dec=0;
  char oldnum[100],newnum[100];
  printf("What is the Base of the number to be converted ? ");
  scanf("%d",&oldbase);
  printf("Enter Your Number : ");
  scanf("  %[^\n]",oldnum);
  printf("What is the New Base ? ");
  scanf("%d",&newbase);
  len=strlen(oldnum);
  for(i=len-1,j=0;i>=0;--i,j++)
  { val=value(oldnum[i]);
    if((val>(oldbase-1)) || (val == -1))
    {  printf("Invalid Character in Your Number ");
       getchar();
       exit(0);
    }
    dec = dec + pow(oldbase,j)*val;
  }
  for(i=0;dec>0;++i)
  { d = dec % newbase;
    newnum[i]=letter(d);
    dec = dec/newbase;
  }
  newnum[i]='\0';
  printf("Number %s in Base %d is %s in Base %d",
                    oldnum,oldbase,strrev(newnum),newbase);
}
/* Function to determine the decimal value of the character */
int value(char dig)
{ switch(toupper(dig))
    {  case '0':  return(0); break;
       case '1':  return(1); break;
       case '2':  return(2); break;
       case '3':  return(3); break;
       case '4':  return(4); break;
       case '5':  return(5); break;
       case '6':  return(6); break;
       case '7':  return(7); break;
       case '8':  return(8); break;
       case '9':  return(9); break;
       case 'A':  return(10); break;
```

```
    case 'B':   return(11); break;
    case 'C':   return(12); break;
    case 'D':   return(13); break;
    case 'E':   return(14); break;
    case 'F':   return(15); break;
    default :   return(-1);
   }
}
/* Function for character form of the digit  */
char letter(int  dig)
{ switch(dig)
   {  case 0: return('0'); break;
      case 1:  return('1'); break;
      case 2:  return('2'); break;
      case 3:  return('3'); break;
      case 4:  return('4'); break;
      case 5:  return('5'); break;
      case 6:  return('6'); break;
      case 7:  return('7'); break;
      case 8:  return('8'); break;
      case 9:  return('9'); break;
      case 10:   return('A'); break;
      case 11:   return('B'); break;
      case 12:   return('C'); break;
      case 13:   return('D'); break;
      case 14:   return('E'); break;
      case 15:   return('F'); break;
    }
  }
```