Unit 4

4.1 Array

- An array is a collection of data items, all of the same type, accessed using a common name.
- A one-dimensional array is like a list. A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.

4.1.1 Declaring Arrays

- Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array.
- Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.
- Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.

  Examples:

```
int i_Array[ 10 ];
float f_Array[ 1000 ];
```

4.1.2 Initializing Arrays

- Arrays may be initialized when they are declared, just as any other variables.
- Place the initialization data in curly {} braces following the equals sign. Note the use of commas in the examples below.
- An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.
- If an array is to be completely initialized, the dimension of the array is not required. The compiler will automatically size the array to fit the initialized data. (Variation: Multidimensional arrays - see below.)
- Examples:

```
int i_Array[ 6 ] = { 1, 2, 3, 4, 5, 6 };
float f_Array[ 100 ] = { 1.0, 5.0, 20.0 };
```

4.1.3 Using Arrays

- Elements of an array are accessed by specifying the index (offset) of the desired element within square [ ] brackets after the array name.
- Array subscripts must be of integer type.

  Very Important: Array indices start at zero in C, and go to one less than the size of the array. For example, a five element array will have indices zero through four. This is because the index in C is actually an offset from the beginning of the array. (The first element is at the beginning of the array, and hence has zero offset.)

Sample Programs Using 1-D Arrays

- The first sample program uses loops and arrays to calculate the first twenty Fibonacci numbers. Fibonacci numbers are used to determine the sample points used in certain optimization methods.

```
/* Program to calculate the first 20 Fibonacci numbers. */
#include <stdlib.h>
#include <stdio.h>

int main( void )
{

    int i, fibonacci[ 20 ];
```

```
    fibonacci[ 0 ] = 0;
    fibonacci[ 1 ] = 1;

    for( i = 2; i < 20; i++ )
      fibonacci[ i ] = fibonacci[ i - 2 ] + fibonacci[ i - 1 ];

    for( i = 0; i < 20; i++ )
       printf( "Fibonacci[ %d ] = %f\n", i, fibonacci[ i ] );

} /* End of sample program to calculate Fibonacci numbers */
```

4.2 Multidimensional Arrays

- Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.
- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.
- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns.
- Two dimensional arrays are considered by C/C++ to be an array of (single dimensional arrays). For example, "int numbers [5] [6]" would refer to a single dimensional array of 5 elements, wherein each element is a single dimensional array of 6 integers.
- C stores two dimensional arrays by rows, with all elements of a row being stored together as a single unit.
- Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.

Sample Program Using 2-D Arrays

```
/* Sample program Using 2-D Arrays */

#include <stdlib.h>
#include <stdio.h>

int main( void ) {

   /* Program to add two multidimensional arrays */

   int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
   int b[ 2 ][ 3 ] = { { 1, 2, 3 }, {  3,  2,  1 } };
   int sum[ 2 ][ 3 ], row, column;

   /* First the addition */

   for( row = 0; row < 2; row++ )
      for( column = 0; column < 3; column++ )
         sum[ row ][ column ] =
            a[ row ][ column ] + b[ row ][ column ];

   /* Then print the results */

   printf( "The sum is: \n\n" );
```

```
    for( row = 0; row < 2; row++ ) {
       for( column = 0; column < 3; column++ )
          printf( "\t%d",  sum[ row ][ column ] );
       printf( '\n' );   /* at end of each row */
    }
            return 0;
 }
```

4.3 User Defined Data Types: The C language gives you five ways to create a custom data type:

- The *structure*, which is a grouping of variables under one name and is called an *aggregate* data type. (The terms *compound* or *conglomerate* are also commonly used.)
- The *union*, which enables the same piece of memory to be defined as two or more different types of variables.
- The *bit-field*, which is a special type of structure or union element that allows easy access to individual bits.
- The *enumeration*, which is a list of named integer constants.
- The typedef keyword, which defines a new name for an existing type.

4.3.1 Structure: A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. A *structure declaration* forms a template that can be used to create structure objects (that is, instances of a structure). The variables that make up the structure are called *members*. (Structure members are also commonly referred to as *elements* or *fields*.) The following code fragment shows how to declare a structure that defines the name and address fields. The keyword **struct** tells the compiler that a structure is being declared.

```
struct addr
{
char name[30];
char street[40];
char city[20];
char state[3];
unsigned long int zip;
};
```

Notice that the declaration is terminated by a semicolon. This is because a structure declaration is a statement. Also, the structure tag **addr** identifies this particular data structure and is its type specifier. At this point, *no variable has actually been created*. Only the form of the data has been defined.

When you declare a structure, you are defining an aggregate *type*, not a variable. Not until you declare a variable of that type does one actually exist. To declare a variable (that is, a physical object) of type **addr**, write:

```
struct addr info;
```

This declares a variable of type **addr** called **addr_info**. Thus, **addr** describes the form of a structure (its type), and **addr_info** is an instance (an object) of the structure. When a structure variable (such as **addr_info**) is declared, the compiler automatically allocates Sufficient memory to accommodate all of its members. The general form of a structure declaration is

```
struct tag {
type member-name;
type member-name;
type member-name;
} structure-variables;
```

- ***Accessing Structure Members:*** Individual members of a structure are accessed through the

use of the . operator (usually called the *dot operator*). For example, the following statement assigns the ZIP code 12345 to the **zip** field of the structure variable **addr_info** declared earlier:

addr_info.zip = 12345;

The general form for accessing a member of a structure is *object-name.member-name* Therefore, to print the ZIP code on the screen, write

```
printf("%lu", addr_info.zip);
```

In the same fashion, the character array **addr_info.name** can be used in a call to **gets( )**, as shown here:

```
gets(addr_info.name);
```

- ***Structure Assignments:*** The information contained in one structure can be assigned to another structure of the same type using a single assignment statement. You do not need to assign the value of each member separately. The following program illustrates structure assignments:

```
#include <stdio.h>
int main(void)
{
struct {
int a;
int b;
} x, y;
x.a = 10;
y = x; /* assign one structure to another */
printf("%d", y.a);
return 0;
}
```

After the assignment, **y.a** will contain the value 10.

- **Arrays of Structures:** To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type **addr** defined earlier, write struct addr addr_list[100];

This creates 100 sets of variables that are organized as defined in the structure **addr**. To access a specific structure, index the array name. For example, to print the ZIP code of structure 3, write

printf("%lu", addr_list[2].zip);

Like all array variables, arrays of structures begin indexing at 0.

When you want to refer to a specific structure within an array of structures, index the structure array name. When you want to index a specific element of a structure, index the element. Thus, the following statement assigns 'X' to the first character of **name** in the third structure of **addr_list**.

addr_list[2].name[0] = 'X';

- **Passing Structures to Functions**
  - ***Passing Structure Members to Functions***

When you pass a member of a structure to a function, you are passing the value of that member to the function. It is irrelevant that the value is obtained from a member of a structure.

For example,
consider this structure:

```
struct fred
{
```

```
char x;
int y;
float z;
char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x); /* passes character value of x */
func2(mike.y); /* passes integer value of y */
func3(mike.z); /* passes float value of z */
func4(mike.s); /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

In each case, it is the value of a specific element that is passed to the function. It does not matter that the element is part of a larger unit.

If you wish to pass the *address* of an individual structure member, put the **&** operator before the structure name. For example, to pass the address of the members of the structure **mike**, write

```
func(&mike.x); /* passes address of character x */
func2(&mike.y); /* passes address of integer y */
func3(&mike.z); /* passes address of float z */
func4(mike.s); /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

Note that the **&** operator precedes the structure name, not the individual member name.

- ***Passing Entire Structures to Functions:*** When a structure is used as an argument to a function, the entire structure is passed using the normal call-by-value method. Of course, this means that any changes made to the contents of the parameter inside the function do not affect the structure passed as the argument.

  When using a structure as a parameter, remember that the type of the argument must match the type of the parameter. For example, in the following program both the argument **arg** and the parameter **parm** are declared as the same type of structure.

```
#include <stdio.h>
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
printf("%d", parm.a);
}
```

  As this program illustrates, if you will be declaring parameters that are structures, you must make the declaration of the structure type global so that all parts of your program can use it. For example, had **struct_type** been declared inside **main( )**, it would not have been visible to **f1( )**.

When a structure is a member of another structure, it is called a *nested structure*. For example, the structure **address** is nested inside **emp** in this example:

```
struct emp {
struct addr address; /* nested structure */
float wage;
} worker;
```

Here, structure **emp** has been defined as having two members. The first is a structure of type **addr**, which contains an employee's address. The other is **wage**, which holds the employee's wage. The following code fragment assigns 93456 to the **zip** element of **address**.

```
worker.address.zip = 93456;
```

4.7 Unions

A *union* is a memory location that is shared by two or more different types of variables. A union provides a way of interpreting the same bit pattern in two or more different ways. Declaring a **union** is similar to declaring a structure. Its general form is

```
union tag {
type member-name;
type member-name;
type member-name;
.
.
.
} union-variables;
```

For example:

```
union u_type {
int i;
char ch;
};
```

This declaration does not create any variables. You can declare a variable either by placing its name at the end of the declaration or by using a separate declaration statement. To declare a **union** variable called **cnvt** of type **u_type** using the definition just given, write

```
union u_type cnvt;
```

In **cnvt**, both integer **i** and character **ch** share the same memory location. Of course, **i** occupies 2 bytes (assuming 2-byte integers), and **ch** uses only 1.

When a **union** variable is declared, the compiler automatically allocates enough storage to hold the largest member of the **union**. For example, (assuming 2-byte integers) **cnvt** is 2 bytes long so that it can hold **i**, even though **ch** requires only 1 byte.

To access a member of a **union**, use the same syntax that you would use for structures: the dot and arrow operators. If you are operating on the **union** directly, use the dot operator. If the **union** is accessed through a pointer, use the arrow operator. For example, to assign the integer 10 to element **i** of **cnvt**, write

```
cnvt.i = 10;
```

4.8 Enumerations

An *enumeration* is a set of named integer constants. Enumerations are common in everyday life. For example, an enumeration of the coins used in the United States is penny, nickel, dime, quarter, half-dollar, dollar.

Enumerations are defined much like structures; the keyword **enum** signals the start of an enumeration type. The general form for enumerations is

```
enum tag { enumeration list } variable_list ;
```

Here, both the tag and the variable list are optional. (But at least one must be present.) The

following code fragment defines an enumeration called **coin**:
enum coin { penny, nickel, dime, quarter, half_dollar, dollar};
The enumeration tag name can be used to declare variables of its type. The following declares **money** to be a variable of type **coin**:
enum coin money;
Given these declarations, the following types of statements are perfectly valid:
money = dime;
if(money==quarter)
printf("Money is a quarter.\n");
The key point to understand about an enumeration is that each of the symbols stands for an integer value. As such, they can be used anywhere that an integer can be used. Each symbol is given a value one greater than the symbol that precedes it. The value of the first enumeration symbol is 0. Therefore,
printf("%d %d", penny, dime);
displays **0 2** on the screen.
You can specify the value of one or more of the symbols by using an initializer. Do this by following the symbol with an equal sign and an integer value. Symbols that appear after an initializer are assigned values greater than the preceding value. For example, the following code assigns the value of 100 to **quarter**:
enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar};
Now, the values of these symbols are
penny 0
nickel 1
dime 2
quarter 100
half_dollar 101
dollar 102

4.9 Strings
The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings.
A string constant is a one-dimensional array of characters terminated by a null ('\0'). For example,
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
Each character in the array occupies one byte of memory and the last character is always '\0'. '\0' is called null character. A string not terminated by a '\0' is not really a string, but merely a collection of characters.
/* Program to demonstrate printing of a string */
main( )
{
char name[ ] = "Klinsman" ;
int i = 0 ;
while (i <= 7)
{
printf ("%c", name[i]) ;
i++ ;
}
And here is the output...
Klinsman
Can we write the **while** loop without using the final value 7? We can; because we know that each character array always ends with a '\0'. Following program illustrates this.
main( )
{

```
char name[ ] = "Klinsman" ;
int i = 0 ;
while ( name[i] != `\0' )
{
printf ( "%c", name[i] ) ;
i++ ;
}
}
```
And here is the output...
Klinsman

Here is another version of the
same program; this one uses a pointer to access the array elements.

```
main( )
{
char name[ ] = "Klinsman" ;
char *ptr ;
ptr = name ; /* store base address of string */
while ( *ptr != `\0' )
{
printf ( "%c", *ptr ) ;
ptr++ ;
}
}
```

The **%s** used in **printf( )** is a format specification for printing out a string. The same specification can be used to receive a string from the keyboard, as shown below.

```
main( )
{
char name[25] ;
printf ( "Enter your name " ) ;
scanf ( "%s", name ) ;
printf ( "Hello %s!", name ) ;
}
```
And here is a sample run of the program...
Enter your name Debashish
Hello Debashish!

Note that the declaration **char name[25]** sets aside 25 bytes under the array **name[ ],** whereas the **scanf( )** function fills in the characters typed at keyboard into this array until the enter key is hit. Once enter is hit, **scanf( )** places a '\0' in the array.
While entering the string using **scanf( )** we must be cautious about two things:
a) The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays. Hence, if you carelessly exceed the bounds there is always a danger of overwriting something important, and in that event, you would have nobody to blame but yourselves.
b) scanf( ) is not capable of receiving multi-word strings. Therefore names such as 'Debashish Roy' would be unacceptable. The way to get around this limitation is by using the function **gets( )**. The usage of functions **gets( )** and its counterpart **puts( )** is shown below.

```
main( )
{
```

```
char name[25] ;
printf ( "Enter your full name " ) ;
gets ( name ) ;
puts ( "Hello!" ) ;
puts ( name ) ;
}
```
And here is the output...
```
Enter your name Debashish Roy
Hello!
Debashish Roy
```
unlike **printf( )**, **puts( )** places the cursor on the next line.

4.10 Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a **char** pointer. This is shown below:
```
char str[ ] = "Hello" ;
char *p = "Hello" ;
```
There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a **char** pointer to another **char** pointer. This is shown in the following program.
```
main( )
{
char str1[ ] = "Hello" ;
char str2[10] ;
char *s = "Good Morning" ;
char *q ;
str2 = str1 ; /* error */
q = s ; /* works */
}
```
Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with **char** pointers.

```
main( )
{
char str1[ ] = "Hello" ;
char *p = "Hello" ;
str1 = "Bye" ; /* error */
p = "Bye" ; /* works */
}
```

4.11 Standard Library String Functions

With every C compiler a large set of useful string handling library functions are provided. Figure 9.2 lists the more commonly used functions along with their purpose.

Function	**Use**
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another

strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

Out of the above list we shall discuss the functions **strlen( )**, **strcpy( )**, **strcat( )** and **strcmp( )**, since these are the most commonly used functions.

strlen( )

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

```
main( )
{
char arr[ ] = "Bamboozled" ;
int len1, len2 ;
len1 = strlen ( arr ) ;
len2 = strlen ( "Humpty Dumpty" ) ;
printf ( "\nstring = %s length = %d", arr, len1 ) ;
printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

The output would be...

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

Note that in the first call to the function **strlen( )**, we are passing the base address of the string, and the function in turn returns the length of the string. While calculating the length it doesn't count '\0'. Even in the second call,

```
len2 = strlen ( "Humpty Dumpty" ) ;
```

what gets passed to **strlen( )** is the address of the string and not the string itself. Can we not write a function **xstrlen( )** which imitates the standard library function **strlen( )**? Let us give it a try...

```
/* A look-alike of the function strlen( ) */
main( )
{
char arr[ ] = "Bamboozled" ;
int len1, len2 ;
len1 = xstrlen ( arr ) ;
len2 = xstrlen ( "Humpty Dumpty" ) ;
printf ( "\nstring = %s length = %d", arr, len1 ) ;
printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

```
xstrlen ( char *s )
{
int length = 0 ;
while ( *s != '\0' )
{
length++ ;
s++ ;
}
return ( length ) ;
}
```
The output would be...
```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

strcpy( )
This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of **strcpy( )** in action...
```
main( )
{
char source[ ] = "Sayonara" ;
char target[20] ;
strcpy ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
```
And here is the output...
```
source string = Sayonara
target string = Sayonara
```

Let us now attempt to mimic **strcpy( )**,via our own string copy function, which we will call **xstrcpy()**
```
main( )
{
char source[ ] = "Sayonara" ;
char target[20] ;
xstrcpy ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
xstrcpy ( char *t, char *s )
{
while ( *s != '\0' )
{
*t = *s ;
s++ ;
t++ ;
}
*t = '\0' ;
}
```
The output of the program would be...
```
source string = Sayonara
target string = Sayonara
```

Note that having copied the entire source string into the target string, it is necessary to place a '\0' into the target string, to mark its end.

strcat( )
This function concatenates the source string at the end of the target string. For example, "Bombay" and "Nagpur" on concatenation would result into a string "BombayNagpur". Here is an example of **strcat( )** at work.

```
main( )
{
char source[ ] = "Folks!" ;
char target[30] = "Hello" ;
strcat ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}
```

And here is the output...

```
source string = Folks!
target string = HelloFolks!
```

strcmp( )
This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, **strcmp( )** returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts **strcmp( )** in action.

```
main( )
{
char string1[ ] = "Jerry" ;
char string2[ ] = "Ferry" ;
int i, j, k ;
i = strcmp ( string1, "Jerry" ) ;
j = strcmp ( string1, string2 ) ;
k = strcmp ( string1, "Jerry boy" ) ;
printf ( "\n%d %d %d", i, j, k ) ;
}
```

And here is the output...

```
0 4 -32
```

In the first call to **strcmp( )**, the two strings are identical—"Jerry" and "Jerry"—and the value returned by **strcmp( )** is zero. In the second call, the first character of "Jerry" doesn't match with the first character of "Ferry" and the result is 4, which is the numeric difference between ASCII value of 'J' and ASCII value of 'F'. In the third call to **strcmp( )** "Jerry" doesn't match with "Jerry boy", because the null character at the end of "Jerry" doesn't match the blank in "Jerry boy". The value returned is -32, which is the value of null character minus the ASCII value of space, i.e., '\0' minus ' ', which is equal to -32.

4.11 Two-Dimensional Array of Characters/ Array of Strings
In Array of Strings each row is used to hold a separate string.
Our example program asks you to type your name. When you do so, it checks your name against a master list to see if you are worthy of entry to the palace. Here's the program...

```
#define FOUND 1
#define NOTFOUND 0
```

```
main( )
{
char masterlist[6][10] = {
"akshay",
"parag",
"raman",
"srinivas",
"gopal",
"rajesh"
} ;
int i, flag, a ;
char yourname[10] ;
printf ( "\nEnter your name " ) ;
scanf ( "%s", yourname ) ;
flag = NOTFOUND ;
for ( i = 0 ; i <= 5 ; i++ )
{
a = strcmp ( &masterlist[i][0], yourname ) ;
if ( a == 0 )
{
printf ( "Welcome, you can enter the palace" ) ;
flag = FOUND ;
break ;
}
}
if ( flag == NOTFOUND )
printf ( "Sorry, you are a trespasser" ) ;
}
```

And here is the output for two sample runs of this program...

```
Enter your name dinesh
Sorry, you are a trespasser
Enter your name raman
Welcome, you can enter the palace
```

The order of the subscripts in the array declaration is important. The first subscript gives the number of names in the array, while the second subscript gives the length of each item in the array. Instead of initializing names, had these names been supplied from the keyboard, the program segment would have looked like this...

```
for ( i = 0 ; i <= 5 ; i++ )
scanf ( "%s", &masterlist[i][0] ) ;
```

Questions

From 2011-12

1. Write a program in 'C' to read a (5×4) matrix using array and to calculate the following:

   i) Sum of the elements of the third row of the matrix.
   ii) Sum of all the elements of the matrix.
2. Write algorithm and function program to sort an array of integer into descending order, where the size of array is input by user.
3. Define a structure called **cricket** that will describe the following information:
   player name
   team name
   batting average

Using **cricket**, declare an array **player** with 50 elements and write a program to read the information about all the 50 players and print a team-wise list containing names of players with their batting average.

4. Write a program which will read a string and rewrite it in the alphabetical order. For example the word **STRING** should be written as **GINRST**.

From 2012-13

1. List the differences between structure and union.
2. Explain in detail about 1-Dimensional and 2-Dimensional array declaration, accessing elements, initialization with suitable examples. Write a program to multiply two N×N matrix.
3. Discuss String handling function.
4. Write a program in C that accepts the Rollno and name of 60 students in a class along with their marks in Physics,
5. Chemistry and Mathematics. Print the roll no. and name of top 10 students on the sum of marks obtained in the three subjects.
6. Write a program to define a structure named employee having empid, name and designation. Use this structure to store the data in a file named "employee.dat". Once all the records are entered display the employee details stored in a file.
7. Write a program to create an integer array of n elements, pass this array as an argument to a function where it is sorted and displayed.
8. Two matrices of real numbers of size 4 × 4 is given. Write the functions sum_matrix() and multiply_matrix() for displaying the addition and multiplication of the given matrices in C language.
9. Create a suitable structure in C language for keeping the records of the employees of an organization about their Code, Name, Designation, Salary, Department, City of posting. Also write a program in C to enter the records of 200 employees and display the names of those who earn greater than Rs 50,000. (Make suitable assumptions for data types).
10. The marks of the N students in a given subject (Minimum mark 0 and Maximum mark 100) is awarded. Write a program in C language to store these marks in an array and calculate the average mark obtained by all the students and then display the deviation of mark for each student from average.
11. What do you mean by sorting. Write an algorithm to sort the given elements in ascending order.

From 2013-14

1. Declare and initialize three dimensional array.
2. Write different between structure and array. Write a program in 'C' to find the largest element of a 3 × 3 matrix.
3. Define Union. Write a program in C to find the record of student having maximum marks from the list of 10 records. Each record has roll no, name, class and marks fields.
4. Write a program in 'C' to multiply the two matrices of M × N.

From 2014-15

1. Write a C program to find the multiplication of two matrices.
2. How to declare an array? Explain about various operations of an array.
3. What is enumerated data type? Write a C program to display months in the year using union.
4. Differentiate structure and union in C. Write a C program to store the student's details using union.