

Problem Solving with Programming Course Material

PROBLEM SOLVING WITH PROGRAMMING

Year	Semester	Hours/Week			C	Marks		
		L	T	P/D		CIE	SEE	Total
I	II	2	-	-	2	30	70	100
Pre-requisite		Nil						

COURSE OUTCOMES

At the end of the course, the students will develop ability to

1. Analyze and implement software development tools like algorithm, pseudo codes and programming structure.
2. Modularize the problems into small modules and then convert them into modular programs
3. Apply the pointers, memory allocation techniques and use of files for dealing with variety of problems.
4. Apply C programming to solve problems related to scientific computing.
5. Develop efficient programs for real world applications.

UNIT I

Pointers

Basics of pointers, pointer to array, array of pointers, void pointer, pointer to pointer- example programs, pointer to string.

Project: Simple C project by using pointers.

UNIT II

Structures

Basics of structure in C, structure members, accessing structure members, nested structures, array of structures, pointers to structures - example programs, Unions- accessing union members- example programs.

Project: Simple C project by using structures/unions.

UNIT III

Functions

Functions: User-defined functions, categories of functions, parameter passing in functions: call by value, call by reference, recursive functions. passing arrays to functions, passing strings to functions, passing a structure to a function.

Problem Solving with Programming Course Material

Project: Simple C project by using functions.

UNIT IV

File Management

Data Files, opening and closing a data file, creating a data file, processing a data file, unformatted data files.

Project: Simple C project by using files.

UNIT V

Memory Management

Memory Management: Dynamic memory allocation and deallocation functions:- malloc(), calloc(), realloc() and free()-examples & discussion for each. Low-level programming, register variables, bitwise operations, bit fields.

UNIT VI

Developing Large Programs

Some Additional features of C, enumerations, command line parameters, more about library functions, macros, C preprocessor. Pre-processor directives: #define, #undef, #if, #endif, #elif, #ifdef, #ifndef, #error.

TEXT BOOKS

1. Computer Science: A Structured Programming Approach Using C- B. A. Forouzan and R.F. Gilberg, Third Edition, Cengage Learning'
2. B.W.Kernighan Dennis M. Ritchie, The C Programming Language, PHI/Pearson Education,ISBN:0-13-110362-8

REFERENCE BOOKS

1. C Programming: A Modern Approach by K.N. King .
2. Let us C by Yashwant Kanetkar. 13th edition, BPB Publications
3. Computer science a structured programming approach using C by Pradeep K.Sinha, Priti Sinha, 3rd edition, Thomson publications.
4. Programming Embedded Systems by Michael Barr and Anthony Massa, 2nd

Problem Solving with Programming Course Material

UNIT I: Topics Included

Pointers

Basics of pointers, pointer to array, array of pointers, void pointer, pointer to pointer- example programs, pointer to string.

Project: Simple C project by using pointers.

Pointers Overview :

Pointers are an important feature of the C language. To understand pointers let us revise certain points about computer memory. You already know that computers store the data and instructions in memory. The computer's memory is a sequential collection of storage cells. Each cell is known as a **byte** of memory. Each cell also has a unique address associated with it. This address is a number. Generally the addresses given to memory locations are numbered sequentially.

Whenever a variable is declared in a program, the system allocates memory to hold the value of the variable. Each byte has a unique memory address, therefore each variable also has a unique address associated with it. eg.

```
int i = 10;
```

This declaration reserves space in memory to store the value of the variable i. The name i gets associated with that particular memory address. The value 10 which has been assigned to the variable i gets stored in that particular memory location. We can represent the location of the variable in memory as follows :

Variable_name	i
Value	10
Address	3245

Let us assume that the computer has allocated the address location 3245 to store the value of the integer variable i. Thus, the variable i gets an address associated with it to store its value. It is possible for us to determine the address of a variable in memory. It can be done as follows :

Example : To determine the address of a variable in memory : main()

```
{ int i = 10; printf("\nValue of I :", i); printf("\nAddress of i:" &i);  
}
```

The output is:

Value of I: 10

Problem Solving with Programming Course Material

Address of i :3245

It is clear from the above example that the address of i is obtained with the expression **&i**. We make use of the **address operator (&)** to determine the address of the variable i. Thus the memory address of a variable is a number which is always positive. Since the memory address is a number we can as well store it in memory like any other variable. This is where **pointers** come into picture. In all the programs we have written so far we have made use of the address operator & when using **scanf** to read the values of the variables. The same address operator can be used to determine the address of the corresponding variable in memory.

Thus, pointers are nothing but variables used to hold memory address. A pointer is a variable which contains the address of another variable. It is therefore possible to access the value of a variable either with the help of the variable name or with its address.

Example : Let us write a program to further understand the concept of pointers : main()
{ int a; char ch1; float b;
a = 100; ch1 = 'Z'; b = 40.85; printf("\nThe value of a is %d:", a); printf("\nThe address of a is :", &a); printf("\nThe value of ch1 is %c:", ch1); printf("\nThe address of ch1 is :", &ch1);
printf("\nThe value of b is %f:", b); printf("\nThe address of b is :", &b);
}

The output of the program is :

The value of a is : 100

The address of a is : 6540

The value of ch1 is : Z

The address of ch1 is : 3400

The value of b is : 40.85

The address of b is : 5284

It is important to note here that the addresses of the variables that are output here may not match with the output you get and that every time you run this program, the compiler may assign different memory locations to the variables and you may get different addresses. We can also make use of the **%u** operator to print addresses since addresses are **unsigned integers**.

Since the address of a variable is a number we can as well store it in another variable. Let us use a variable j to store the address of the variable i. The address of i can be stored in j as j = &i; But every variable in C has to be declared before we can use it. Since we are using j for the purpose of storing the address of i we make use of the operator ****** to declare it. This is called the **value at address** operator.

Problem Solving with Programming Course Material

We declare j as follows :

```
int*j;
```

This declaration tells the compiler that j is a variable used to store the address of an integer value. i.e. j points to an integer.

j itself is a variable and so will have its own unique address associated with it. The value at j is the address of the variable i. This can be depicted as follows :

Variable	I	j
Value	10	3245
Address	3245	4000

Declaring a pointer variable :

The general form of declaring a pointer variable is as follows :

*data type *pointer_name;* In this declaration the * means it is a pointer variable *pointer_name* is the name given to the pointer variable and it being a variable needs space in memory.

The data type indicates the type of the data to which the pointer variable points. eg. `int *a;`

This declares that the pointer variable points to an integer data type. `char *ch1;`

Here the pointer variable points to a character data type.

`float *p;` declares p as a pointer to a floating point variable.

When you declare a pointer variable you have to initialise it by assigning to it the address of a variable eg.

`int *p. i; p = &i;` Thus p is initialised and now contains the address of i. Pointers should not be used before they are initialised. When the type of a pointer is declared it will hold the address of that data type only.

eg. `int *i, j;`

`float p; i = &p;`

is invalid since p is declared **float** and the data type in the pointer declaration is declared to be of type **int**.

A pointer variable can also be initialised at the time of declaration also as follows: `float a, *z = &a;`

Here a is declared to be of type **float**. Pointer variable z is also declared to hold address of data type **float** and hence can hold the address of a. (Note that a has to be first declared before assigning its address to z i.e. the statement `float *z = &a, a;` is invalid. Absolute value cannot be assigned to any pointer variable. Thus the following is invalid : `int *p;`

Problem Solving with Programming Course Material

p = 100;

Having seen how to declare and initialise pointer variables let us now see how to make use of the pointer variable to access the value of a variable. **Example** : To determine the value of a variable using a pointer. main()

```
{ int i, *ptr, val; i = 100; ptr = &i; val = *ptr; printf("\nThe value of i is %d", i); printf("\nThe value of i is %d", *ptr); printf("\nThe value of i is %d", *&ptr); printf("\nThe value of i is %d", val); printf("\nThe address of i is %u", &i); printf("\nThe address of i is %u", ptr); }
```

The output of the program will be :

The value of i is 100

The value of i is 100

The value of i is 100

The value of i is 100

The address of i is 65496

The address of i is 65496

The program demonstrates the various ways which can be used to determine the value of a variable. The statement `val = *ptr;` returns the value of the variable whose address is stored in `ptr` to `val`. Thus `*ptr` returns the value of the variable `i`.

`ptr = &i; val = *ptr;` can be combined and written as : `val = *&i;`

Problem Solving with Programming Course Material

Check Your Progress.

1. What will be the output of the following :

a) `int i = 10, *j;
j = &i;
printf("%d\t%u", i, &i);`
.....
.....

b) `float f = 15.3, *ptr = &f;
printf("%u\t%f", &f, f);`
.....
.....

2. Are the following valid ?

a) `int *p;
p = 100;`
.....
.....

b) `float *j;
int i;
j = &i;`
.....
.....

c) `int i, *j = &i;`
.....
.....

d) `char ch1, *cptr;
int i;
cptr = i;`
.....
.....

POINTER ARITHMETIC

Pointer variables can be used in expressions

Remember that `val` will have the same value as the value of the variable `i`. Study thoroughly the concept of pointers before proceeding to the further topics.

eg.

```
float x, p, q, z, *ptr1, *ptr2; ptr1 = &p; ptr2 = &q;
```

Here `ptr1` and `ptr2` are pointer variables which point to variables of type **float**. They are therefore initialised with the addresses of **float** variables `p` and `q` respectively.

then

(i) `x = *ptr1 / *ptr2; *ptr1 = *ptr2 - 10; z = *ptr1 x 10;` are valid.

Example: `main()`

{

Problem Solving with Programming Course Material

```
float a, b, *p1 = &a, *p2 = &b;
float z;
a = 100; b = 21.8; printf("\nThe value of a is %6.2f", a); a = *p1 * 10; printf("The new value of a
is %6.2f", a); z = *p1/*p2; printf("The value of z is %6.2f", z); z = *p1-*p2; printf("The new
value of z is %6.2f", z);
}
```

The output of the program will be:

The value of a is 100.00

The new value of a is 1000.00

The value of z is 45.87

The new value of z is 978.20

Note : *When using '/' (division operator) in pointer arithmetic remember to have a space between the/and * else /* will be mistaken as a comment. Thus write *ptr1/ *ptr2 and not *ptr1/*ptr2. With the above example it is clear that with the use of pointers we have been able to manipulate the values of variables.*

(ii) Pointers can also be incremented or decremented. Thus

ptr1 ++ or ptr2 -- are valid in C.

In this case, it is important to understand what happens when pointers are incremented or decremented. ptr++ will cause the pointer ptr1 to point to the next value of its type. Thus if a is an integer and ptr1 is a pointer to a, then when ptr is incremented its value will be incremented by 2 since an integer is stored in 2 bytes of memory. Thus if the pointer ptr1 has an initial value 4820 then ptr++ will cause its value to be 4822 i.e. its value is incremented by the length of the data type to which it points. This length is called the scale factor.

For the purpose of revising let us once again see the various data types and the number of bytes they occupy in memory

Int	2
	bytes
Char	1 byte
Float	4
	bytes
long int	4
	bytes

Problem Solving with Programming Course Material

Double 8
 bytes

Example : To demonstrate increment and decrement of pointers main()

```
{ int a, *ptr1; float b, *ptr2; ptr1 = &a;
printf("\nptr1 is : %u", ptr1); ptr1++; printf("\nnew ptr1 is %u", ptr1); ptr2 = &b; printf("\nptr2
is : %u", ptr2); ptr2--; printf("\nnew ptr2 is %u", ptr2);
}
```

The output of the program is :

ptr1 is : 65492 new ptr1 is : 65494 ptr2 is : 65494 new ptr2 is : 65490

In this program ptr1 points to an integer variable whereas ptr2 points to a **float** variable. Incrementing ptr1 causes its value to increase by 2 (since int occupies 2 bytes in memory). Decrementing ptr2 causes its value to be decremented by 4 since a **float** occupies 4 bytes in memory.

(iii) Point (ii) can be extended as follows :

C also allows us to add integers to pointers. eg.

ptr2 + 4 ptr1+10

C also allows to subtract integers from pointers :

ptr1 -10 ptr2 - 2

Subtraction of pointers is allowed in C p1-p2

Example : To add and subtract integers from pointers : main()

```
{
int a, b, *ptr1 = &a, *ptr2 = &b; int q = 10, b = 20; printf("\nThe value of ptr1 is %d", ptr1);
*ptr1 = *ptr1 + 10; printf("\nThe new value of ptr1 is %d", *ptr1); printf("\nThe value of ptr2 is
%d", *ptr2);
*ptr2 = *ptr2 - 40; printf("\nThe new value of ptr2 is %d", *ptr2);
*ptr2 = ptr2 - ptr1;
printf("\nThe new value of ptr2 is %d", *ptr2);
}
```

Check the output of this program and study what happens to the values of ptr1 and ptr2 when integers are added to and subtracted from them.

(iv) Pointers can also be compared as :

ptr1 > ptr2 ptr2 < ptr1 ptr1 == ptr2 ptr1 != ptr2 and so on. It is however important to note that pointer variables can

Problem Solving with Programming Course Material

be compared provided both variables point to objects of same data type. (v) **Remember that you cannot -**

- add two pointers $\text{ptr1} + \text{ptr2}$ is invalid
- use pointers in multiplication
 $\text{ptr1} * \text{ptr2}$ is invalid $\text{ptr2} * 10$ is invalid

Check Your Progress.

1. Write True or False :

- Pointers cannot be used in expressions.
- Pointers can be added to integers.
- Multiplication of pointers is valid in C.
- Pointers can be assigned negative values.

2. Answer in 1-2 sentences :

- What cannot be done with pointers in pointer arithmetic?

.....
.....

- What happens when pointers are incremented?

.....
.....

- use pointers in division $\text{ptr1}/20$ is invalid $\text{ptr2}/\text{ptr1}$ is invalid

POINTERS AND ARRAYS

We have already seen that when we declare an array the elements of the array

```
(*fnptr)(a, b);
```

(Note that there is a parenthesis around *fnptr).

This is as good as calling function add():

Problem Solving with Programming Course Material

are stored in contiguous memory locations. In pointers we have studied that whenever we increment a pointer it points to the next memory location of its type. Using this, let us now study pointers and arrays.

When an array is declared, the compiler immediately allocates a base address and sufficient memory to hold all the elements of the array in contiguous memory locations. The base address is the address of the first element (i.e. 0th index) of the array. The compiler also defines the array name as a constant pointer to the first element of the array.

Pointers to one dimensional arrays :

Let us study about pointers and one dimensional arrays with the help of the following example:

Suppose you declare an array as follows :

```
int arr[5] = { 10, 20, 30, 40, 50};
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	← Array Elements
10	20	30	40	50	← Values
2400	2402	2404	2406	2408	← Addresses of Elements

arr is an array of type **int** whose size is five and the elements of the array are initialised in the declaration. Let us assume that the base address of arr is 2400. Then since each element of the array is of type **int** and there are five elements in the array the five elements will be stored as follows

arr is a constant pointer which points to the first element arr[0]. Thus the address of arr[0] in our example is 2400.

We can declare an **int** pointer ptr to point to the array arr as ptr = arr; or ptr = &arr[0];

We already know that when we use the increment operator with the pointer, its value gets increased by the length of the data type to which it points. Here ptr points to data type **int**, therefore incrementing ptr will cause its value to increase by 2 and hence it will point to the next element of the array which is arr[1]. Thus it is possible to obtain the addresses of all the elements of arr[] as follows :

Ptr	=&arr[0]	=2400
ptr + 1	=&arr[1]	=2402
ptr + 2	=&arr[2]	=2404
ptr + 3	=&arr[3]	=2406
ptr + 4	=&arr[4]	=2408

Problem Solving with Programming Course Material

Thus you can obtain the address of the element as :

address of n^{th} element = base address + ($n \times \text{scale factor of data type}$) In our case we can determine the address of the 4th element as : address of the 4th element = 2400 + (4 x scale factor of int)
= 2400 + (4 x 2)
= 2408

We can use pointers to access the elements of an array. Thus we can access arr[3] as *(ptr+3), arr[2] as *(ptr + 2) and so on. The pointer accessing method is very fast as compared to accessing by the index number as arr[2], arr[4] etc. **Example** : A program to obtain the addresses of all elements in the array :

main()

```
{ int arr[5] = {10,20, 30, 40, 50}; int i, *ptr; ptr = &arr[0]; for (i = 0; i <5; i++)  
{  
printf("Element: %d\tAddress : %u", *ptr, ptr); ptr++;  
}  
}
```

A sample run would give the following :

Element: 10	Address :
	65488
Element: 20	Address :
	65490
Element: 30	Address :
	65492
Element: 40	Address :
	65494
Element: 50	Address :
	65496

In the above program note that we have not used indexing to access the elements of the array. Instead we have incremented the pointer ptr everytime so that it points to the next memory location of its type. Accessing array elements with pointers is always faster as compared to accessing them by subscripts. This method can be very effectively used if the elements are to be accessed in a fixed order according to some definite logic. If elements are to be accessed

Problem Solving with Programming Course Material

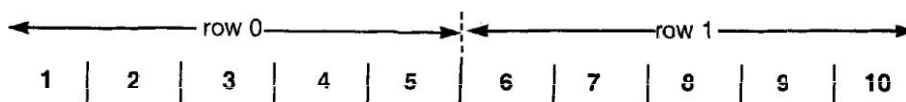
randomly, then using subscripts would be easier though not as fast as accessing them using pointers.

Pointers to 2 dimensional arrays :

We already know that elements of a two dimensional array are stored row wise. Thus the base address of a two dimensional array `arr[]` is the address of the `arr[0][0]`th element which will be obtained as `&arr[0][0]`. The compiler then allocates contiguous memory locations to the array elements row wise i.e first all the elements of row 1 are stored then all elements of row 2 and so on. Let us see this representation with the help of the following example :

```
int arr[2][5] = {(1,2,3,4,5),
(6,7,8,9,10)
};
```

The elements will be stored in memory rowwise as : If we want to access the element `arr[1][2]`



Representation of elements of a 2-dimensional array

we can do it as :

```
arr[1][2] = *(ptr + 5 x 1 + 2)
= *(ptr + 7)
= 8
```

where `ptr` points to the base address of the array. Thus to access an element `arr[i][j]` the formula would be :

```
a[i][j] = *(ptr + no. of cols x i + j)
```

Hence it is essential to define the number of columns i.e. size of each row when declaring a two dimensional array so that the compiler can determine the storage mapping for the array elements.

Pointers and strings :

A string is an array of characters which is terminated by the null character “\0”. Thus the concept of pointers and one dimensional arrays can be extended to array of characters. Let us write a program to determine the values of the elements of the character array with the help of pointers.

Example : To access elements of a string with pointers `main()`

Problem Solving with Programming Course Material

```
{  
char str1[25] = "Pointers"; char *cp; cp = &str1[0]; while(*cp != '\0')  
{  
printf("\nCharacter :%c\tAddress : %u", *cp, cp); cp++;  
}  
}
```

The output of the program will be :

Character: P	Address :
	65472
Character: o	Address :
	65473
Character: i	Address :
	65474
Character: n	Address :
	65475
Character: t	Address :
	65476
Character: e	Address :
	65477
Character: r	Address :
	65478
Character: s	Address :
	65479

Since characters require one byte of storage in memory, incrementing pointer cp will increment its value by 1 and it will point to the next character in the string.

The concept of single dimension array of characters i.e. string can be extended to the table of strings. When we declare a two dimensional array of strings, each string will be allocated equal length as specified in its declaration. However, in practice the all strings of the table are rarely equal in length. Hence instead of making each row of a fixed number of characters we can make each row a pointer to a string of varying lengths.

eg. char *name[3] =

```
{  
"Jimmy";
```

Problem Solving with Programming Course Material

```
“Jill”;  
“Joseph”  
};
```

The above declaration declares name to be an array of three pointers where each pointer points to the particular name. This can be shown as follows :

name[0]-----> Jimmy name[1]-----> Jill name[2]-----> Joseph

Had we declared name to be a two dimensional array of strings as name[3][10] it would have reserved 30 bytes for the array name, where each name would be allocated 10 bytes. But when we declare name to be an array of pointers, where each element points to a name, the total memory allocated would be 18 bytes as follows

```
  J | i | m | m | y | \0  
  J | i | l | l | \0  
:  
  J | o | s | e | p | h | \0
```

In order to access the jth character of the ith row :

$*(name[i] + j)$ would be useful. Note that we first select the row, then the jth element of that particular row and then determine value at address.

We can print the names in the array as shown in the following example:

Example : To demonstrate an array of pointers to strings.

```
main()  
{  
int i;  
char *name[3] = {  
“Jimmy”,  
“Jill”,  
“Joseph”  
};  
printf(“\nNames in the array :”); for(i=0; i<3; i++) printf(“\n%s”, name[i]);  
}
```

Check Your Progress.

1. Write the formulae for accessing the following with pointers:

- a) The nth element of a one dimensional array of float:

Problem Solving with Programming Course Material

-
.....
b) The *i*th character of a string

-
.....
c) The *a*[*i*][*j*]th element of the array two dimensional array *a*[][].

2. Write programs in C for the following using pointers :

- a) Find the average of elements of array *a*[5] of type *int*.
b) Reverse the string “Pointers&Arrays” using pointers to strings.
c) Write a program using pointers to input elements to an integer array and print them in the reverse order.

POINTERS TO POINTERS

We know that a pointer variable contains the address of a data type. Since the addresses are always whole numbers the pointers will always contain whole numbers. The declaration `char *ch` does not imply that *ch* contains a data type **char**. It implies that *ch* is a pointer to a data type **char**, i.e *ch* contains the address of a **char** variable, i.e. the value pointed to by *ch* is of type-**char**.

The concept of pointers can thus be extended further. A pointer contains the address of a variable. This variable itself can be a pointer. We can have a pointer to contain the address of another pointer.

eg.

```
int i, *j,**k; j = &i; k = &j;
```

i is an **int** data type and *j* is a pointer to *i*. *k* is a pointer variable which points to the integer pointer *j*. The value at *k* will be the address of *j*. In principle, there is no limit how far you can extend the concept of pointers to pointers. You can further have another pointer to point to *k* and so on.

Example : Let us write a small program to illustrate pointers to pointers :

```
main()
{
char ch, *ch_ptr; int **ptr; ch = 'A'; ch_ptr = &ch; ptr = &ch_ptr; printf("\nCharacter is : %c :",
ch); printf("\nAddress of ch is : %u", ch_ptr); printf("\nValue of ch_ptr is : %u", ch_ptr);
```


Problem Solving with Programming Course Material

```
printf("\nAddress of ch_ptr is : %u", ptr); printf("\nValue of ptr is : %u", ptr);  
printf("\nCharacter is :%c", *ch_ptr);  
printf("\nCharacter is :%c", **ptr);  
}
```

Carefully follow the program to see how to obtain addresses and values at addresses with the help of the above program. Also note that although `ch` is of type **char** and `ch_ptr` is a pointer to `ch`, the pointer `ptr` to `ch_ptr` is declared of type **int**, since it is going to hold the address of `ch_ptr` which is always going to be a whole number.

Check Your Progress.

1. Write the declaration and initialisation for the following :

- a) A pointer to contain the address of a data type float:
.....
.....
- b) A pointer to contain the address of a data type int and another to contain the address of this pointer:
.....
.....
- c) A pointer to contain the address of a data of type char.

UNIT II: Topics included

Structures

Basics of structure in C, structure members, accessing structure members, nested structures, array of structures, pointers to structures - example programs, Unions- accessing union members- example programs.

Project: Simple C project by using structures/unions.

It is the collection of dissimilar data types or heterogeneous data types grouped together. It means the data types may or may not be of same type. Structure is a collection of variables (can be of different types) under a single name.

For example: You want to store information about a person: his/her name, citizenship number and salary. You can create different variables *name*, *citNo* and *salary* to store these information separately.

Problem Solving with Programming Course Material

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: *name1*, *citNo1*, *salary1*, *name2*, *citNo2*, *salary2* etc.

A better approach would be to have a collection of all related information under a single name Person structure, and use it for every person.

Structure declaration-

```
struct tagname
{
Data type member1;
Data type member2;
Data type member3;
.....
.....
Data type member n;
};
```

OR

```
struct
{
Data type member1;
Data type member2;
Data type member3;
.....
.....
Data type member n;
};
```

OR

```
struct tagname
{
struct element 1;
```

Problem Solving with Programming Course Material

```
struct element 2;  
struct element 3;  
.....  
.....  
struct element n;  
}
```

Structure variable declaration;

```
struct student  
{  
int age;  
    char name[20];  
    char branch[20];  
}; struct student s;
```

Initialization of structure variable-

Like primary variables structure variables can also be initialized when they are declared.

Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program. We cant initialize structure members while defining the structure

```
struct student  
{  
int age=20;  
    char name[20]="sona";  
}s1;
```

The above is **invalid**.

A structure can be initialized as

```
struct student  
{  
int age,roll;  
    char name[20];  
} struct student s1={16,101,"sona"};
```

Problem Solving with Programming Course Material

```
struct student s2={17,102,"rupa"};
```

If initialize is less than no.of structure variable, automatically rest values are taken as zero.

Accessing structure elements-

Dot operator is used to access the structure elements. Its associativity is from left to right.

structure variable ;

```
s1.name[];
```

```
s1.roll;
```

```
s1.age;
```

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
    int roll, age;
```

```
    char branch;
```

```
} s1,s2;
```

```
printf("\n enter roll, age, branch=");
```

```
scanf("%d %d %c", &s1.roll, &s1.age, &s1.branch);
```

```
s2.roll=s1.roll;
```

```
printf(" students details=\n");
```

```
printf("%d %d %c", s1.roll, s1.age, s1.branch);
```

```
printf("%d", s2.roll);
```

```
}
```

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

Size of structure- Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

```
sizeof(struct student); or
```

```
sizeof(s1);
```

```
sizeof(s2);
```

Problem Solving with Programming Course Material

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

```
#include<stdio.h>
#include<string.h>
struct student
{
char name[30];
char branch[25];
int roll;
};
void main()
{
struct student s[200];
int i;
s[i].roll=i+1;
printf("\nEnter information of students:");
for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll);
printf("\nEnter the name:");
scanf("%s",s[i].name);
printf("\nEnter the branch:");
scanf("%s",s[i].branch); printf("\n");
}
printf("\nDisplaying information of students:\n\n");
```

Problem Solving with Programming Course Material

```
for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
printf("\nName:");
puts(s[i].name);
printf("\nBranch:");
puts(s[i].branch);
}
}
```

In Array of structures each element of array is of structure type as in above example.

Array within structures

```
struct student
{
char name[30];
int roll,age,marks[5];
}; struct student s[200];
```

We can also initialize using same syntax as in array.

Problem Solving with Programming Course Material

Nested structure

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

```
struct student
{
    element 1;
    element 2;
    .....
    .....
    struct student1
    {
        member 1;
        member 2;
        }variable 1;
    .....
    .....
    element n;
    }variable 2;
```

It is possible to define structure outside & declare its variable inside other structure.

```
struct date
{
    int date,month;
};
```

```
struct student
{
    char nm[20];
    int roll;
    struct date d;
}; struct student s1;
struct student s2,s3;
```

Problem Solving with Programming Course Material

Nested structure may also be initialized at the time of declaration like in above example.

```
student s={"name",200, {date, month}}; {"ram",201, {12,11}};
```


Problem Solving with Programming Course Material

Nesting of structure within itself is not valid. Nesting of structure can be extended to any level.

```
struct time
{
    int hr,min;
};
struct day
{
    int date,month;
    struct time t1;
};
struct student
{
    char nm[20];
    struct day d;
}stud1, stud2, stud3;
```

Structure Pointers

As we have said already we need call by reference calls which are much more efficient than normal call by value calls when passing structures as parameters. This applies even if we do not intend the function to change the structure argument.

A structure pointer is declared in the same way as any pointer for example

```
struct address {
    char name[20] ;
    char street[20] ;
};
struct address person ;
struct address *addr_ptr ;
```

declares a pointer `addr_ptr` to data type *struct address*. To point to the variable `person` declared above we simply write

```
addr_ptr = &person ;
```

which assigns the address of `person` to `addr_ptr`.

To access the elements using a pointer we need a new operator called the arrow operator, `->`, which can be used **only** with structure pointers. For example

Problem Solving with Programming Course Material

```
puts( addr_ptr -> name );
```

Problem Solving with Programming Course Material

For Example :- Program using a structure to store time values.

```
#include <stdio.h>
struct time_var {
    int hours, minutes, seconds ;
};
void display ( const struct time_var * ) ;
/* note structure pointer and const */
void main()
{
    struct time_var time ;
    time.hours = 12 ;
    time.minutes = 0 ;
    time.seconds = 0 ;
    display( &time ) ;
}
void display( const struct time_var *t )
{
    printf( "%2d:%2d:%2d\n", t -> hours, t -> minutes, t -> seconds ) ;
}
```

Note that even though we are not changing any values in the structure variable we still employ call by reference for speed and efficiency. To clarify this situation the *const* keyword has been employed.

Dynamic allocation of structures

The memory allocation functions may also be used to allocate memory for user defined types such as structures. All malloc() basically needs to know is how much memory to reserve.

For Example :-

```
struct coordinate {
    int x, y, z ;
};
struct coordinate *ptr ;
ptr = (struct coordinate * ) malloc( sizeof ( struct coordinate ) ) ;
```

Problem Solving with Programming Course Material

Problem Solving with Programming Course Material

UNION

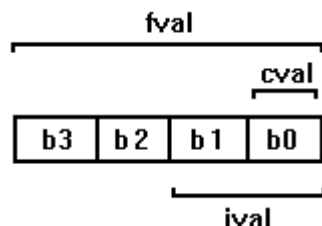
Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

A union is data type where the data area is shared by two or more members generally of different type at different times.

For Example :-



```
union u_tag {  
    short ival ;  
    float fval ;  
    char cval ;  
} uval ;
```

The size of uval will be the size required to store the largest single member, 4 bytes in this case to accommodate the floating point member.

Union members are accessed in the same way as structure members and union pointers are valid.

```
uval.ival = 10 ;  
uval.cval = 'c' ;
```

When the union is accessed as a character we are only using the bottom byte of storage, when it is accessed as a short integer the bottom two bytes etc. It is up to the programmer to ensure that the element accessed contains a meaningful value.

Problem Solving with Programming Course Material

A union might be used in conjunction with the bit-field *struct status* in the previous section to implement binary conversions in C.

For Example :-

```
union conversion {
    unsigned short num ;
    struct status bits ;
} number ;
```

Syntax of union:

```
union student
{
    datatype member1;
    datatype member2;
};
```

Like structure variable, union variable can be declared with definition or separately such as

```
union union name
{
    Datatype member1;
}var1;
```

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

```
struct student
{
    int i;
    char ch[10];
};struct student s;
```

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Nested of Union

When one union is inside the another union it is called nested of union.

Problem Solving with Programming Course Material

Example:- union a

```
{
int i;
int age;
};
union b
{
char name[10];
union a aa;
}; union b bb;
```

There can also be union inside structure or structure in union.

Example:-

```
void main()
{
struct a
{
    int i;
    char ch[20];
};
struct b
{
int i;
char d[10];
};
union z
{
struct a a1;
struct b b1;
}; union z z1;
z1.b1.j=20;
z1.a1.i=10;
z1.a1.ch[10]= "i";
z1.b1.d[0]="j";
```

Problem Solving with Programming Course Material

```
printf(" ");
```

Example Programs

Program to add two distances which is in feet and inches

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

```
struct Distance
```

```
{
```

```
    int feet;
```

```
    float inch;
```

```
} dist1, dist2, sum;
```

```
int main()
```

```
{
```

```
    printf("1st distance\n");
```

```
    printf("Enter feet: ");
```

```
    scanf("%d", &dist1.feet);
```

```
    printf("Enter inch: ");
```

```
    scanf("%f", &dist1.inch);
```

```
    printf("2nd distance\n");
```

```
    printf("Enter feet: ");
```

```
    scanf("%d", &dist2.feet);
```

```
    printf("Enter inch: ");
```

```
    scanf("%f", &dist2.inch);
```

```
    // adding feet
```

```
    sum.feet = dist1.feet + dist2.feet;
```

```
    // adding inches
```

```
    sum.inch = dist1.inch + dist2.inch;
```

```
    // changing feet if inch is greater than 12
```

```
    while (sum.inch >= 12)
```

```
    {
```

```
        ++sum.feet;
```

```
        sum.inch = sum.inch - 12;
```

```
    }
```

```
    printf("Sum of distances = %d\'-%.1f'", sum.feet, sum.inch);
```


Problem Solving with Programming Course Material

```
    return 0;
}
```

Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
```

Sum of distances = 15'-5.7"

Example: Store Information and Display it Using Structure

```
#include <stdio.h>

struct student
{
    char name[50];
    int roll;
    float marks;
} s;

int main()
{
    printf("Enter information:\n");
    printf("Enter name: ");
    scanf("%s", s.name);
    printf("Enter roll number: ");
    scanf("%d", &s.roll);
    printf("Enter marks: ");
    scanf("%f", &s.marks);
    printf("Displaying Information:\n");
    printf("Name: ");
    puts(s.name);
    printf("Roll number: %d\n",s.roll);
    printf("Marks: %.1f\n", s.marks);
    return 0;
}
```

Problem Solving with Programming Course Material

```
}
```

Output: Enter information:

Enter name: Jack

Enter roll number: 23

Enter marks: 34.5

Displaying Information:

Name: Jack

Roll number: 23

Marks: 34.5

Example: Access structure members using pointer

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

```
struct person
```

```
{
```

```
    int age;
```

```
    float weight;
```

```
};
```

```
int main()
```

```
{
```

```
    struct person *personPtr, person1;
```

```
    personPtr = &person1;
```

```
    printf("Enter age:");
```

```
    scanf("%d", &personPtr->age);
```

```
    printf("Enter weight:");
```

```
    scanf("%f", &personPtr->weight);
```

```
    printf("Displaying:\n");
```

```
    printf("Age: %d\n", personPtr->age);
```

```
    printf("weight: %f", personPtr->weight);
```

```
    return 0;
```

```
}
```

Example: Access structure members using pointer

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

```
struct dog
```

```
{
```

Problem Solving with Programming Course Material

```
char name[10];
char breed[10];
int age;
char color[10];
};
int main()
{
    struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
    struct dog *ptr_dog;
    ptr_dog = &my_dog;
    printf("Dog's name: %s\n", ptr_dog->name);
    printf("Dog's breed: %s\n", ptr_dog->breed);
    printf("Dog's age: %d\n", ptr_dog->age);
    printf("Dog's color: %s\n", ptr_dog->color);
    // changing the name of dog from tyke to jack
    strcpy(ptr_dog->name, "jack");
    // increasing age of dog by 1 year
    ptr_dog->age++;
    printf("Dog's new name is: %s\n", ptr_dog->name);
    printf("Dog's age is: %d\n", ptr_dog->age);
    // signal to operating system program ran fine
    return 0;
}
```

Example: Access structure members using pointer

```
#include <stdio.h>
struct my_structure {
    char name[20];
    int number;
    int rank;
};
int main()
```

Problem Solving with Programming Course Material

```
{
    struct my_structure variable = {"StudyTonight", 35, 1};
    struct my_structure *ptr;
    ptr = &variable;
    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);
    return 0;
}
```

Example: Access Union members using Structures

```
#include <stdio.h>
union unionJob
{
    //defining a union
    char name[32];
    float salary;
    int workerNo;
} uJob;
struct structJob
{
    char name[32];
    float salary;
    int workerNo;
} sJob;

int main()
{
    printf("size of union = %d", sizeof(uJob));
    printf("\nsize of structure = %d", sizeof(sJob));
    return 0;
}
```

Problem Solving with Programming Course Material

Example: Access Union members using Structures one union member can be accessed at a time

```
#include <stdio.h>
union job
{
    char name[32];
    float salary;
    int workerNo;
} job1;

int main()
{
    printf("Enter name:\n");
    scanf("%s", &job1.name);
    printf("Enter salary: \n");
    scanf("%f", &job1.salary);
    printf("Displaying\nName :%s\n", job1.name);
    printf("Salary: %.1f", job1.salary);
    return 0;
}
```

Project: Simple C project by using structures/unions.

1. Let us introduce two new functions intended for strings. You may enter “You’re making me blue” and then enter “All my loving”. Show the output and answer the questions.

```
#include <stdio.h>
void main()
{
    char a[25], b[25];
    puts("Name your favorite song title: ");
    // gets(a); gets_s(a, 25);
    puts("\nName another song title:");
    // scanf("%s", &b);
    scanf_s("%s", &b, 25);
}
```

Problem Solving with Programming Course Material

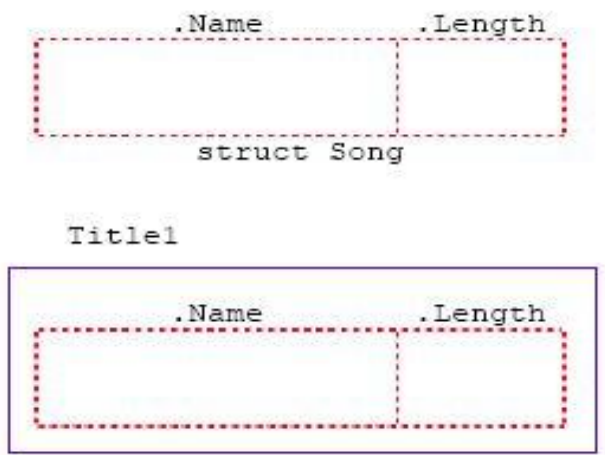
```
printf("\na = %s, b = %s\n", a, b);  
}
```

- a. Was the `scanf_s()` able to read the entire string or only the first word?
- b. Was the `gets_s()` able to read the entire string or only the first word?
- c. Did the `gets_s()` stop reading at the first space or the first return that was entered?
- d. Did the `puts()` add a `'\n'` at the end of the string that is printed?
- e. Which function would you prefer to read in a string, `scanf_s()` or `gets_s ()`?

Problem Solving with Programming Course Material

2. Firstly, enter “Ooowee Babe” and then 2.40 for the following program. Show the output and answer the questions.

```
#include <stdio.h>
void main()
{
definition of structure struct Song
{
char Name[25]; float Length;
};
// end of definition
// variable declaration struct Song Title1;
printf("The size of Title1 structure variable is %d bytes.\n", sizeof(Title1)); puts("Name your
favorite song title: "); gets_s(Title1.Name, 25);
puts("How long is it?");
scanf_s("%f", &Title1.Length);
printf("\nYour song is ");
puts(Title1.Name);
printf("And it is %.2f min. long.\n", Title1.Length);
}
```



Just as the int and float are data types, struct Song is a new data type that we have defined.

- What are the two known types used to define struct Song?
- struct Song is defined with two members. What are their names?
- Do the words char or float by themselves create new variables or allocate space in memory?
- Since struct Song is also a new data type, do you think that it creates in itself a new variable?

Problem Solving with Programming Course Material

- e. Is a semicolon used at the end of the structure definition?
- f. Since the definition of the structure doesn't create a new variable, what is the name of the variable declared using the struct Song data type?
- g. Title1 is a new variable that takes up space in memory. How many parts does it have? What are their names?
- h. Show the contents of each member inside the box Title1.
- i. An array is **a collection of many items of the same data type**, such as int or char. Similarly, a structure data type is a collection of many items. Do they have to be of the same data type?

```
Name your favorite song title:
Wind of Change

Name another song title:
Seek and Destroy

a = Wind of Change, b = Seek
Press any key to continue . . .
```

- a. Only the first word.
- b. The entire string.
- c. The first return that was entered.
- d. Yes it did.
- e. Of course gets()/gets_s().

```
The size of Title1 structure variable is 32 bytes.
Name your favorite song title:
Oo wee Babe
How long is it?
2.40

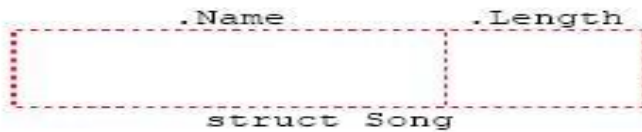
Your song is Oo wee Babe
And it is 2.40 min. long.
Press any key to continue . . .
```

- a. char and float.
- b. Name and Length.
- c. They allocate space in memory.
- d. Yes it creates in itself a new variable.
- e. Yes.

Problem Solving with Programming Course Material

f. Title1.

g. It has two parts named name and length.



Title1



j. When accessing a slot in an array, a set of brackets is used, such as `a[2] = 0;`. When we want to access a member of a structure, what do we use?

k. How would you have assigned the Length member of Title1 to 0?

l. How would you have assigned the name member of Title1 to "Mr. Moonlighting"?

3. In the following program example enter "Riders on the Storm" and 3.10 for the sample input data. Show the output and answer the questions.

- i. No. structure can have different data types.
- j. We use a dot operator and the structure's member name.
- k. `Title1.Length = 0;`
- l. `Title1.Name = "Mr. Moonlighting";`

```
#include <stdio.h>
#include <string.h>
void main()
{
struct Song
{
char Name[25];
float Length;
};
struct Song Title1, Title2;
strcpy_s(Title2.Name, 25, "My Teardrops");
```

Problem Solving with Programming Course Material

```
Title2.Length = 2.35f;
puts("Name your favorite song:");
gets_s(Title1.Name, 25);
puts("How long is it?");
scanf_s("%f", &Title1.Length);
printf("\nMy song is %s\n Your song is ", Title2.Name); puts(Title1.Name);
printf("Yours is %.2f min. longer \n", Title1.Length - Title2.Length);
}
```

- a. Can you print out both members of Title2 without specifying the member names as shown below? Does this work?

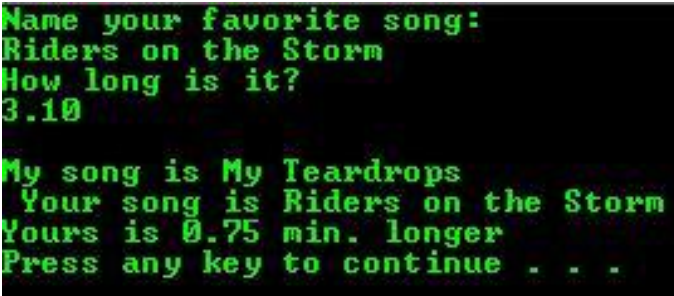
```
printf("%s %.2f\n", Title2);
```

- b. Can you assign Title2 to Title1 without specifying their members as shown below? Does this work?

```
Title1 = Title2;
```

- c. Does the following code work? Why or why not?

```
Title1.Name = Title2.Name
```



```
Name your favorite song:
Riders on the Storm
How long is it?
3.10

My song is My Teardrops
Your song is Riders on the Storm
Yours is 0.75 min. longer
Press any key to continue . . .
```

- a. No, we can't and this doesn't work. We need the name of the member.
- b. Yes we can and it does work provided that they both have same structure as in this example. Here we assign the whole structure.
- c. It doesn't work. We cannot assign the value of Title2.Name directly to Title1.Name but we can copy the value as the following code:

```
strcpy_s(Title1.Name, 25, Title2.Name);
```

Problem Solving with Programming Course Material

UNIT III: Topics included

Functions

Functions: User-defined functions, categories of functions, parameter passing in functions: call by value, call by reference, recursive functions. passing arrays to functions, passing strings to functions, passing a structure to a function.

User-Defined Functions:

C functions can be classified in two categories: Library functions and User defined functions. The difference between this two functions is, Library functions are already built in and we need not to write its code, whereas a User defined function has to be written to get it executed during the output.

Need for User-Defined functions:

There are times when certain operation or calculation are to be repeated during a program. For instance, we may use a factorial of a number or printing some string lines in the program. In this situations we may repeat the code in our program .Here, user-defined functions can be really helpful and can save our time and program space.

Syntax :

```
1 function_type function_name(parameter list)
2 {
3     local variable declaration;
4     executable statement 1;
5     executable statement 2;
6     .....;
7     .....;
8     return statement;
9 }
```

Explanation of the Syntax:

1) Function Header :

The first line in the above given syntax is known as header function. The function header consists of three parts: *function type*, *function name* and the *function parameter list*.

Problem Solving with Programming Course Material

- Function type: This may consist of the datatypes that you use. For example float, int, double.

NOTE: If datatype is not specified then C will assume it as *int* type. And if the function does not return any value then use *void*.

-Function name: This may consist of any variable that is suitable for user's understanding. For example: If you have made user defined function for factorial then use *fact* and if for simple multiplication then *mul*.

-Parameter List: It declares the variables that are to be used in the function and that are going to be called in the program

2) Function Body:

The function body contains the declaration statement necessary for performing the required task. The body enclosed in braces contain three parts:

- A *return* statement that returns the value evaluated by the function.

```
1 float mul (float x, float y)
2 {
3     float result; /*local variable*/
4     result = x*y; /*computes the product*/
5     return (result); /*returns the result*/
6 }
```

- Function statement that perform the task of the function.Using void as shown below:

```
1 void sum (int a, int b)
2 {
3     printf("sum=%s", a+b); /*no local variables*/
4     return; /*Good habit to use it even in void*/
5 }
```

Local declaration that specify the variable needed by the function.

```
1 void display (void)
2 {
3     printf("No type, No parameters"); /*no local variables*/
4     /*no return statement*/
5 }
```

Problem Solving with Programming Course Material

NOTE: If function does not return any value, we can omit the return statement.

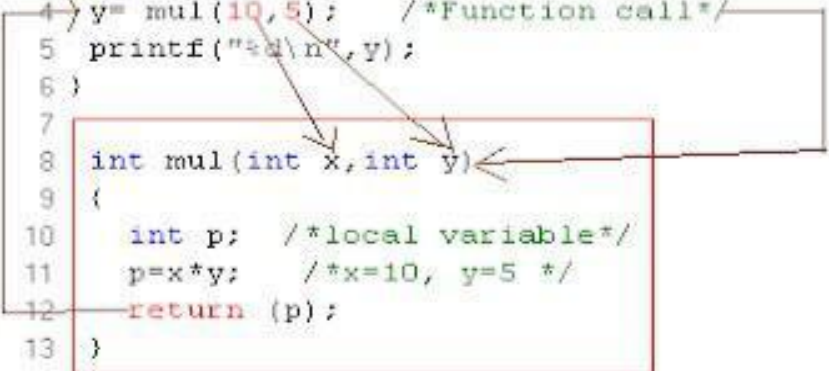
Function call :

A function can be called simply using the function name followed by a list of actual parameters(or arguments), if any enclosed in parentheses. Let's take an example for multiplication of two numbers.

```
1 main()
2 {
3  int y;
4  y= mul(10,5); /*Function call*/
5  printf("%d\n",y);
6 }
```

When the compiler encounters a function call, the control is transferred to the function *mul()*. This function is then executed line by line as described and a value is returned when a *return* statement is encountered. This value is assigned to *y*. This is illustrated below:

```
1 main()
2 {
3  int y;
4  y= mul(10,5); /*Function call*/
5  printf("%d\n",y);
6 }
7
8 int mul(int x,int y)
9 {
10  int p; /*local variable*/
11  p=x*y; /*x=10, y=5 */
12  return (p);
13 }
```



Example of user-defined function

Write a C program to add two integers. Make a function add to add integers and display sum in main() function.

```
/*Program to demonstrate the working of user defined function*/
```

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

```
int add(int a, int b); //function prototype(declaration)
```

Problem Solving with Programming Course Material

```
int main(){
    int num1,num2,sum;
    printf("Enters two number to add\n");
    scanf("%d %d",&num1,&num2);
    sum=add(num1,num2);      //function call
    printf("sum=%d",sum);
}
int add(int a,int b)      //function declarator
{
/* Start of function definition. */
    int add;
    add=a+b;
    return add; //return statement of function /* End of function
definition. */ }
```

Function prototype(declaration):

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax of function prototype

```
return_type function_name(type(1) argument(1),...,type(n) argument(n));
```

In the above example,int add(int a, int b); is a function prototype which provides following information to the compiler:

1. name of the function is add()
2. return type of the function is int.
3. two arguments of type int are passed to function.

Function prototype are not needed if user-definition function is written before main() function.

Function call

Control of the program cannot be transferred to user-defined function unless it is called invoked).

Syntax of function call

```
function_name(argument(1),...,argument(n));
```

Problem Solving with Programming Course Material

In the above example, function call is made using statement `add(num1,num2);` from `main()`. This make the control of program jump from that statement to function definition and executes the codes inside that function.

Function definition

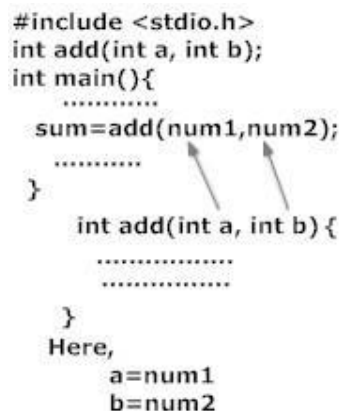
Function definition contains programming codes to perform specific task.

Passing arguments to functions:

In programming, argument/parameter is a piece of data(constant or variable) passed from a program to the function.

In above example two variable, `num1` and `num2` are passed to function during function call and these arguments are accepted by arguments `a` and `b` in function definition.

```
#include <stdio.h>
int add(int a, int b);
int main(){
.....
sum=add(num1,num2);
.....
}
int add(int a, int b){
.....
}
Here,
a=num1
b=num2
```



Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument `num1` was of `int` type and `num2` was of `float` type then, argument variable `a` should be of type `int` and `b` should be of type `float`,i.e., type of argument during function call and function definition should be same.

A function can be called with or without an argument.

Return Statement

Return statement is used for returning a value from function definition to calling function.

Syntax of return statement

```
return (expression);
```

Problem Solving with Programming Course Material

OR

```
return;
```

For example:

```
return;
```

```
return a;
```

```
return (a+b);
```

In above example, value of variable *add* in *add()* function is returned and that value is stored in variable *sum* in *main()* function. The data type of expression in return statement should also match the return type of function.

Types of User-defined Functions(Categories of Functions) in C:

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

1. Function with no arguments and no return value
2. Function with no arguments and return value
3. Function with arguments but no return value
4. Function with arguments and return value.

Let's take an example to find whether a number is prime or not using above 4 categories of user defined functions.

Function with no arguments and no return value.

```
/*C program to check whether a number entered by user is prime or not using function with no arguments and no return value*/ #include <stdio.h>
```

```
void prime();
```

```
int main(){
```

```
    prime();        //No argument is passed to prime().
```

```
    return 0;
```

```
}
```

```
void prime(){
```

```
/* There is no return value to calling function main(). Hence, return type of prime() is
```

```
void */
```

```
    int num,i,flag=0;
```

```
    printf("Enter positive integer enter to check:\n");
```


Problem Solving with Programming Course Material

```
scanf("%d",&num);
for(i=2;i<=num/2;++i){
    if(num%i==0){
        flag=1;
    }
}
if (flag==1)
    printf("%d is not prime",num);
else
    printf("%d is 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

/*C program to check whether a number entered by user is prime or not using function with no arguments but having return value */ #include <stdio.h>

```
int input();
int main(){
    int num,i,flag;
    num=input();    /* No argument is passed to input() */
    for(i=2,flag=i;i<=num/2;++i,flag=i){
        if(num%i==0){
            *printf("%d is not prime",num);
            ++flag;
            break;
        }
    }
    if(flag==i)
        printf("%d is prime",num);
    return 0;
}
```

Problem Solving with Programming Course Material

```
int input(){      /* Integer value is returned from input() to calling
function */
    int n;
    printf("Enter positive enter to check:\n");
    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 and no return value

/*Program to check whether a number entered by user is prime or not using function with arguments and no return value */ #include <stdio.h>

```
void check_display(int n);
int main(){
    int num;
    printf("Enter positive enter to check:\n");
    scanf("%d",&num);
    check_display(num); /* Argument num is passed to function. */ return
    0;
}
void check_display(int n){
/* There is no return value to calling function. Hence, return type of function is
void. */
    int i,flag;
    for(i=2,flag=i;i<=n/2;++i,flag=i){
        if(n%i==0){
            printf("%d is not prime",n);
            ++flag;
            break;
        }
    }
    if(flag==i)
        printf("%d is prime",n);
}
```

Problem Solving with Programming Course Material

```
}
```

Here, check_display() function is used for check whether it is prime or not and display it accordingly. Here, argument is passed to user-defined function but, value is not returned from it to calling function.

Function with argument and a return value

```
/* Program to check whether a number entered by user is prime or not using function
with argument and return value */ #include <stdio.h>
int check(int n);
int main(){
    int num,num_check=0;
    printf("Enter positive enter to check:\n");
    scanf("%d",&num);
    num_check=check(num); /* Argument num is passed to check() function. */
    if(num_check==1)
        printf("%d in not prime",num);
    else
        printf("%d is prime",num);
    return 0;
}*/
int check(int n){
/* Integer value is returned from function check() */ int i;
    for(i=2;i<=n/2;++i){
        if(n%i==0)
            return 1;
    }
    return 0;
}
```

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

Problem Solving with Programming Course Material

Parameter passing in functions: call by value, call by reference

In C Programming we have different ways of parameter passing schemes such as Call by Value and Call by Reference.

Function is good programming style in which we can write reusable code that can be called whenever require.

Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called argument.

Two Ways of Passing Argument to Function in C Language :

Call by Reference

Call by Value

Let us discuss different ways one by one –

A.Call by Value :

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

```
void interchange(int number1,int number2)
```

```
{
```

```
    int temp;
```

```
    temp = number1;
```

```
    number1 = number2;
```

```
    number2 = temp;
```

```
}
```

```
int main() {
```

```
    int num1=50,num2=70;
```

```
    interchange(num1,num2);
```

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

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

```
    return(0);
```

```
}
```

Problem Solving with Programming Course Material

Output :

Number 1 : 50

Number 2 : 70

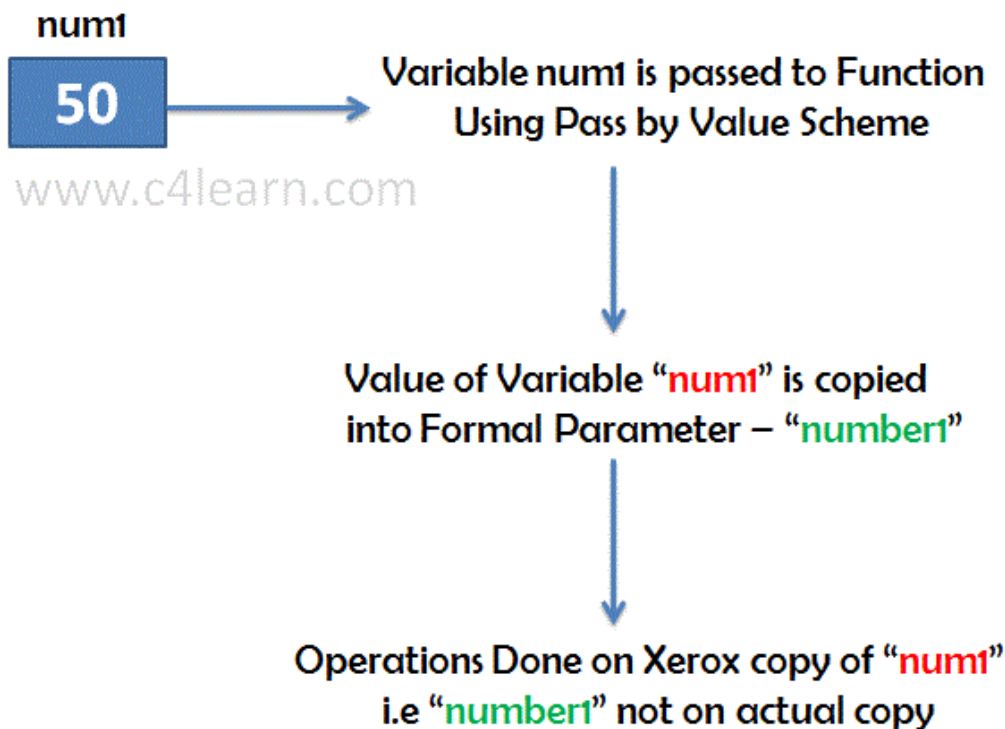
Explanation : Call by Value

While Passing Parameters using call by value , xerox copy of original parameter is created and passed to the called function.

Any update made inside method will not affect the original value of variable in calling function.

In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively.

As their scope is limited to only function so they cannot alter the values inside main function.



B.Call by Reference/Pointer/Address :

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

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

```
{
```

```
    int temp;
```

```
    temp = *num1;
```

```
    *num1 = *num2;
```

Problem Solving with Programming Course Material

```
*num2 = temp;
}
int main() {
    int num1=50,num2=70;
    interchange(&num1,&num2);

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

    return(0);
}
```

Output :

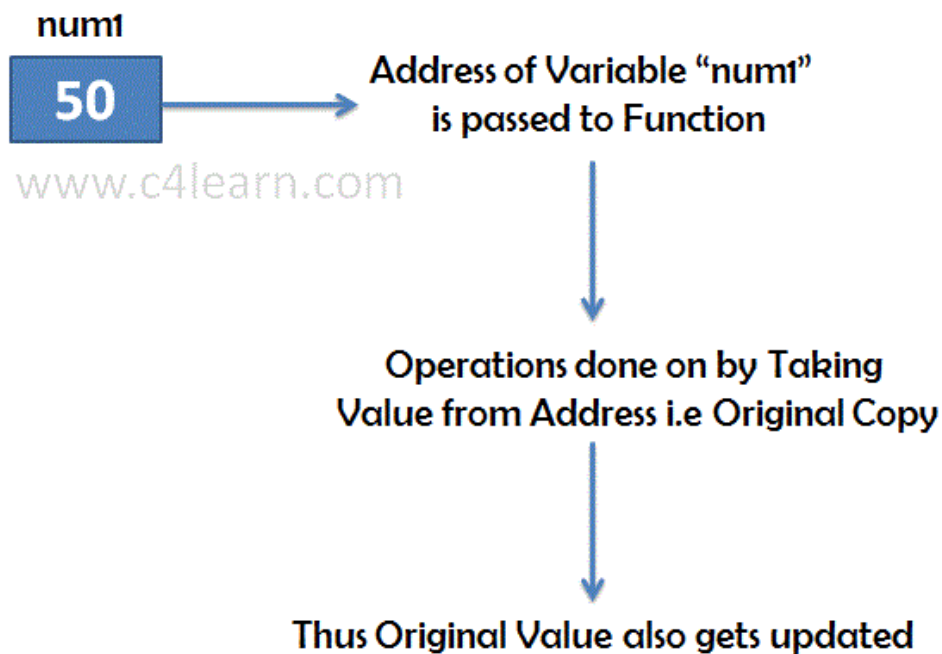
Number 1 : 70

Number 2 : 50

Explanation : Call by Address

While passing parameter using call by address scheme , we are passing the actual address of the variable to the called function.

Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.



Problem Solving with Programming Course Material

Summary of Call By Value and Call By Reference :

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

Recursive functions:

A function that calls itself is known as recursive function and the process in which a function calls itself is known as recursion in C programming.

Example of recursion in C programming

Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include <stdio.h>
int sum(int n);
int main(){
int num,add;
printf("Enter a positive integer:\n");
scanf("%d",&num);
add=sum(num);
printf("sum=%d",add);

}
```

Problem Solving with Programming Course Material

```
int sum(int n){
if(n==0)

return n;
else
return n+sum(n-1);          /*self call    to function sum() */
}
```

Output

Enter a positive integer:

5

15

In, this simple C program, sum() function is invoked from the same function. If n is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, n is 5 initially. Then, during next function calls, 4 is passed to function and the value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.

For better visualization of recursion in this example:

```
sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)

=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0

=5+4+3+2+1
=5+4+3+3
=5+4+6
```


Problem Solving with Programming Course Material

=5+10

=15

Every recursive function must be provided with a way to end the recursion. In this example when, n is equal to 0, there is no recursive call and recursion ends.

Advantages and Disadvantages of Recursion

Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.

In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

Comparison of Recursion and iteration

The differences between recursion and iteration cab be stated as below:

Recursion Vs. Iteration	
Recursion	Iteration

Recursion is the term given to the mechanism of defining a set or procedure in terms of itself. A conditional statement is required in the body of the function for stopping the function

--

Problem Solving with Programming Course Material

Iteration is the block of statement executed repeatedly using loops.

The iteration control statement itself contains statement for stopping the iteration. At every

Problem Solving with Programming Course Material

Recursion Vs. Iteration

Recursion	Iteration
-----------	-----------

execution.

execution, the condition is checked.

At some places, use of recursion generates extra overhead. Hence, better to skip when easy solution is available with iteration.

All problems can be solved with iteration.

Recursion is expensive in terms of speed and memory.

Iteration does not create any overhead. All the programming languages support iteration.

Problem Solving with Programming Course Material

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go in infinite loop.

Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

Number Factorial

Following is an example, which calculates factorial for a given number using a recursive function:

```
#include <stdio.h>

int factorial(unsigned int i)
{
    if(i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}

int main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i)); return 0;
}
```

When the above code is compiled and executed, it produces the following result:
Factorial of 15 is 2004310016

Iterative version to find factorial of a number.

```
/* Iterative Version */
unsigned int iter_factorial(int n)
{
    int f = 1;
    int i;
    for(i = 1; i <= n; i++)
    {
        f *= i;
    }
    return f;
}
```

Fibonacci Series

Following is another example, which generates Fibonacci series for a given number using a recursive function:

Problem Solving with Programming Course Material

```
#include <stdio.h>

int fibonaci(int i)
{
    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
    }
    return fibonaci(i-1) + fibonaci(i-2);
}

int main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("%d\t", fibonaci(i));
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
0    1    1    2    3    5    8    13    21    34
```

Iterative version to find Fibonacci series

```
int Fibonacci(int n)
{
```

Problem Solving with Programming Course Material

```
int f1 = 0;
```

```
int f2 = 1;
```

```
int fn;
```

```
for ( int i = 2; i < n; i++ )
```

Problem Solving with Programming Course Material

```
{  
    fn = f1 + f2;  
    f1 = f2;  
    f2 = fn;  
}  
}
```

Array can be passed to function by two ways :

1. **Pass Entire array**
2. **Pass Array element by element**

1 . Pass Entire array

- Here entire array can be passed as a argument to function .
- Function gets **complete access** to the original array .
- While passing entire array Address of first element is passed to function , any changes made inside function , directly **affects the Original value** .
- Function Passing method : **“Pass by Address“**

2 . Pass Array element by element

- Here individual elements are passed to function as argument.
- Duplicate **carbon copy of Original variable** is passed to function .
- So any changes made inside function **does not affects the original value.**
- Function doesn't get complete access to the original array element.
- Function passing method is **“Pass by Value“**

Passing entire array to function :

Parameter Passing Scheme : Pass by Reference

- Pass name of array as function parameter .
- Name contains the base address i.e (Address of 0th element)
- Array values are updated in function .
- Values are reflected inside main function also.

```
#include<stdio.h>  
#include<conio.h>  
void fun(int arr[])  
{
```

Problem Solving with Programming Course Material

```
int i;
for(i=0;i< 5;i++)
    arr[i] = arr[i] + 10;
}
void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
    scanf("%d",&arr[i]);

printf("\nPassing entire array .....");
fun(arr); // Pass only name of array

for(i=0;i< 5;i++)
    printf("\nAfter Function call a[%d] : %d",i,arr[i]);

getch();
}
```

Output :

Enter the array elements : 1 2 3 4 5

Passing entire array

After Function call a[0] : 11

After Function call a[1] : 12

After Function call a[2] : 13

After Function call a[3] : 14

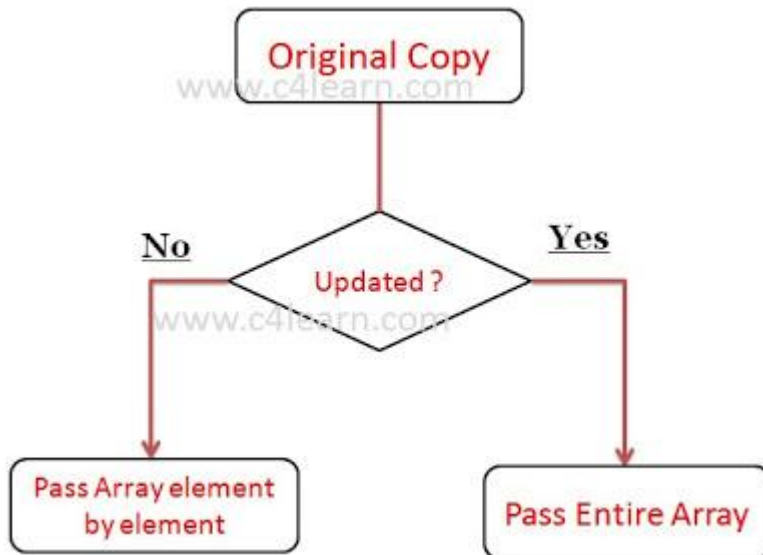
After Function call a[4] : 15

Problem Solving with Programming Course Material

Graphical

Flowchart

:



Passing Entire 1-D Array to Function in C Programming

Array is passed to function Completely.

Parameter Passing Method : Pass by Reference

- It is Also Called “Pass by Address“
- Original Copy is Passed to Function
- Function Body Can Modify Original Value.

Example :

```
#include<stdio.h>
#include<conio.h>
void modify(int b[3]);
void main()
{
int arr[3] = {1,2,3};
modify(arr);
for(i=0;i<3;i++)
printf("%d",arr[i]);
getch();
}

void modify(int a[3])
```

Problem Solving with Programming Course Material

```
{  
int i;  
for(i=0;i<3;i++)  
    a[i] = a[i]*a[i];  
}
```

Output :

1 4 9

Passing array element by element to function :

- Individual element is passed to function using Pass By Value parameter passing scheme
- Original Array elements remains same as Actual Element is never Passed to Function. thus function body cannot modify Original Value.
- Suppose we have declared an array 'arr[5]' then its individual elements are arr[0],arr[1]...arr[4]. Thus we need 5 function calls to pass complete array to a function.

Tabular Explanation :

Consider following array

```
int arr[5] = {11,22,33,44,55};
```

Iteration	Element Passed to Function	Value of Element
1	arr[0]	11
2	arr[1]	22
3	arr[2]	33
4	arr[3]	44
5	arr[4]	55

C Program to Pass Array to Function Element by Element :

```
#include< stdio.h>  
#include< conio.h>  
void fun(int num)  
{
```

Problem Solving with Programming Course Material

```
printf("\nElement : %d",num);
}
void main()
{
int arr[5],i;
clrscr();
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
scanf("%d",&arr[i]);
printf("\nPassing array element by element.....");

for(i=0;i< 5;i++)
    fun(arr[i]);
getch();
}
```

Output :

Enter the array elements : 1 2 3 4 5

Passing array element by element.....

Element : 1

Element : 2

Element : 3

Element : 4

Element : 5

Disadvantage of this Scheme :

This type of scheme in which we are calling the function again and again but with different array element is too much time consuming. In this scheme we need to call function by pushing the current status into the system stack.

It is better to pass complete array to the function so that we can save some system time required for pushing and popping.

Passing strings to functions:

Function declaration to accept one dimensional string

We know that strings are saved in arrays so, to pass an one dimensional array to a function we will have the following declaration.

Problem Solving with Programming Course Material

```
returnType functionName(char str[]);
```

Example:

```
void displayString(char str[]);
```

In the above example we have a function by the name displayString and it takes an argument of type char and the argument is an one dimensional array as we are using the [] square brackets.

Passing one dimensional string to a function

To pass a one dimensional string to a function as an argument we just write the name of the string array variable.

In the following example we have a string array variable message and it is passed to the displayString function.

```
#include <stdio.h>
void displayString(char []);
int main(void) {
    // variables
    char
        message[] = "Hello World";

    // print the string message
    displayString(message);

    return 0;
}
void displayString(char str[]) {
    printf("String: %s\n", str);
}
```

Output:

```
String: Hello World
```

Another way we can print the string is by using a loop like for or while and print characters till we hit the NULL character.

In the following example we have redefined the displayString function logic.

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

Problem Solving with Programming Course Material

```
void displayString(char []);
int main(void) {
    // variables
    char
        message[] = "Hello World";

    // print the string message
    displayString(message);

    return 0;
}
void displayString(char str[]) {
    int i = 0;
    printf("String: ");
    while (str[i] != '\0') {
        printf("%c", str[i]);
        i++;
    }
    printf("\n");
}
```

Output:

String: Hello World

Function declaration to accept two dimensional string

In order to accept two dimensional string array the function declaration will look like the following.

returnType functionName(char [][][C], type rows);

Example:

```
void displayCities(char str[][50], int rows);
```

In the above example we have a function by the name displayCities and it takes a two dimensional string array of type char.

The str is a two dimensional array as because we are using two [][] square brackets.

It is important to specify the second dimension of the array and in this example the second dimension i.e., total number of columns is 50.

Problem Solving with Programming Course Material

The second parameter rows tell us about the total number of rows in the give two dimensional string array str.

The return type for this function is set to void that means it will return no value.

Passing two dimensional string to a function

To pass a two dimensional string to a function we just write the name of the string array variable as the function argument.

In the following example we have the name of 5 cities saved in an array cities of type char. We will be using the displayCitiesfunction to print the names.

```
#include <stdio.h>
void displayCities(char [][][50], int rows);
int main(void) {
    // variables
    char
    cities[][50] = {
        "Bangalore",
        "Chennai",
        "Kolkata",
        "Mumbai",
        "New Delhi"
    };
    int rows = 5;

    // print the name of the cities
    displayCities(cities, rows);

    return 0;
}
void displayCities(char str[][50], int rows) {
    // variables
    int r, i;

    printf("Cities:\n");
```

Problem Solving with Programming Course Material

```
for (r = 0; r < rows; r++) {  
    i = 0;  
    while(str[r][i] != '\0') {  
        printf("%c", str[r][i]);  
        i++;  
    }  
    printf("\n");  
}
```

Output:

Cities:

Bangalore

Chennai

Kolkata

Mumbai

New Delhi

Passing a structure to a function:

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address(reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

It can be done in below 3 ways.

- Passing structure to a function by value
- Passing structure to a function by address(reference)
- No need to pass a structure – Declare structure variable as global

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY VALUE:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

Problem Solving with Programming Course Material

```
1 #include <stdio.h>
2 #include <string.h>
3 struct student
4 {
5     int id;
6     char name[20];
7     float percentage;
8 };
9 void func(struct student record);
10 int main()
11 {
12     struct student record;
13     record.id=1;
14     strcpy(record.name, "Raju");
15     record.percentage = 86.5;
16     func(record);
17     return 0;
18 }
19 void func(struct student record)
20 {
21     printf(" Id is: %d \n", record.id);
22     printf(" Name is: %s \n", record.name);
23     printf(" Percentage is: %f \n", record.percentage);
24 }
```

OUTPUT:

Id is: 1

Name is: Raju

Percentage is: 86.500000

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

Problem Solving with Programming Course Material

```
1 #include <stdio.h>
2 #include <string.h>
3 struct student
4 {
5     int id;
6     char name[20];
7     float percentage;
8 };
9 void func(struct student *record);
10 int main()
11 {
12     struct student record;
13
14     record.id=1;
15     strcpy(record.name, "Raju");
16     record.percentage = 86.5;
17     func(&record);
18     return 0;
19 }
20 void func(struct student *record)
21 {
22     printf(" Id is: %d \n", record->id);
23     printf(" Name is: %s \n", record->name);
24     printf(" Percentage is: %f \n", record-
25 >percentage);
26 }
```

OUTPUT:

Id is: 1

Name is: Raju

Percentage is: 86.500000

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:

Problem Solving with Programming Course Material

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don't need to pass the structure to any function separately.

```
1 #include <stdio.h>
2 #include <string.h>
3 struct student
4 {
5     int id;
6     char name[20];
7     float percentage;
8 };
9 struct student record; // Global declaration of structure
10 void structure_demo();
11 int main()
12 {
13     record.id=1;
14     strcpy(record.name, "Raju");
15     record.percentage = 86.5;
16     structure_demo();
17     return 0;
18 }
19 void structure_demo()
20 {
21     printf(" Id is: %d \n", record.id);
22     printf(" Name is: %s \n", record.name);
23     printf(" Percentage is: %f \n", record.percentage);
24 }
```

OUTPUT:

Id is: 1

Name is: Raju

Percentage is: 86.500000

Problem Solving with Programming Course Material

C – Storage Class Specifiers

Storage class specifiers in C language tells the compiler where to store a variable, how to store the variable, what is the initial value of the variable and life time of the variable.

Syntax:

```
storage_specifier data_type variable _name;
```

TYPES OF STORAGE CLASS SPECIFIERS IN C:

There are 4 storage class specifiers available in C language. They are,

1. auto
2. extern
3. static
4. register

Storage Specifier	Description
auto	Storage place: CPU Memory Initial/default value: Garbage value Scope: local Life: Within the function only.
extern	Storage place: CPU memory Initial/default value: Zero Scope: Global Life: Till the end of the main program. Variable definition might be anywhere in the C program.
static	Storage place: CPU memory Initial/default value: Zero Scope: local Life: Retains the value of the variable between different

Problem Solving with Programming Course Material

	function calls.
register	Storage place: Register memory Initial/default value: Garbage value Scope: local Life: Within the function only.

NOTE:

- For faster access of a variable, it is better to go for register specifiers rather than auto specifiers.
- Because, register variables are stored in register memory whereas auto variables are stored in main CPU memory.
- Only few variables can be stored in register memory. So, we can use variables as register that are used very often in a C program.

1. EXAMPLE PROGRAM FOR AUTO VARIABLE IN C:

The scope of this auto variable is within the function only. It is equivalent to local variable. All local variables are auto variables by default.

```
1 #include<stdio.h>
2 void increment(void);
3 int main()
4 {
5     increment();
6     increment();
7     increment();
8     increment();
9     return 0;
10 }
11 void increment(void)
12 {
13     auto int i = 0 ;
14     printf ( "%d ", i );
15     i++;
16 }
```

Problem Solving with Programming Course Material

OUTPUT:

0 0 0 0

2. EXAMPLE PROGRAM FOR STATIC VARIABLE IN C:

Static variables retain the value of the variable between different function calls.

```
1 //C static example
2 #include<stdio.h>
3 void increment(void);
4 int main()
5 {
6 increment();
7 increment();
8 increment();
9 increment();
10 return 0;
11 }
12 void increment(void)
13 {
14 static int i = 0 ;
15 printf ( "%d ", i );
16 i++;
17 }
```

OUTPUT:

0 1 2 3

3. EXAMPLE PROGRAM FOR EXTERN VARIABLE IN C:

The scope of this extern variable is throughout the main program. It is equivalent to global variable.

Definition for extern variable might be anywhere in the C program.

```
1 #include<stdio.h>
2 int x = 10 ;
3 int main( )
4 {
5 extern int y;
```

Problem Solving with Programming Course Material

```
6 printf("The value of x is %d \n",x);
7 printf("The value of y is %d",y);
8 return 0;
9 }
10 int y=50;
11
```

OUTPUT:

The value of x is 10
The value of y is 50

4. EXAMPLE PROGRAM FOR REGISTER VARIABLE IN C:

- Register variables are also local variables, but stored in register memory. Whereas, auto variables are stored in main CPU memory.
- Register variables will be accessed very faster than the normal variables since they are stored in register memory rather than main memory.
- But, only limited variables can be used as register since register size is very low. (16 bits, 32 bits or 64 bits)

```
1 #include <stdio.h>
2 int main()
3 {
4     register int i;
5     int arr[5]; // declaring array
6     arr[0] = 10; // Initializing array
7     arr[1] = 20;
8     arr[2] = 30;
9     arr[3] = 40;
10    arr[4] = 50;
11    for (i=0;i<5;i++)
12    {
13        // Accessing each variable
14        printf("value of arr[%d] is %d \n", i, arr[i]);
15    }
16    return 0;
```

Problem Solving with Programming Course Material

17 }

OUTPUT:

value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50

Simple C Project: At the end of the Unit III every student should take up this project.

Write a program that takes input of employee information like

- 1) Id
- 2) Name
- 3) Birth date
- 4) Salary

And provides below functionality

- 1) Search on name
- 2) Sort on ID
- 3) Sort on Name
- 4) Sort on Birth Date
- 5) Sort on Salary
- 6) Show all Records

using nested structures and functions.

UNIT IV: TOPICS INCLUDED

File Management

Data Files, opening and closing a data file, creating a data file, processing a data file, unformatted data files.

Project: Simple C project by using files.

CONCEPT OF A FILE

Generally we write programs to display the data on the screen the data which is displayed is limited (i.e, not more than 24 lines). If we want to display more than 24 lines the last 24 lines are only viewed. When we execute a program the output is stored in volatile memory (**RAM**) and its contents would be lost once the program is terminated. So if we need the same data again we have to re-execute the program with same input.

Problem Solving with Programming Course Material

Obviously both these operations would be tedious. At such time it becomes necessary to store the data in a manner that can be later retrieved and displayed either in part or in whole. This medium is usually a 'file' on secondary storage.

The type **FILE** is a **structure** defined in the **stdio.h** file. C requires a **file pointer** to a **FILE** type to access a file or to perform the various file operations, the pointer name can be any valid identifier name. **A file is an external collection of related data treated as a unit.**

A **file** may be stored on anything from a disk (hard disk, CD, DVD) or a tape drive. To access a file first of all we need to open the file by performing an open operation. Once a file is opened, information may be exchanged between file and our program using a temporary area called **buffer**. Not all files have the same capabilities. For example, a disk file can support random access while tape drives cannot.

A Stream is flow of bytes of data. All streams are the same but all files are not. If the file can support **position requests**, opening that file also initializes the **file position indicator** to the start of the file. As each character is read from or written to the file, the position indicator is incremented till it reaches **EOF** (End of File). After which we can close the file by performing close operation. Even if we wont perform the close operation all files are closed automatically when our program terminates normally, either by **main()** or by a call to **exit()**. Files are not closed when a program terminates abnormally, such as when it crashes or when it calls **abort()**. Each stream that is associated with a file has a file control structure of type **FILE**. There are different operations that can be carried out on a file. These are:

- Creating a new file.
- Opening an existing file.
- Moving to a specific location in a file.
- Reading from a file.
- Writing to a file.
- Closing a file.

In order to read or write files, our program needs to use **file pointers**. To obtain a file pointer variable, we use a statement like this: **FILE *fp;**

Problem Solving with Programming Course Material

File name: Whenever we create a new file using our program we need to follow some naming conventions provided by concerned operating system to provide a file name. For example according to windows operating system a file name can be any number of characters,

File information table: All data stored on the disk is in **binary format (0 or 1)**. How this binary data is stored on the disk varies from one Operating system to another. However, this doesn't affect since we use the library functions written for the particular Operating system to be able to perform input/output. It is the compiler vendor's responsibility to correctly implement these library functions.

To implement library functions a program requires several File information such as operating system, Name of the file, Position of the current character in file,.. All such information is handled by **stdio.h** file. Language compiler maintains a table to maintain all such information called as File Information Table. For example: DOS records the location of every directory and file on a disk in a table called the **FAT** (File Allocation Table).

STREAMS

A stream is an entity created by a program. A stream is an abstraction that represents a device on which input and output operations are performed. A stream can basically be represented as a source or destination of characters of indefinite length. Streams are generally associated to a physical source or destination of characters, like a disk file, the keyboard, or the console, so the characters are written to/from a stream which is physically input or output to the physical device.

Text and binary streams

A **text stream** is a sequence of characters. Standard C allows a text stream to be organized into lines terminated by a newline character ('\n'). The newline character is optional. In a text stream, certain character translations may occur as required by the computer. For example, a newline may be converted to a carriage return or linefeed pair. Therefore, there may **not** be a **one-to-one relationship** between the characters that are written (or read) and those on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

Problem Solving with Programming Course Material

A **binary stream** is a sequence of bytes consists of data such as integers, real values or complex numbers. These streams have a **one-to-one correspondence** to those in the external device that is, no character translations occur. The number of bytes written (or read) is the same as the number on the external device. However, an implementation-defined number of null bytes may be appended to a binary stream. These null bytes might be used to fill a sector on a disk.

There are four steps to process a file:

- **Create a stream:** we create a stream by declaring a file pointer of type **FILE** structure. For Example: **FILE* fp;** here **fp** is a pointer to stream(stream pointer) which holds the starting address of stream.
- **Open a file:** once we create a file pointer we can open a file using the standard open function. When the file is opened both file and stream are linked to each other. The file open function returns the address of file type, which is stored in stream pointer variable **fp**.
- **Read or write data:** after creating the stream name we can use the stream pointer to use any stream function (read or write) data using its corresponding stream. If a program reads data from a file then the stream used is **Input Text Stream**. If a program stores data to a file then the stream used is **Output Text Stream**. We can read the data till we reach the end of the file (**EOF**).
- **Close the file:** once the file processing is completed we can close the file using close function. Closing breaks the link between the stream name and the file name. After closing the file the contents on the stream are destroyed automatically since stream is created on **buffer** (temporary memory), which is also called as flushing.

System created streams: C has three system created streams (pointers) available in **stdio.h** file:

Stream	Details
stdin	Returns the numeric value corresponding to the standard input stream. This is filtered through the command line editing functions.
stdout	Returns the numeric value corresponding to the standard output stream. Data written to the standard output is normally filtered through the pager.
stderr	Returns the numeric value corresponding to the standard error stream. It is useful for error messages and prompts.

We should not create any stream names using **stdin** or **stdout** or **stderr** since there streams are already declared in **stdio.h** file. The association in between all these three standard streams,

Problem Solving with Programming Course Material

keyboard and monitor is done automatically. Therefore we can open a file but we cannot open a standard stream in our program code. Like files we need to close streams also at the end of the program, or else when program terminates they are closed automatically.

STANDARD INPUT & OUTPUT FUNCTIONS

All the input & output functions are declared in **stdio.h** header file, which are mainly divided into eight different categories & they are:

- File Open/close.
- Formatted input/output.
- Character input/output.
- Line input/output.
- Block input/output.
- File Positioning.
- System File Operations.
- File Status.

Problem Solving with Programming Course Material

File open: The **fopen()** function opens a stream and links file with that stream. Then it returns the file pointer associated with that file. Syntax: **FILE *fopen(const char *filename, const char *mode);**

Where **filename** is a pointer to a string of characters which is a valid filename (may include file path). **mode** determines how the file will be opened. Table shows the legal values for **mode**. Strings like "**r+b**" may also represent as "**rb+.**"

The **fopen()** function returns a file pointer. Our program should never alter the value of this pointer. If an error occurs while opening file, **fopen()** returns a **null pointer**.

In example the **fopen()** is used to open a file named **test** for output. **fopen()** will detect an error while opening a file which is **write-protected** or when **disk is full**. That's why we need to confirm that **fopen()** succeeded before attempting to perform any operations on the file.

Mode	Meaning
r	Open a text file for reading.
W	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

```
FILE *fp;
if ((fp = fopen("test", "w")) == NULL)
{
    printf("Cannot open file.\n");
    exit(1);
}
```

When opening a file for **read-only** operations, the file does not exist, **fopen()** will fail. When opening a file using **append mode**, if the file does not exist, it will be created. Further, when a file is opened for **append**, all new data written to the file will be written to the **end of the file**.

The original contents will remain unchanged. If, when a file is opened for **writing**, the file does not exist, it will be created. If it does exist, the contents of the original file will be destroyed and a new file created. The difference between modes **r+** and **w+** is that **r+** will not create a file if it

Problem Solving with Programming Course Material

does not exist; however, **w+** will. If the file already exists, opening it with **w+** destroys its contents; opening it with **r+** does not.

File Close:

The **fclose()** function closes a stream that was opened by a call to **fopen()**. It writes any data still remaining in the disk buffer to the file closes the file. Failure to close a stream invites all kinds of trouble, including lost data, destroyed files, and possible intermittent errors in your program.

fclose() also frees the file control block associated with the stream, making it available for reuse.

There is an operating-system limit to the number of open files you may have at any one time, so you may have to close one file before opening another. The **fclose()** function has this prototype:

```
int fclose(FILE *fp);
```

Where **fp** is the file pointer returned by the call to **fopen()**. A successful close operation returns value **zero**. The function returns **EOF** if an error occurs. You can use the standard function **feof()** to determine and report any problems. Generally, **fclose()** will fail only when a disk has been prematurely removed from the drive or there is no more space on the disk.

FORMATTED INPUT AND OUTPUT FUNCTIONS

printf and **scanf** are the two standard functions which are used regularly in our programs. The **scanf** function receives a text stream from the **stdin** (keyboard) and stores stream in variables.

The **printf** function receives data from program which is converted into text stream which is displayed on the **stdout** (monitor).

Apart from displaying and reading text streams from keyboard and to monitor we can store or retrieve streams to or from files. Standard **c** provides two general purpose functions **fscanf** and **fprintf** which are available in **stdio.h** file.

Syntax: **int fprintf(FILE *fp, const char *control_string, . . .);**

int fscanf(FILE *fp, const char *control_string, . . .);

In example **fp** is a file pointer returned by a call to **fopen()** may be to any file. Here **s** is a string variable

```
fscanf(stdin, "%s%d", s, &t); /* read from keyboard */
fscanf(fp, "%s%d", s, &t); /* read from file */
fprintf(stdout, "%s %d", s, t); /* print on screen */
fprintf(fp, "%s %d", s, t); /* write to file */
```

Problem Solving with Programming Course Material

and **t** is an integer variable where we want to store or retrieve values i.e., a **string** and **int** value.

Format control strings: When we read or write the data to or from a file the functions **fscanf** and **fprintf** uses format strings, **%d** for int, **%s** for string are written in above examples i.e., conversion specification as part of format string. A format string may comprise of a **whitespace** or **Text characters** or the **conversion specifications**, which represents the type of data stored in a file.

➤ A **whitespace** or a tab '\t' character in **fscanf** function causes leading whitespaces in format string causes zero, one or more whitespaces in input stream to be omitted. Where as a whitespace characters or tab '\t' in **fprintf** function will copy whitespaces as it is to the file.

➤ Any **Text** character other than a whitespace in a **fscanf** function must exactly match with **fprintf** function that is, if we store first a int value then a string value in a file. We need to read in the same order other wise a conflict will raise which leads to abnormal termination, and the conflicting text will be available in the stream to be read by the next input operation.

Option Field	Particulars
%	Percentage
Flag	The only flag used is (*) input is discarded.
Maximum Field Size	Maximum number of characters can be read
Size	h -short; l -long; l -double; L -long double;
Conversion Code	d -int; s -string; c -char; f -float; u -unsigned int;...

➤ The **conversion specification** consists of a percent character '%', optional formatting instructions, and a conversion code.

A **scanf** or **fscanf** functions will contain beside conversion specification:

A **printf** or **fprintf** functions will contain beside conversion specification:

Option Field	Particulars
%	Percentage
Flag	'-' left justify; '+' right justify; '0' zero padding;
Minimum width	Minimum number of characters can be read
precision	Sets the maximum number of characters after '.' dot.
Size	h -short; l -long; l -double; L -long double;
Conversion Code	d -int; s -string; c -char; f -float; u -unsigned int;...

Input formatting (scanf & fscanf): The difference between **scanf** and **fscanf** is **scanf** reads data from **stdin** and **fscanf** reads data from the **first parameter** specified that is a stream. **scanf** means **Scan Formatted** and **fscanf** means **File Scan Formatted**.

Problem Solving with Programming Course Material

We must specify the address for every variable in order to store the data to that specified variable. If we don't do this the result is unpredictable. The data is read in `scanf` or `fscanf` till either it reaches end of file or in appropriate character is encountered (while reading an **int** value if a **char** is encountered) or the number of characters read is equal to explicitly specified maximum field width.

The conversion specifications of **scanf** and **fscanf** functions are:

- **Flag:** the only flag for input formatting allowed is * (**assignment suppression**) which tells `scanf` to read the data but not to store in variable. For example `scanf("%d%*c%d", &x, &y);`. The function will read an integer a character and then again an integer, only integer values are stored to `x` & `y` the character is discarded (not stored).
- **Width:** specifies the allowed maximum width of input in characters. That may allow us to display the content in a neat manner by storing the whitespaces if the characters are less than width. For example: `scanf("%3d",&n);` will read only 3 characters.
- **Size:** is a modifier for conversion code. Where **h** can be used for representing short data, **l** can be used for long and double as (`%ld` for **long** and `%lf` for **double** and `%Lf` for **long double**).

scanf and **fscanf** functions return **side effect** when it reads and converts a stream of characters from the input file and stores the converted values in the list of variables found in the address list. **scanf** and **fscanf** functions return **value** when it returns the number of successful data conversions if end of file is reached before any data are converted, it returns **EOF**.

The major **side effect** with `scanf` while reading an input stream is: the input stream is stored in the buffer by operating system without passing the data until we press the enter key. This means there is always there is a **return character** associated with the string at end in the buffer, this character is not read by the `scanf` function, means return character is still available in the buffer. When `scanf` reads the next value from buffer it must discard the existing return character available in buffer left by previous `scanf`. This can be done using **assignment suppression flag** `*?`.

Problem Solving with Programming Course Material

One more **concern** with scanf function is it won't terminate until the format string is completed, means if we want to read 3 values for example `scanf("%d%d%d", &a, &b, &c);`. This function will terminate only when it reads all the three values, means the scanf function terminates when the format string is completed. Whenever a whitespace is encountered in scanf it replaces a **return character** in the place of a whitespace.

Output formatting (printf & fprintf): These functions will print the data in human readable format. The format string of printf is similar to that of a scanf function but the working of both the functions is opposite.

The conversion specifications of **printf** and **fprintf** functions are:

- **Flag:** the justification flag controls the placement of a value when it is shorter than the specified value. Printf and fprintf allows only left '-' justification or right justification (if '-' is not specified).
- **Padding:** specifies how to fill the unused space when the value is smaller than the specified width. It can be a space (default) or may be a **0** (zero). For example: `printf("%05d",n);` if **n** has value **10**. The output will be **00010**.
- **Sign Flag:** is used with the numeric values to represent whether a value is a negative or positive value by using -(minus) or +(plus) signs. If the value is a space then positive value is printed.
- **Alternative Flag #:** is used with real, hexadecimal, octal conversion codes.
- **Precision:** can be used to specify the period followed by an integer value. The number of digits needs to be printed after the decimal point (zeros).
- **Width and Size:** same as scanf functions.

The major **side effect** with printf functions regarding a output stream is while writing text data that may be either strings or characters has to get converted to their equivalent values which may be printed on standard output or stored to a file. **printf** and **fprintf** functions returns **value** when it returns the number of characters written to output file. In case of it reaches end of file **EOF**.

TEXT VS BINARY FILES

Problem Solving with Programming Course Material

In c language files are of two types' text and binary files. Depending on our program requirement we can use any one of the type. Both the types are having their own importance and usage criteria.

Text Files:

In a text file data is stored as a sequence of characters, which is human readable. Text files are written using **text streams**. We can use **fprintf, fscanf** functions to handle text files, in both the cases text is converted to internal format based on the conversion specifications. We can also use character input output functions **getchar, putchar** for reading or writing character by character.

A text files can contain any type of data. There is no limit to the number of entries that we can write in a text file. We can also use white space (empty lines) throughout the file. We should note that surrounding spaces are not included with an individual line when it is stored or retrieved.

Binary Files:

A binary file is a collection of data stored that is only understandable by computer or the data is stored in a file as if it is stored in the same manner of that of file stored in memory. Binary data is read or written using **binary streams**. Here data is moved or retrieved in a **block manner**, not character by character as in text files.

Differences between Text & Binary Files:

In a text file all the data is human readable, each line ends with a new line character '**\n**' and every file ends with a special character called as end of file **EOF** character. Where as a binary file stores data in internal computer format i.e., a integer value is stored in 4 bytes a character value is stored in 1 byte and so on. There are no new line characters but end of file character is available in binary files.

For example if we want to store 123 a integer value in text files '1', '2', '3' are stored as independent characters in 3 bytes continuously, where as in binary files since 123 is a integer value it is stored as a single value in 4 bytes of memory.

Problem Solving with Programming Course Material

State of a file: A file may be in any one of the four states: **read** or **write** or **error** or **update** State.

- **Read State:** if we want our program to read data from a file then it must be in read state, means we need to open the file in a mode which leaves the file in a read state.
- **Write State:** if we want our program to write some data from our program to a file then the file must be in write state, means we need to open the file such that it leaves the file in a write state. If the file is not available with the name specified while opening, a new file is created with the same name.

To open a file in write state we can open a file either in write mode or append mode. If we open the file in **write** mode the data is written from beginning of file, if any data is already available will be lost. When a program opens the file in a **append** mode, if file exist then the new data will be written at the end of file, if file does not exist then a new file is created then the data is returned to the file.

- **Error State:** is resulted when an error is raised may be due to any reason: for example if program opens a file which doesn't exist in read mode. If program try's to write data in a file which is opened in write mode.
- **Update State:** to open a file in update mode we can use + symbol, for example: **r+** for read state, **w+** and **a+** the file will be in write state. Update state allows a file to be in both read state or write state, but at a time only one state is allowed.

FILE INPUT AND OUTPUT FUNCTIONS

To write binary data in a file we require functions which can perform block input and output. When we read or write binary files the data is similar to the data available in the memory storage (0 or 1). There are no format conversions for binary files means we cant use **fscanf** and **fprintf** functions (used only for text files). We use **fread** and **fwrite** functions for storing and retrieving the data to or from binary files.

fread & fwrite: To read and write data types that are longer than one byte (int, float, double,...) means a block of data, the C file system provides two functions: **fread()** and **fwrite()**. These

Problem Solving with Programming Course Material

functions allow the reading and writing of blocks of any type of data to or from a binary file.

Their prototypes are:

```
size_t fread(void *buffer, size_t num_bytes, size_t count, FILE *fp);
```

```
size_t fwrite(const void *buffer, size_t num_bytes, size_t count, FILE *fp);
```

For **fread()**, **buffer** is a pointer to a region of memory that will receive the data from the file. For **fwrite()**, **buffer** is a pointer to the information that will be written to the file. The value of **count** determines how many items are read or written, with each item being **num_bytes** bytes in length. (Remember, the type **size_t** is defined as some type of unsigned integer.) Finally, **fp** is a file pointer to a previously opened stream. The **fread()** function returns the number of items read. This value may be less than **count** if the end of the file is reached or an error occurs. The **fwrite()** function returns the number of items written. This value will equal to **count** unless an error occurs.

As long as the file has been opened for binary data, **fread()** and **fwrite()** can read and write any type of information. One of the most useful applications of **fread()** and **fwrite()** involves reading and writing user-defined data types, especially structures.

Writing a Character: The **putc()** function writes characters to a file that was previously opened for writing using the **fopen()** function. The prototype of this function is: **int putc(int ch, FILE *fp);** where **fp** is the file pointer returned by **fopen()** and **ch** is the character to be output. The file pointer tells **putc()** which file to write to. If a **putc()** operation is successful, it returns the character written. Otherwise, it returns **EOF**.

Reading a Character: The **getc()** function reads characters from a file opened in read mode by **fopen()**. The prototype of **getc()** is: **int getc(FILE *fp);** where **fp** is a file pointer of type **FILE** returned by **fopen()**. **getc()** returns an integer, unless an error occurs, and returns an **EOF** when the end of the file has been reached. Therefore, to read to the end of a text file, you could use the following code:

```
do {  
    ch = getc(fp);  
} while(ch != EOF);
```

However, **getc()** also returns **EOF** if an error occurs. We can use **ferror()** to determine what happened.

Problem Solving with Programming Course Material

FILE STATUS FUNCTIONS

Feof: is used to check whether the end of file has been encountered. The file system can operate on both text and binary files. When a file is opened for binary input, an integer value that will test equal to **EOF** may be read. This would cause the input function to indicate an end-of-file condition even though the physical end of the file had not been reached.

Function **getc()** also returns **EOF** when it fails and when it reaches the end of the file. Using only the return value of **getc()**, it is impossible to know which occurred. To solve these problems, the C file system includes the function **feof()**, which determines when the end of the file has been encountered.

The **feof()** prototype: **int feof(FILE *fp)**; this returns **true** if the end of the file has been reached; otherwise, it returns **0**. Therefore, the following example reads a binary file until the end of the file is encountered: **while(!feof(fp)) ch = getc(fp)**;

Ferror: function determines whether a file operation has produced an error. The **ferror()** function has this prototype: **int ferror(FILE *fp)**; where **fp** is a valid file pointer. It returns **true** if an error has occurred during the last file operation; otherwise, it returns **false**. Because each file operation sets the error condition, **ferror()** should be called after each file operation; otherwise, an error may be lost.

Clearer: function resets (sets to zero) the error flag associated with the stream (**ferror**) pointed to by stream. The **EOF** indicator is also reset. The error flags for all streams are initially set to zero at the successful call to **fopen()**. Once an error has occurred, the flags are modified until an explicit call to either **clearerr()** or **rewind()** is made. The **clearerr()** function's prototype: **void clearerr(FILE *fp)**;

Where **fp** is a valid file pointer.

Rewind: function moves the file position indicator to the start of the specified stream. It also clears the end-of-file and error flags associated with stream. It has no return value. The **rewind()** functions prototype is: **void rewind(FILE *fp)**; where **fp** is a valid file pointer.

Problem Solving with Programming Course Material

Rename: function changes the name of the file specified by **oldfname** to **newfname**. The **newfname** must not match any existing directory entry. The **rename()** function returns zero if successful and nonzero if an error has occurred. The **rename()** functions prototype is:

int rename(const char *oldfname, const char *newfname);

Remove: function erases the specified file. Its prototype is **int remove(const char *filename);** This function returns **zero** if successful; otherwise, it returns a **nonzero** value.

If we wish to **flush** the contents of an output stream, use the **fflush()** function, whose prototype is: **int fflush(FILE *fp);** This function writes the contents of any buffered data to the file associated with **fp**. If we call **fflush()** with **fp** being **null**, all files opened for output are flushed. The **fflush()** function returns **0** if successful; otherwise, it returns **EOF**.

Tmpfile: function opens a temporary file for update & returns a pointer to the stream. It automatically uses a unique filename to avoid conflicts with existing files. **tmpfile()** functions prototype is **FILE *tmpfile(void);** it returns a null pointer on failure; otherwise it returns a pointer to the stream. The temporary file created by **tmpfile()** is automatically removed when file is closed or program terminated.

POSITIONING FUNCTIONS

Fseek: We can perform random-access read and write operations using **fseek()**, which sets the file position indicator. Its prototype is: **int fseek(FILE *fp, long numbytes, int origin);** Here, **fp** is a file pointer returned by a call to **fopen()**. **numbytes** is the

number of bytes from **origin** that will become the new current position, and **origin** is one of the macros.

Macro Name	Meaning
SEEK_SET	Beginning of file
SEEK_CUR	Current position
SEEK_END	End of file

To seek **numbytes** from the start of the file, **origin** should be **SEEK_SET**. To seek from the current position, use **SEEK_CUR**; and to seek from the end of the file, use **SEEK_END**. The **fseek()** function returns **0** when successful and a **nonzero** value if an error occurs. We can use **fseek()** to seek in multiples of any type of data by simply multiplying the size of the data by the number of the item you want to reach. For example: **fseek(fp, 9*sizeof(char), SEEK_SET);** will

Problem Solving with Programming Course Material

seek to the tenth address in the file that holds the addresses. We can determine the current location of a file using **ftell()**.

Ftell: We can determine the current location of a file using **ftell()** . Its prototype: **long ftell(FILE *fp)**; It returns the location of the current position of the file associated with **fp**. If a failure occurs, it returns **-1**. Random access is mostly implemented on binary files. The reason is text files may have character translations performed on them, there may not be a direct correspondence between what is in the file and the byte to which it would appear that we want to seek. The only time you should use **fseek()** with a text file is when seeking to a position previously determined by **ftell()** , using **SEEK_SET**.

Sample Projecjts:

1. Write a C program to read name and marks of n number of students from user and store them in a file.

```
#include <stdio.h>
int main()
{
    char name[50];
    int marks, i, num;

    printf("Enter number of students: ");
    scanf("%d", &num);

    FILE *fptr;
    fptr = (fopen("C:\\student.txt", "w"));
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    for(i = 0; i < num; ++i)
    {
```

Problem Solving with Programming Course Material

```
printf("For student%d\nEnter name: ", i+1);
scanf("%s", name);

printf("Enter marks: ");
scanf("%d", &marks);

fprintf(fptr, "\nName: %s \nMarks=%d \n", name, marks);
}

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

2. Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exists, add the information of n students.

```
#include <stdio.h>

int main()
{
    char name[50];
    int marks, i, num;

    printf("Enter number of students: ");
    scanf("%d", &num);

    FILE *fptr;
    fptr = (fopen("C:\\student.txt", "a"));
    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    for(i = 0; i < num; ++i)
```

Problem Solving with Programming Course Material

```
{
    printf("For student%d\nEnter name: ", i+1);
    scanf("%s", name);

    printf("Enter marks: ");
    scanf("%d", &marks);

    fprintf(fp, "\nName: %s \nMarks=%d \n", name, marks);
}

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

3. Write a C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.

```
#include <stdio.h>
struct student
{
    char name[50];
    int height;
};
int main(){
    struct student stud1[5], stud2[5];
    FILE *fp;
    int i;

    fp = fopen("file.txt", "wb");
    for(i = 0; i < 5; ++i)
    {
        fflush(stdin);
        printf("Enter name: ");
```


Problem Solving with Programming Course Material

```
gets(stud1[i].name);

printf("Enter height: ");
scanf("%d", &stud1[i].height);
}

fwrite(stud1, sizeof(stud1), 1, fptr);
fclose(fptr);

fptr = fopen("file.txt", "rb");
fread(stud2, sizeof(stud2), 1, fptr);
for(i = 0; i < 5; ++i)
{
    printf("Name: %s\nHeight: %d", stud2[i].name, stud2[i].height);
}
fclose(fptr);
}
```

UNIT V : Topics Included

Memory Management

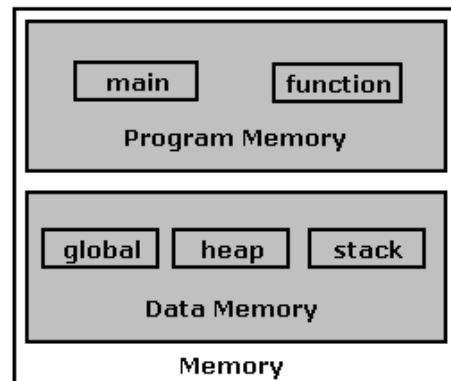
Memory Management: Dynamic memory allocation and deallocation functions:- malloc(), calloc(), realloc() and free()-examples & discussion for each. Low-level programming, register variables, bitwise operations, bit fields.

Memory Management

We usually create variables when we write programs and the memory for these variables will be allocated by compiler at runtime implicitly, if we want to allocate memory explicitly we can do so using any one of the two ways: Static Memory Allocation and Dynamic Memory Allocation.

Memory usage:

When we execute a program, compiler requires some memory to execute the program, where in a



Problem Solving with Programming Course Material

program we write functions, we declare some variables, we sometimes write recursive functions which intern requires stack memory, and so on. Memory is allocated depending on how the compiler is written (16 bit compiler, 32 bit compiler...) and on the operating system which is used. Conceptually memory allocated for executing a program is divided into two types **Data Memory** and **Program Memory**.

- **Program Memory** is allocated for all the available functions (including main()) in a program. Main is the first & the last function which is available in memory, and if other functions are called in the program all such functions must also be available in program memory. If a function is called more than one time only one copy of function is available.
- **Data Memory** is allocated for the variables created (global data, local data and dynamic data) in a program. Local variables are active only when the function block in which they are created is active. If a function is called more than one time those many copies of local variables are created in stack memory.
- **Heap memory** is a predefined area of memory which can be accessed by the program to store data and variables. The data and variables are allocated on the heap by memory allocation functions (**malloc, calloc, realloc**). The system keeps track of where the data is stored. Data and variables can be deallocated from the heap. The system knows where the free space is and will use them for additional data storage. We can refer to heap memory only using a pointer.

Dynamic memory allocation and de-allocation functions

Dynamic memory allocation uses predefined functions for allocating and de-allocating the memory at runtime on Heap memory, which can be accessed only through pointers. There are many situations when you do not know the exact sizes of arrays used in our programs till they got executed. You can specify the sizes of arrays in advance, but the arrays can be too small or too big. If the numbers of data items you want to put into the arrays changes at runtime, we can do so using dynamic memory allocation functions.

Memory allocation functions:

Problem Solving with Programming Course Material

C provides four dynamic memory allocation functions that we can use to allocate or reallocate memory spaces while our program is running. We can also release allocated memory as soon as we don't need it. These four C functions are `malloc()`, `calloc()`, `realloc()`, and `free()` which are available in **stdlib.h** header file which we need to include if we want to use any one of these functions.

- 1) **malloc:** **malloc** function **allocates memory at runtime**. It takes the size in bytes and allocates that much space in the memory. It means that `malloc(50)` will allocate 50 byte in the memory. It returns a void pointer and is defined in **stdlib.h**.

Let's understand it with the help of an example.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char name[20];
        char *address;
    strcpy(name, "SREC");
    /* allocating memory dynamically */
    address = (char*)malloc( 50 * sizeof(char) );
    strcpy( address, "Warangal, Telanagana State");
    printf("Name = %s\n", name );
    printf("Address: %s\n", address );
    return 0;
}
```

Output:

Name = SREC

Address: Warangal, Telangana State

In the above example, we assigned and printed the name as we used to do till now. For address, we estimated that the number of characters should be around 50. So, the **size of address** will be **50 * sizeof(char)**.

Problem Solving with Programming Course Material

char *address - Here we declared a pointer to character for address without specifying how much memory is required.

```
address = (char*)malloc( 50 * sizeof(char) )
```

By writing this, we assigned a memory of '**50 * sizeof(char)**' bytes for address. We used (**char***) to typecast the pointer returned by malloc to character.

```
strcpy( address, "Warangal, Telangana State")
```

By writing this, finally we assigned the address.

Note: By default, malloc returns a pointer of type void but we can typecast it into a pointer of any other form (as we converted it into character type in the above example).

Note: If the space in memory allocated by malloc is insufficient, then the allocation fails and malloc returns NULL pointer

Let's see another example.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)malloc(n * sizeof(int)); //memory allocated using malloc
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
        printf("Enter elements of array:\n");
        for(i=0;i<n;++i)
            scanf("%d",&*(p+i));
        printf("Elements of array are\n");
        for(i=0;i<n;i++)
```

Problem Solving with Programming Course Material

```
    printf("%d\n",*(p+i));  
}  
return 0;  
}
```

Output:

Enter number of elements:5

Enter elements of array:

1

2

3

4

5

Elements of array are

1

2

3

4

5

In this example, firstly, we declared a pointer **p** of type **int** which contains **n elements**. Thus, **p** is an integer array containing **n** elements. So, we assigned a memory of size **n * sizeof(int)** to the array which the pointer 'p' is pointing to. Thus, we now have a memory space allocated to the integer array consisting of 'n' elements. Further, we ask the user to input the values of the elements of the array and display those values.

- 2) **Calloc:** Now, suppose you want to put more than one toy in a box and you have only an approximate idea of the number of toys and the size of each. For that, you need a box the size of which is equal to the sum of the sizes of all the toys.

In such cases, we use **calloc** function. Like **malloc**, **calloc** also allocates memory at runtime and is defined in **stdlib.h**. It takes the number of elements and the size of each element(in bytes), initializes each element to zero and then returns a void pointer to the memory. Its syntax is: **void *calloc(n, element-size);**

Problem Solving with Programming Course Material

Here, n is the number of elements and element-size is the size of each element. Let's see the last example of malloc again, but this time with calloc.

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

int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n, sizeof(int)); //memory allocated using malloc
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
        printf("Enter elements of array:\n");
        for(i=0;i<n;++i)
        {
            scanf("%d",&*(p+i));
        }
        printf("Elements of array are\n");
        for(i=0;i<n;i++)
        {
            printf("%d\n",*(p+i));
        }
    }
    return 0;
}
```

Output:

Enter number of elements:5

Enter elements of array:

Problem Solving with Programming Course Material

1

2

3

4

5

Elements of array are

1

2

3

4

5

So, this is the same as the example of malloc, with a difference in the syntax of calloc. Here we wrote `(int*)calloc(n, sizeof(int))`, where n is the number of elements in the integer array (5 in this case) and sizeof(int) is the size of each of that element. So the total size of the array is $n * \text{sizeof(int)}$.

Note: calloc initializes the allocated memory to zero value whereas malloc doesn't.

calloc is used to allocate memory to mostly arrays and structures.

- 3) **Realloc:** If suppose we allocated more or less memory than required, then we can change the size of the previously allocated memory space using realloc. Its syntax is as follows.

```
void *realloc(pointer, new-size);
```

Let's see an example on realloc.

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

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char *p1;
```

```
    int m1=10, m2=20;
```

```
    p1 = (char*)malloc(m1);
```

```
    strcpy(p1, "Old String");
```

Problem Solving with Programming Course Material

```
p1 = (char*)realloc(p1, m2);
strcat(p1, "New String");
printf("%s\n", p1);
return 0;
}
```

Output: Old StringNew String

In the above example, we declared a pointer **p1** which will be used to dynamically allocate a memory space: **p1 = (char*)malloc(m1)** - By writing this, we assigned a memory space of 10 bytes which the pointer p1 is pointing to. We used (char*) to typecast the pointer returned by malloc to character.

strcpy(p1, "Old String") - This assigns a string value "Old String" to the memory which the pointer p1 is pointing to. Currently, the memory space is 10 bytes which can easily store the string "Old String", but what if now we want that memory space to store the string "Old StringNew String"? For this, we need to expand the size of our memory space which we can easily do with **realloc**.

p1 = (char*)realloc(p1, m2) - This increases the size of the memory space (whose address is stored in p1) to 20 bytes(since the value of m2 is 20) which can easily store the string "Old StringNew String".

strcat(p1, "New String") - This adds the string "New String" at the end of the string stored in the memory pointed by p1 i.e. "Old String". So now, the memory pointed by p1, now stores the string value "Old StringNew String".

- 4) **free:** function is used to deallocate or free the memory after the program finishes which was dynamically allocated in the program. It is advised to free the dynamically allocated memory after the program finishes so that it becomes available for future use.

void free(pointer);

This was the syntax of free function whose return type is void. Now, let's see an example where we released the dynamically allocated memory at the end of the program using free.

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


Problem Solving with Programming Course Material

```
int main()
{
    int n,i,*p;
    printf("Enter number of elements:\n");
    scanf("%d",&n);
    p=(int*)malloc(n * sizeof(int));
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
        printf("Enter elements of array:\n");
        for(i=0;i<n;++i)
        {
            scanf("%d",&*(p+i));
        }
        printf("Elements of array are\n");
        for(i=0;i<n;i++)
        {
            printf("%d\n",*(p+i));
        }
    }
    free(p);
    return 0;
}
```

Output:

Enter number of elements:5

Enter elements of array:

- 1
- 2
- 3
- 4

Problem Solving with Programming Course Material

5

Elements of array are

1

2

3

4

5

So here by writing **free(p)**, we released the memory which was dynamically allocated using malloc.

Low-level programming

In addition to the high level language constructs, (data type and operators), C also supports **low level** programming features which enable the programmer to carry out bit-wise operations. These features are normally provided in *assembly language* or *machine language*.

A low-level programming language is a programming language that provides little or no abstraction from a computer's instruction set architecture—commands or functions in the language map closely to processor instructions. Generally this refers to either machine code or assembly language. The word "low" refers to the small or nonexistent amount of abstraction between the language and machine language; because of this, low-level languages are sometimes described as being "close to the hardware". Programs written in low-level languages tend to be relatively non-portable, mainly because of the close relationship between the language and the hardware architecture.

Low-level languages can convert to machine code without a compiler or interpreter— second-generation programming languages use a simpler processor called an assembler— and the resulting code runs directly on the processor. A program written in a low-level language can be made to run very quickly, with a small memory footprint. An equivalent program in a high-level language can be less efficient and use more memory. Low-level languages are simple, but considered difficult to use, due to numerous technical details that the programmer must remember. By comparison, a high-level programming language isolates execution semantics of a

Problem Solving with Programming Course Material

computer architecture from the specification of the program, which simplifies development.

Low-level programming languages are sometimes divided into two categories: first generation, and second generation.

Register Variables

C supports four different storage classes, viz. static, auto, extern and register. As such, general purpose registers are special storage areas within the Central Processing Unit (CPU). The CPU registers are used for temporarily holding intermediate results generated by the Arithmetic Logic Unit (ALU). It also stores information that are transferred from the main memory of the computer for further processing, this reduces the traffic between CPU and RAM which inevitably leads to higher degree of efficiency.

In C, the content of register variables reside inside registers. A program that uses register variables execute faster since their values are stored inside the registers within the CPU rather than in the RAM. A variable can be declared of storage class type *register* by prefixing the variable declaration by the keyword register. For example,

```
register int cnt = 0;
```

However, only a few register variables can effectively be used in a C function. The exact number of register variable declarations possible in a function is Machine dependent. The scope of a register variable is identical to that of auto type variables.

It is not always the case that a variable defined to be of storage class *register* has to be a register variable only. The declaration is valid only when the requested register space is available, otherwise the variable declared to have storage class register, will be treated as automatic variable only. The important distinction between an automatic and register variable is that a register variable can never be preceded by the unary operator because a register variable does not have a l-value.

Bitwise Operations

Problem Solving with Programming Course Material

A **bitwise operation** operates on one or two bit patterns or binary numerals at the level of their individual bits. Bitwise operations are slightly faster than addition and subtraction operations and usually significantly faster than multiplication and division operations.

Bitwise operations are necessary for much low-level programming, such as writing device drivers, low-level graphics, communications protocol packet assembly and decoding.

Although machines often have efficient built-in instructions for performing arithmetic and logical operations, in fact all these operations can be performed just by combining the bitwise operators and zero-testing in various ways.

The available bitwise operators are:

1. **NOT:** - The **bitwise NOT**, or **complement**, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Digits which were 0 become 1, and vice versa. In many programming languages the bitwise NOT operator is "~" (**tilde**). This operator must not be confused with the "logical not" operator, "!" (exclamation point).

NOT 0111 (decimal 7) = 1000 (decimal 8)
--

2. **OR:** - A **bitwise OR** takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits (the first of each; the second of each; and so on) and performing the logical **inclusive OR** operation on each pair of corresponding bits. In each pair, the result is **1** if the first bit is **1** **OR** the second bit is **1** **OR** both bits are **1**, and otherwise the result is **0**. The bitwise OR operator is "|" (**pipe**). Again, this operator must not be confused with its Boolean "logical or" counterpart, which treats its operands as Boolean values, and is written "||" (two pipes).

0101 (decimal 5) OR 0011 (decimal 3) = 0111 (decimal 7)

3. **XOR:** - A **bitwise exclusive or** takes two bit patterns of equal length and performs the logical **XOR** operation on each pair of corresponding bits. The result in each position is **1** if the two bits are different, and **0** if they are the same. The bitwise **XOR** operator is "^" (**caret**).

0101 XOR 0011 = 0110

4. **AND:** - A **bitwise AND** takes two binary representations of equal length and performs the logical **AND** operation on each pair of corresponding bits. In each pair, the result is **1** if the first bit is **1** **AND** the second bit is **1**. Otherwise, the result is **0**. The **bitwise AND** operator is "&" (**ampersand**). Again, this operator

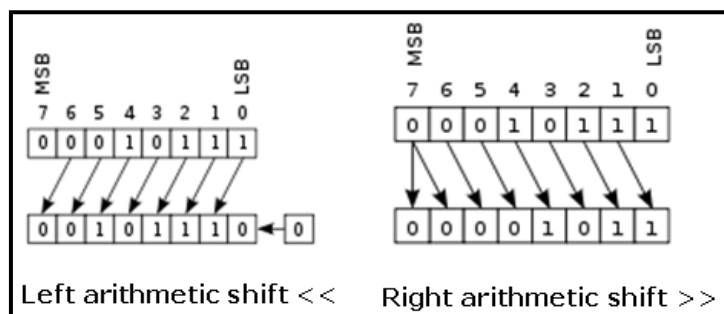
0101 AND 0011 = 0001

Problem Solving with Programming Course Material

must not be confused with its Boolean "logical and" counterpart, which treats its operands as Boolean values, and is written "&&" (two ampersands).

5. **Bit shifts:**- The bit shifts are sometimes considered **bitwise operations**, since they operate on the binary representation of an integer instead of its numerical value; however, the bit shifts do not operate on pairs of corresponding bits, and therefore cannot properly be called bit-wise operations. In this operation, the digits are moved, or shifted, to the left or right.

In an **arithmetic shift**, the bits that are shifted out of either end are discarded. In a left arithmetic shift, zeros are shifted in on the right; in a right arithmetic shift. This example uses an 8-bit register.



In the first case, the leftmost digit was shifted past the end of the register, and a new **0** was shifted into the rightmost position. In the second case, the rightmost **1** was shifted out, and a new **0** was copied into the leftmost position. Multiple shifts are sometimes shortened to a single shift by some number of digits. For example:

```
00010111 LEFT-SHIFT-BY-TWO
= 01011100
```

The left and right shift operators are "<<" and ">>", respectively. The number of places to shift is given as the second argument to the shift operators. For example: $x = y \ll 2;$ assigns x the result of shifting y to the left by two bits.

For example showing how to multiply two integers a and b (a less than b) uses bitshifts and addition:

```
void main()
{ int a=5,b=6;
  int c=0;
  while(b!=0)
  {
    if((b&1)!=0)
```

Problem Solving with Programming Course Material

```
{ c=c+a; }  
a=a<<1;  
b=b>>1;  
}  
printf("%d", c);  
}
```

Bit Fields

Bitfields can only be declared inside a structure or a union, and allow you to specify some very small objects of a given number of bits in length. Their usefulness is limited and they aren't seen in many programs. **Bit fields** do not have addresses so we can't have pointers to them or arrays of them.

A **bit field** is different from a **bit array** which is used to store a large set of bits indexed by integers and is often wider than any integral type. Bit fields typically fit within a machine word, and the denotation of bits is independent of their numerical index.

However, **bit members** in **structs** have practical drawbacks. First, the ordering of bits in memory is **architecture dependent** and **memory padding** rules vary from compiler to compiler. In addition, many popular compilers generate inefficient code for reading and writing bit members, and there are potentially thread safety issues relating to bit fields due to the fact that most machines cannot manipulate arbitrary sets of bits in memory, but must instead load and store whole words.

Use of Bitfields:

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if we use them our program will be non-portable and will be compiler-dependent too.

The Standard says that fields are packed into **storage units**, which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are

Problem Solving with Programming Course Material

implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned.

```
struct foo
{ int flag : 1;
  int counter : 15;
};
void main()
{ struct foo my_foo;
  my_foo.flag = !my_foo.flag;
  printf("\n\tflag = %d",my_foo.flag);
  printf("\n\tCounter = %d",my_foo.counter);
  ++my_foo.counter;
  printf("\n\tCounter = %d",my_foo.counter);
}
```

UNIT VI : Topics Included

Developing Large Programs

Some Additional features of C, enumerations, command line parameters, more about library functions, macros, C preprocessor. Pre-processor directives: #define, #undef, #if, #endif, #elif, #ifdef, #ifndef, #error.

Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can

create variables of enum type.

```
enum boolean { false, true };
```

```
enum boolean check;
```

```
// Changing default values of enum
```

```
enum suit
```

```
{
```

```
    club = 0,
```

```
    diamonds = 10,
```

Problem Solving with Programming Course Material

```
hearts = 20,  
spades = 3,  
};
```

- An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword `enum` is used.
- `enum flag { const1, const2, ..., constN };` Here, name of the enumeration is *flag*.
- And, *const1, const2, ..., constN* are values of type *flag*.
- By default, *const1* is 0, *const2* is 1 and so on. You can change default values of enum elements during declaration (if necessary).

Here, a variable *check* of type **enum boolean** is created.

Here is another way to declare same *check* variable using different syntax.

```
enum boolean { false, true } check;  
#include <stdio.h>  
enum week { sunday, monday, tuesday, wednesday, thursday, friday, saturday };  
int main()  
{  
    enum week today;  
    today = wednesday;  
    printf("Day %d",today+1);  
    return 0;  
}
```

Why enums are used in C programming?

Enum variable takes only one value out of many possible values. Example to demonstrate it,

```
#include <stdio.h>  
enum suit  
{  
    club = 0, diamonds = 10,  
    hearts = 20,  
    spades = 3  
} card;
```


Problem Solving with Programming Course Material

```
int main()
{
    card = club;
    printf("Size of enum variable = %d bytes", sizeof(card));
    return 0;
}
```

1. It's because the size of an integer is 4 bytes.
2. This makes enum a good choice to work with flags.
3. You can accomplish the same task using structures. However, working with enums gives you efficiency along with flexibility.

Command line arguments

- It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.
- The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly

A sample Program

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 )
    {
        printf("Too many arguments supplied.\n");
    }
}
```

Problem Solving with Programming Course Material

```
else
{
    printf("One argument expected.\n");
}
}
```

- When the above code is compiled and executed with single argument, it produces the following result.
- `./a.out testing` The argument supplied is testing When the above code is compiled and executed with a two arguments, it produces the following result.
- `./a.out testing1 testing2` Too many arguments supplied. When the above code is compiled and executed without passing any argument, it produces the following result.
- `./a.out` One argument expected It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and ***argv[n]** is the last argument. If no arguments are supplied, **argc** will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ". Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes –

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
    printf("Program name %s\n", argv[0]);
    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 )
    {
        printf("Too many arguments supplied.\n");
    }
}
```

Problem Solving with Programming Course Material

```
else
{
    printf("One argument expected.\n");
}
}
```

- When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.
- `./a.out "testing1 testing2"` Program name `./a.out` The argument supplied is testing1 testing2

C Standard library functions

- C Standard library functions or simply C Library functions are inbuilt functions in C programming.
- The prototype and data definitions of the functions are present in their respective header files, and must be included in your program to access them.
- **For example:** If you want to use `printf()` function, the header file `<stdio.h>` should be included.

```
#include <stdio.h>
int main()
{
    // If you use printf() function without including the <stdio.h> // header file, this program will
    show an error.
    printf("Catch me if you can.");
}
```

Advantages of using C library functions

There are many library functions available in C programming to help you write a good and efficient program. But, why should you use it?

Problem Solving with Programming Course Material

Below are the 4 most important advantages of using standard library functions.

1. They work

One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better.

In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

It saves valuable time and your code may not always be the most efficient.

3. The functions are portable

With ever changing real world needs, your application is expected to work every time, everywhere.

And, these library functions help you in that they do the same thing on every computer.

This saves time, effort and makes your program portable.

Use Of Library Function To Find Square root

- Suppose, you want to find the square root of a number.
- You can always write your own piece of code to find square root but, this process is time consuming and it might not be the most efficient process to find square root.
- However, in C programming you can find the square root by just using `sqrt()` function which is defined under header file "math.h"

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

```
#include <math.h>
```

```
int main()
```

```
{
```

```
    float num, root;
```

```
    printf("Enter a number: ");
```

```
    scanf("%f", &num);
```

Problem Solving with Programming Course Material

```
// Computes the square root of num and stores in
root. root = sqrt(num);
printf("Square root of %.2f = %.2f", num, root); return 0;
}
```

C Library Functions Under Different Header File

C Header Files	
<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions
<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

C preprocessor - Pre-processor directives

Problem Solving with Programming Course Material

- `#define` preprocessor directive is the most useful preprocessor directive in C language. We use it to define a name for particular value/constant/expression.

- C preprocessor processes the defined name and replace each occurrence of a particular string/defined name (macro name) with a given value (macro body).

Syntax to define a MACRO using `#define`

`#define MACRO_NAME macro_body`

- Here, `MACRO_NAME` is a C identifier and `macro_body` is string/value/expression.

```
#include <stdio.h> // Macro definition
#define COUNTRY "INDIA" // String constant
#define TRUE 1 // Integer constant
#define FALSE 0 // Integer constant
#define SUM (10 + 20) // Macro definition
int main()
{
    printf("COUNTRY: %s\n", COUNTRY);
    printf("TRUE: %d\n", TRUE);
    printf("FALSE: %d\n", FALSE);
    printf("SUM(10 + 20): %d\n", SUM);
    return 0;
}
```

- Here, `COUNTRY`, `TRUE` and `FALSE` are compile time constants and `SUM` is a macro. The C pre-processor replaces all occurrences of `COUNTRY` with "India" and all other constants with their respective value before compilation.

`#undef` preprocessor directive

- We use `#undef` directive to remove a defined macro. We generally un-define a macro, when we do not require or want to redefine it. To redefine a macro, we need to undefine the macro and redefine it.

Syntax:

`#undef MACRO_NAME`

Where `MACRO_NAME` is the macro to remove

```
#include <stdio.h> // Macro definition
```

```
#define TRUE 1
```

Problem Solving with Programming Course Material

```
#define FALSE 0
int main()
{
    printf("TRUE: %d\n", TRUE);
    printf("FALSE: %d\n", FALSE); // Undefine a previously defined macro
    #undef TRUE
    #undef FALSE // Re-define macro values
    #define TRUE 0
    #define FALSE 1
    printf("\nMacro values are redefinition\n");
    printf("TRUE: %d\n", TRUE);
    printf("FALSE: %d\n", FALSE);
    return 0;
}
```

Parameterized Macros (Function like Macros)

- We use Function like macros to rewrite small functions using macros to save the execution time.
- If we use functions, then at every function call programs execution will move between function definition and calling, which consumes time.
- To save this time, we can use parameterized macros. The macro definition will be expanded to the function call during pre-processing.

- **Syntax:**

```
#define MACRO_NAME(parameters) macro_body
#include <stdio.h> // Function to find sum of two numbers
int sum (int a, int b)
{
    return (a + b);
} // Function like Macro for above function #define #define SUM(a, b) (a + b)
int main()
{
    printf("SUM using function: %d\n", sum(100, 200));    printf("SUM using macro: %d\n",
    SUM(100, 200));
```

Problem Solving with Programming Course Material

```
return 0;
}
```

#ifdef, #ifndef and #endif conditional preprocessor directive

- We use conditional directives to check if a macro is defined or not. Both `#ifdef` and `#ifndef` are complements of each other.

- **#ifdef** will compile all code if a given macro is defined.

- **Syntax:**

```
#ifdef MACRO_NAME
```

Where `MACRO_NAME` is our macro to test if defined.

#ifndef conditional directive

- **#ifndef** is similar to `#ifdef` just works as a complement. It process block of code if a given macro is not defined.

- **Syntax:**

```
#ifndef MACRO_NAME
```

Where `MACRO_NAME` is our macro to test if not defined.

```
#include <stdio.h> // MACRO definition
```

```
#define COUNTRY "INDIA"
```

```
int main()
```

```
{ // If COUNTRY is defined, print a message
```

```
    #ifdef COUNTRY
```

```
        printf("Country is defined\n");
```

```
    #endif // If STATE is not defined, define it
```

```
    #ifndef STATE
```

```
        printf("State is not defined. Defining state. \n");
```

```
        #define STATE "PATNA"
```

```
    #endif
```

```
        printf("State is: %s\n", STATE);
```

```
return 0;
```

```
}
```

- **NOTE:** Conditional directive `#ifdef` or `#ifndef` must end with `#endif`. `#endif` defines scope of the conditional directive.

- We use conditional directives to restrict file loading multiple times.

Problem Solving with Programming Course Material

#if, #elif, #else and #endif preprocessor directives

- C supports conditional compilation pre-processors directives, similar to if...else statements in C. We use them for conditional compilations. C conditional preprocessor directives compiles multiple conditional code blocks.

Syntax:

```
#if expression
// If condition is true
#elif expression
// If else if condition is true
#else
// If no condition is true
#endif
```

A Sample Program

```
#include <stdio.h>
#define IND 1
#define USA 2
#define UK 3
#define COUNTRY IND
int main()
{
    #if COUNTRY == IND
        printf("Selected country code is: %d\n", COUNTRY);
        // Do some task if country is India
    #elif COUNTRY == USA    printf("Selected country code is: %d\n", COUNTRY);
        // Do some task if country is USA
    #else printf("Selected country code is: %d\n", COUNTRY);
        // Do some task if country is UK
    #endif
    return 0;
}
```