Chapter Table of Contents

Chapter 1.2

Pointers and Recursion

Aim

To provide the students with the knowledge of Pointers and Recursion

Instructional Objectives

After completing this chapter, you should be able to:

- Demonstrate the role of Pointers in data structures
- Explain memory allocation functions
- Explain Recursion and its advantages
- Describe how variables are accessed through pointers

Learning Outcomes

At the end of this chapter, you are expected to:

- Outline the steps to declare and initialise pointers
- List the advantages of Recursion
- Differentiate malloc() and calloc() and realloc()
- Write programs for binomial coefficient and Fibonacci using recursion

1.2.1 Introduction to Pointers and Recursive functions

In any programming language, creating and accessing variables is very important. So far we have seen how to access variables using their variable names. In this chapter we introduce the concept of indirect access of objects using their address. This chapter describes the use of Pointers in accessing the variables using their memory addresses. As memory is a resource of a computer, it should be allocated and deallocated properly. This chapter also describes the different memory allocation techniques.

Further we introduce the concept of Recursion where a function calls itself again and again to complete a task. We will also study some Recursive functions and their program implementations. We will also look at the application of Recursion to various problems like factorial, GCD, Fibonacci series etc.

1.2.2 Declaring and Initializing Pointers

Introduction to pointers and its need in programming

In all the previous programs, we referred to a variable with its variable name. Hence, the program did not care about the physical address of those variables. So, whenever we need to use a variable, we access them using the identifier which describes that variable.

Computers memory is divided into cells or locations which has unique addresses. Every variable that we declare in our program has an address associated with it. Thus a variable can also be accessed using the address of that variable. This can be achieved by using Pointers.

Pointers can be defined as the special variables which have a capability to store address of any variable. They are used in C++ programs to access the memory and manipulate the data using addresses.

Pointers are a very important feature of C++ Programming as it allows to access data using their memory addresses and not directly using their variable names. Pointers do not have much significance when simple primitive data types like integer, character or float are used. But as the data structures become more complex, pointers play a very vital role in accessing the data.

For example, consider an integer variable “a”. This variable will have 3 things associated with itself. First is its name, second is its value and third is the memory address. Assume that a variable “a” is having value “5” and address as “1000”. Hence we can access this variable “a” by using a pointer variable which will store the address of variable a. Thus we can manipulate values of any variable using a pointer variable.

Pointers are used for dynamic memory allocation so as to handle a huge amount of data. It would have been very difficult to allocate memory globally or through functions, without pointers.

Declaration and Initialization of Pointer Variables

Like any other variable in C++, pointer variables should be also declared before they are used for storing addresses.

In this chapter we are going to study 2 operators known as Pointer Operators:

1. **& (address of) Operator:** This operator gives the address of any variable.

   For example, if “max” is an integer variable, then &max will give memory address of variable max.

2. *** (dereference) Operator:** This operator returns the value at any memory address. Thus, the argument to this operator must be a pointer. It is called as a dereference operator as it works in a opposite manner to the & operator.

   For example, if “ptr” is a pointer variable which stores the address of variable “a”, then *ptr will return the value located at the memory address pointed by “ptr”.

A pointer variable can be declared as follows:

Syntax: Data_type * pointer_variable_name;

We need to specify the data type followed by * symbol and finally the name of pointer variable terminated by a semicolon.

For example, int *ptr;

This declaration tells the compiler that “ptr” is a pointer variable of type integer.

A Pointer variable can be initialized as given below:

Syntax: pointer_variable_name=& variable_name;

For example, int a; //variable a is declared as a integer variable

int *ptr; //declare ptr as pointer variable

ptr=&a; //ptr is a pointer variable which stores address of variable a

We can combine declaration and initialisation in one step also:

Syntax: data_type *pointer_variable_name=&variable_name

For example, int *ptr= &a;

It means ptr is a pointer variable storing the address of variable a.

Note: pointer variable should always point to address of variable of same data type:

For example, char a;

int *ptr;

ptr=&a; //Invalid as a is char type and ptr is integer type.

Dereferencing a pointer

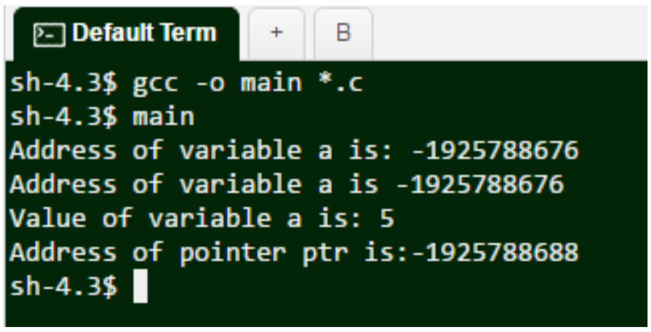
As it is already discussed, we can use * (dereference) operator to access the value stored at the address pointed by the pointer variable. This * operator is called as "value at operator" or "indirection operator".

For example, /*pointer variable declaration and initialization*/.

```
#include<stdio.h>
int main()
{
      int a=5;            //a is a integer variable
      int *ptr;           //ptr is a pointer variable declared
      ptr=&a;                 //pointer ptr stores the address of variable a
      printf("Address of variable a is: %d\n", &a);
            //prints the address of variable a
       printf("Address of variable a is %d\n", ptr);
                    //prints address of var a as it is stored in ptr
      printf("Value of variable a is: %d\n", *ptr);
                  //prints the value of variable a
      printf("Address of pointer ptr is:%d\n", &ptr);
```

```
                    //prints address of pointer variable ptr
    return 0;
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Address of variable a is: -1925788676
Address of variable a is -1925788676
Value of variable a is: 5
Address of pointer ptr is:-1925788688
sh-4.3$
```

Self-assessment Questions

1) Pointer is special kind of variable which is used to store __________ of the variable.

   a) Data Variable　　b) Variable Name

   c) Value　　d)Address

2) Pointer variable is declared using preceding _________ sign.

   a) *　　b) %

   c) &　　d) ^

3) Consider the 32 bit compiler. We need to store address of integer variable to integer pointer. What will be the size of integer pointer,

   a) 2 Bytes　　b) 6 Bytes

   c) 10 Bytes　　d) 4 Bytes

1.2.3 Accessing a variable through its pointer

Pointers are special kind of variables which can hold the address of another variable. Once a pointer has been assigned the address of any variable, we can use the value of that variable and manipulate it as per the requirement.

We know that the pointers can store the address of any variable using & (address of) operator. Once the address is stored in pointer, we can use a * (dereferencing/ indirection) operator followed by variable name to access the value of that variable.

For example,

```
int a, b;
a=60;
int *ptr;
ptr= &a;
b= *ptr;
```

Considering the above section of program, we declare 2 integer variables a and b.

We have also declared a integer pointer variable "ptr". In the next statement we assign the address of variable a to the pointer "ptr".

The statement

b=*ptr;

will assign the value at the address pointed by "ptr" to the variable "b". Thus variable "b" becomes same as variable "a".

This is equivalent to the statement

b= a;

The below figure 1.2.1 shows how a variable can be accessed using a pointer.

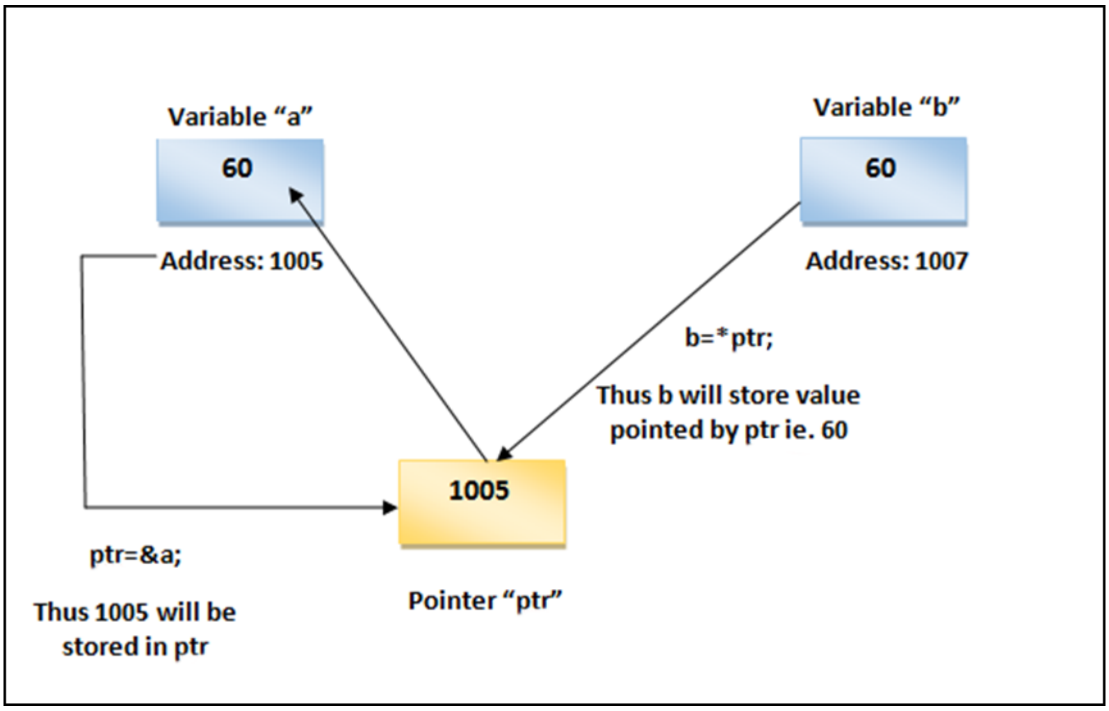
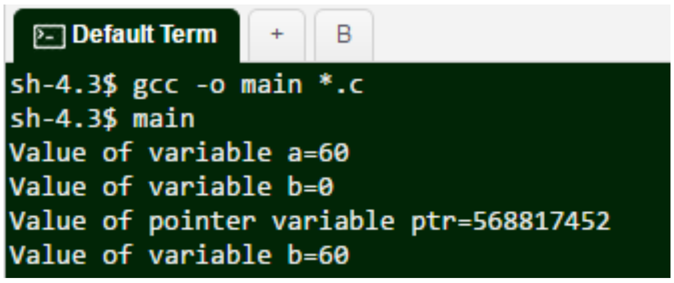


Figure 1.2.1: Accessing a variable through its pointer

Program:

```
/*Accessing a variable through pointer*/
#include <stdio.h>
int main()
{
    int a, b;
    a=60;
    b=0;
    int *ptr;
    ptr= &a;
    printf("Value of variable a=%d\n", a);
    printf("Value of variable b=%d\n", b);
    b=*ptr;       //assign value at address pointed by ptr to the
variable b
    printf("Value of pointer variable ptr=%d\n", ptr);
    printf("Value of variable b=%d\n", b);
    return 0;
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Value of variable a=60
Value of variable b=0
Value of pointer variable ptr=568817452
Value of variable b=60
```

Variable “a” is assigned a value 60. Pointer variable “ptr” will store the address of variable “a” say 1005. When we say “b=*ptr”, b will be assigned a value 60 pointed by the pointer “ptr”. Thus after execution of above program, a=60 and b will also have value 60.

Different Types of Pointer variables and their use:

In the following program, we have created 4 pointers: one integer pointer “iptr”, one float pointer “fptr”, one double type pointer “cptr” and one character type pointer “chptr”.

Each type of pointer variable stores the address of respective type of variable. For example, Character pointer variable will store the address of variable “ch” which is of type character.

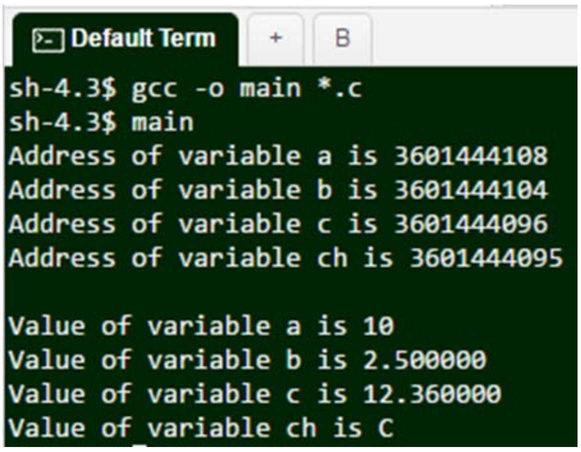
The indirection operator *i.e.,* * accesses an object of a specified data type at an address. Accessing any variable by its memory address is called indirect access. In the below given example *iptr indirectly accesses the variable that iptr points to *i.e.,* variable a.

Similarly pointer variable *fptr indirectly accesses the variable that fptr points to i.e. variable b.

Program:

```
/*different types of pointer variables */
#include <stdio.h>
int main()
{
     int a, *iptr;
     float b, *fptr;
     double c, *cptr;
     char ch, *chptr;
     iptr=&a;   //iptr stores address of integer variable a
     fptr=&b;   //fptr stores address of float variable b
     cptr=&c;   //cptr stores address of double variable c
     chptr=&ch; //chptr stores address of character variable ch
     a = 10;
     b = 2.5;
     c = 12.36;
     ch = 'C';
     printf("Address of variable a is %u \n", iptr);
     printf("Address of variable b is %u \n", fptr);
     printf("Address of variable c is %u \n", cptr);
     printf("Address of variable ch is %u \n\n", chptr);
     printf("Value of variable a is %d \n", *iptr);
     printf("Value of variable b is %f \n", *fptr);
     printf("Value of variable c is %f \n", *cptr);
     printf("Value of variable ch is %c \n", *chptr);
     return 0;
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Address of variable a is 3601444108
Address of variable b is 3601444104
Address of variable c is 3601444096
Address of variable ch is 3601444095

Value of variable a is 10
Value of variable b is 2.500000
Value of variable c is 12.360000
Value of variable ch is C
```

When the following pointer variable declarations are encountered, memory spaces are allocated for these variables at some addresses.

int a, *iptr;

float b, *fptr;

double c, *cptr;

char ch, *chptr;

The memory layout during declaration phase is shown in Figure 1.2.2.

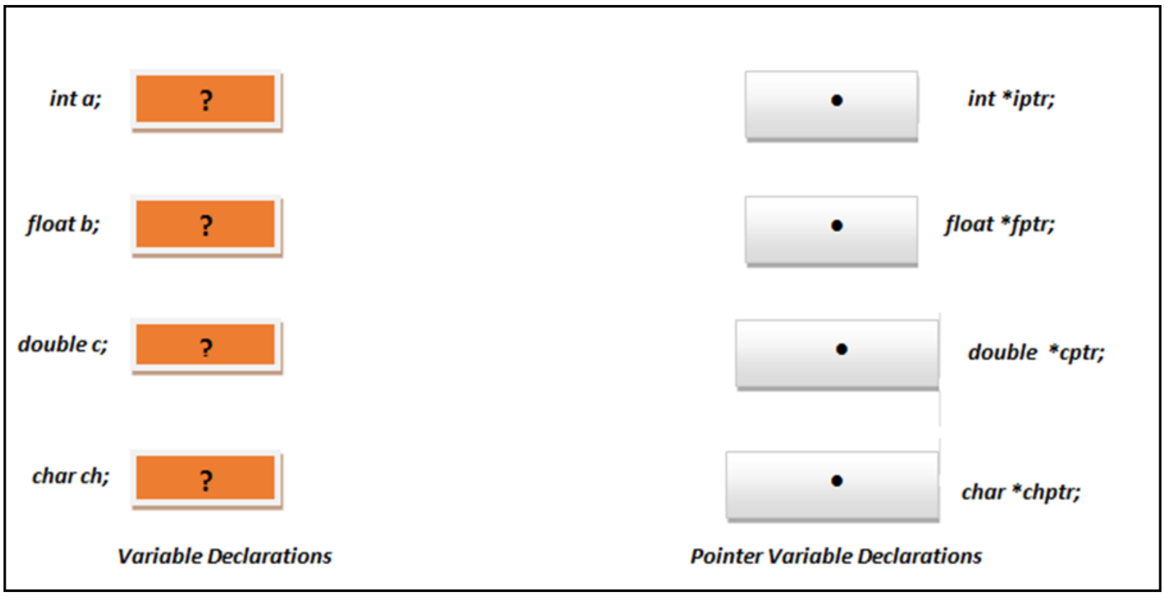


Figure 1.2.2: Declaration of Pointer variables

But when we assign the addresses of variables to the respective pointer variables, the memory layout will look the way shown in below figure 1.2.3

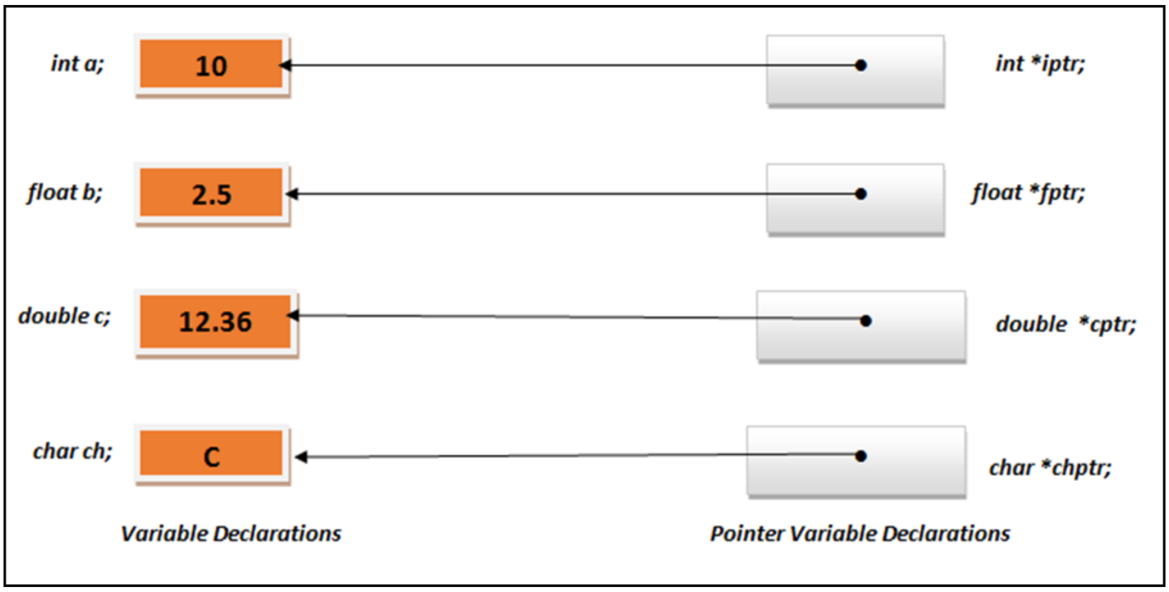


Figure 1.2.3: Effect of indirect Access and Assignments of Pointers

These initialized pointers may now be used to indirectly access the variables they are pointing.

Pointer arithmetic

As we perform arithmetic operations on regular integer variables, we can also perform arithmetic operations on pointer variables. Only addition and subtraction operations can be performed on pointer types.

But behaviour of addition and subtraction on pointer variables is slightly different. The operations behave differently according to the data type they are pointing to. The sizes of basic data types like integer, char, float etc. are already defined.

Suppose we define 3 pointer variables as given below:

char *cptr;

short *sptr;

long *lptr;

Let us assume that they point to memory locations 4000, 5000 and 6000 respectively.

If we write an increment statement as given below, it

will increment the address contained in it. Thus, its value will become 5001.

++cptr ;

This is because **cptr** is a character pointer and character is of 1 byte. Thus, incrementing a character pointer will add 1 to the memory address.

Similarly the statements,

++sptr ; will increment the address contained in sptr by 2 bytes as short is 2 bytes in size and

++lptr ; will increment the address contained in lptr by 4 bytes as long is 4 bytes in size.

Thus, when we increment a pointer, the pointer is made to point to the following element of the same type. Hence, the size in bytes of the type it points to is added to the pointer after incrementing it.

Same rules will follow for addition as well as subtraction operation. The below given statements give the same result as that of increment operator.

cptr = cptr + 1;

sptr = sptr + 1;

lptr = lptr + 1;

These increment (++) and decrement (--) operators can be used as either prefix or postfix operator in any expression. So in case of pointers, these operators can be used in similar way but with slight difference.

In case of prefix operator, the value is incremented first and then the expression is evaluated. In case of postfix operator, the expression is evaluated first and then the value is incremented.

Same rules follow for incrementing and decrementing pointers. As per the operator precedence rules, postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (*). Thus, the following expression:

***ptr ++;**

is same as *(ptr++). As ++ operator is used as postfix, the whole expression is evaluated as the value pointed originally by the pointer is then incremented.

There are the four possible combinations of the dereference operator with both the prefix and suffix increment operators.

1. *ptr++ //equivalent to *(ptr++)

   //increment pointer ptr and dereference unincremented address
2. *++ptr //equivalent to *(++ptr)

   //increment pointer ptr and dereference incremented address
3. ++*ptr //equivalent to ++(*ptr)

   //dereference pointer and increment the value stored in it
4. (*ptr)++ //equivalent to (*ptr)++

   //dereference pointer and post-increment the value stored in it

If we consider the following statement,

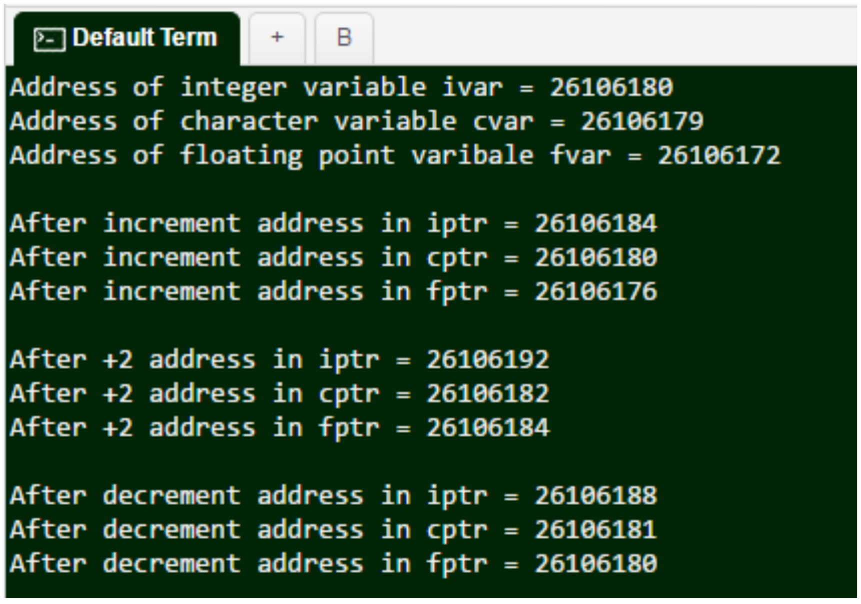
***ptr++ = *qtr++;**

As ++ has a higher precedence than *, both ptr and qtr pointers are incremented. Because both increment operators (++) are used as postfix operators, thus incrementing the value stored at address pointed by pointer ptr and qtr.

Program:

```
/*Pointer Arithmetic*/
#include<stdio.h>

int main()
{
     int ivar = 5, *iptr;
     char cvar = 'C', *cptr;
     float fval = 4.45, *fptr;

     iptr = &ivar;
     cptr = &cvar;
     fptr = &fval;

     printf("Address of integer variable ivar = %u\n", iptr);
     printf("Address of character variable cvar = %u\n", cptr);
     printf("Address of floating point varibale fvar = %u\n\n", fptr);

     /* Increment*/
     iptr++;
     cptr++;
     fptr++;
     printf("After increment address in iptr = %u\n", iptr);
     printf("After increment address in cptr = %u\n", cptr);
     printf("After increment address in fptr = %u\n\n", fptr);

     /* increment by 2*/
     iptr = iptr + 2;
     cptr = cptr + 2;
     fptr = fptr + 2;
     printf("After +2 address in iptr = %u\n", iptr);
     printf("After +2 address in cptr = %u\n", cptr);
     printf("After +2 address in fptr = %u\n\n", fptr);

      /* Decrement*/
     iptr--;
     cptr--;
     fptr--;
     printf("After decrement address in iptr = %u\n", iptr);
     printf("After decrement address in cptr = %u\n", cptr);
     printf("After decrement address in fptr = %u\n\n", fptr);
     return 0;
}
```

Output:



```
Address of integer variable ivar = 26106180
Address of character variable cvar = 26106179
Address of floating point varibale fvar = 26106172

After increment address in iptr = 26106184
After increment address in cptr = 26106180
After increment address in fptr = 26106176

After +2 address in iptr = 26106192
After +2 address in cptr = 26106182
After +2 address in fptr = 26106184

After decrement address in iptr = 26106188
After decrement address in cptr = 26106181
After decrement address in fptr = 26106180
```

Self-assessment Questions

4) Comment on following pointer declarations int *ptr, p;.

   a) ptr is a pointer to integers , p Is not

   b) ptr and p both are pointers to integer

   c) ptr is pointer to integer, p may or may not be

   d) ptr and p both are not pointers to integer

5) What will be the output?

```
main()
{
    char *p;
     p = "Hello";
     printf("%c\n",*&*p);
}
```

   a) Hello

   b) H

   c) 1005 (memory address of variable p)

   d) 1008(memory address of character H)

6) The statement int **a;,

   a) Is illegal b) Is legal but meaningless

   c) Is syntactically and semantically correct d) Stacks

7) Comment on the following,

   const int *ptr;

   a) We cannot change the value pointed by ptr.

   b) We cannot change the pointer ptr itself.

   c) Is illegal

   d) We can change the pointer as well as the value pointed by it

1.2.4 Memory allocation functions

Memory is a resource of computer system and it needs to be allocated properly for any kind of data structures used in programs. Dynamic memory allocation is a process of allocating memory to the data during program execution.

Normally when we are dealing with simple arrays or strings, we allocate the required amount of memory during compile time itself. We cannot extend the allocated memory during run-time. Hence, in such cases we need to allocate sufficient amount of memory at the compile time. But in compile time memory management, sometimes the allocated memory may not be used hence wasting the memory space.

Thus, we can make use of Dynamic memory allocation technique to allocate and de-allocate memory at runtime. Dynamic memory allocation helps us to increase or decrease the memory when the program is under execution.

The following are the dynamic memory allocation functions in C:

1. **malloc ()**

   It Allocates requested size of bytes and returns a pointer of first byte of allocated space.

2. **calloc()**

   It Allocates space for an array elements, initializes to zero and then returns a pointer to memory

3. **realloc()**

   It deallocate the previously allocated space.

4. **free()**

   We change the size of previously allocated space.

(i) malloc()

malloc, as the name indicates, stands for memory allocation. This function reserves a block of memory of specified size to return a pointer of type void.

Syntax of malloc()

```
ptr=(cast-type*)malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

(ii) calloc()

Calloc stands for "contiguous allocation". The difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of n elements.

For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, *i.e.,* 4 bytes.

(iii) free()

This function is used to explicitly free the memory allocated by malloc() and calloc() functions. It releases all the memory reserved for program.

```
free(ptr);
```

(iv) Realloc()

Sometimes a programmer requires extra memory or allocated memory becomes more than sufficient. In these cases, a programmer can change memory size previously allocated using realloc().

Syntax of realloc()

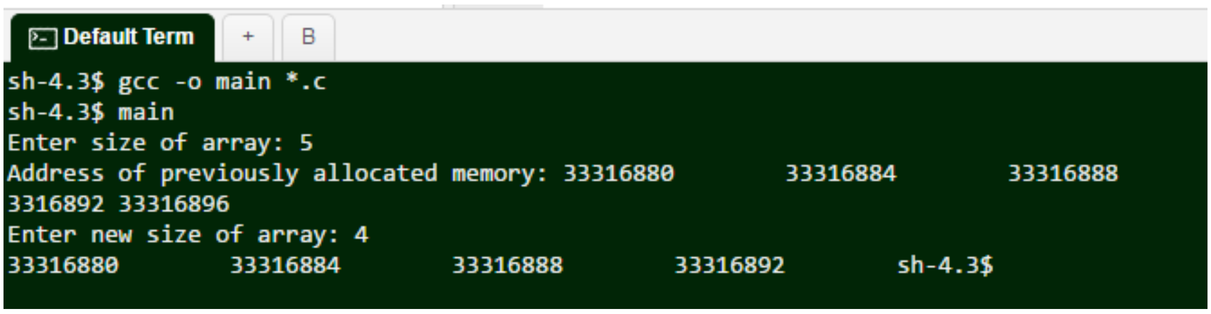
```
ptr=realloc(ptr,newsize);
```

Here, ptr is reallocated with size of newsize.

For example:

```
 #include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
         printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    for(i=0;i<n2;++i)
         printf("%u\t",ptr+i);
    return 0;
}
```

Output:



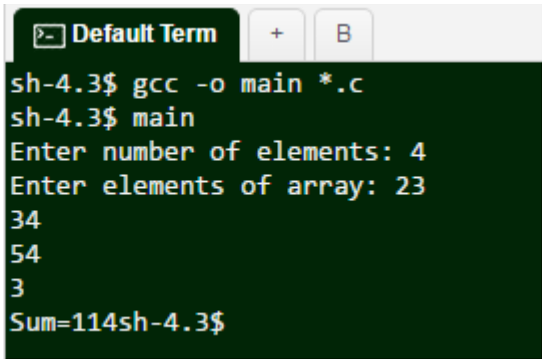
```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter size of array: 5
Address of previously allocated memory: 33316880        33316884        33316888
3316892 33316896
Enter new size of array: 4
33316880        33316884        33316888        33316892        sh-4.3$
```

Example showing use of malloc(), calloc() and free()

Program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
        if(ptr==NULL)
        {
            printf("Error! memory not allocated.");
              exit(0);
        }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter number of elements: 4
Enter elements of array: 23
34
54
3
Sum=114sh-4.3$
```

Self-assessment Questions

8) What function should be used to free the memory allocated by calloc()?

a) dealloc();	b) malloc(variable_name, 0)
c) free();	d) memalloc(variable_name, 0)

9) Which header file should be included to use functions like malloc() and calloc()?

a) memory.h	b) stdlib.h
c) string.h	d) dos.h

10) How will you free the memory allocated by the following program?

```
#include<stdio.h>
#include<stdlib.h>
#define MAXROW 3
#define MAXCOL 4
int main()
{
     int **p, i, j;
    p = (int **) malloc(MAXROW * sizeof(int*));
    return 0;
}
```

a) The name of array	b) The data type of array
c) The first data from the set to be stored	d) the index set of the array

11) Specify the 2 library functions to dynamically allocate memory?

a) malloc() and memalloc()	b) alloc() and memalloc()
c) malloc() and calloc()	d) memalloc() and faralloc()

1.2.5 Recursion

Recursion is considered to be the most powerful tool in a programming language. But sometimes, Recursion is also considered as the most tricky and threatening concept to a lot of programmers. This is because of the uncertainty of conditions specified by user.

In short Something Referring to itself is called as a Recursive Definition

(i) Definition

Recursion can be defined as defining anything in terms of itself. It can be also defined as repeating items in a self-similar way.

In programming, if one function calls itself to accomplish some task then it is said to be a recursive function. Recursion concept is used in solving those problems where iterative multiple executions are involved.

Thus, to make any function execute repeatedly until we obtain the desired output, we can make use of Recursion.

Example of Recursion:

The best example in mathematics is the factorial function.

n! = 1.2.3………(n-1).n

If n=6, then factorial of 6 is calculated as,

6! = 6(5)(4)(3)(2)(1)= 720

Consider we are calculating the factorial of any given using a simple. If we have to calculate factorial of 6 then what remains is the calculation of 5!

In general we can say

n ! = n (n-1)!   (i.e., 6! = 6 (5!))

it means we need to execute same factorial code again and again which is nothing but Recursion.

Thus, the Recursive definition for factorial is:

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ n * f(n-1) & \text{otherwise} \end{cases}$$

The above Recursive function says that the factorial of any number n=0 is 1, else the factorial of any other number n is defined to be the product of that number n and the factorial of one less than that number .

For example, consider n=4

As n is not equal to 0, the first case will not satisfy.

Thus, applying second case we get

4! = 4(4-1)! = 4(3!)

To find 3! Again we have to apply the same definition.

4! = 4(3!)=4[(3)(2!)]

Now, we have to calculate 2! , which requires 1! , which requires 0!.

As 0! is 1 by definition , we reach the end of it. Now we have to substitute the calculated values one by one in reverse order.

4!=4(3!)= 4(3)(2!)=4(3)(2)(1!)= 4(3)(2)(1)(0!)= 4(3)(2)(1)(1)= 24

Thus, 4!= 24

From the above solution it is clear that the each time we need to calculate the factorial of a value one less than the original one. Thus we reach value 0 where we have to stop applying same function of factorial.

Any recursive definitions will have some properties. They are:

- There are one or more base cases for which recursions are not needed.
- All cycles of recursion stops at one of the base cases.

We should make sure that each recursion always occurs on a smaller version of the original problem.

In C Programming a recursive factorial function will look like:

```
int factorial(int n)
{
        if (n==0)                          //Base Case
                return 1;
        else
                return n*factorial (n-1);      //Recursive Case
}
```

The above program is for calculating factorial of any number n. First when we call this factorial function, it checks for the base case. It checks if value of n equals 0. If n equals 0, then by definition it returns 1.

Otherwise it means that the base case is not yet been satisfied. Hence, it returns the product of n and factorial of n-1.

Thus, it calls the factorial function once again to find factorial of n-1. Thus forming recursive calls until base case is met.

Figure 1.2.4 shows the series of recursive calls involved in the calculation of 5!. The values of n are stored on the way down the recursive chain and then used while returning from function calls.

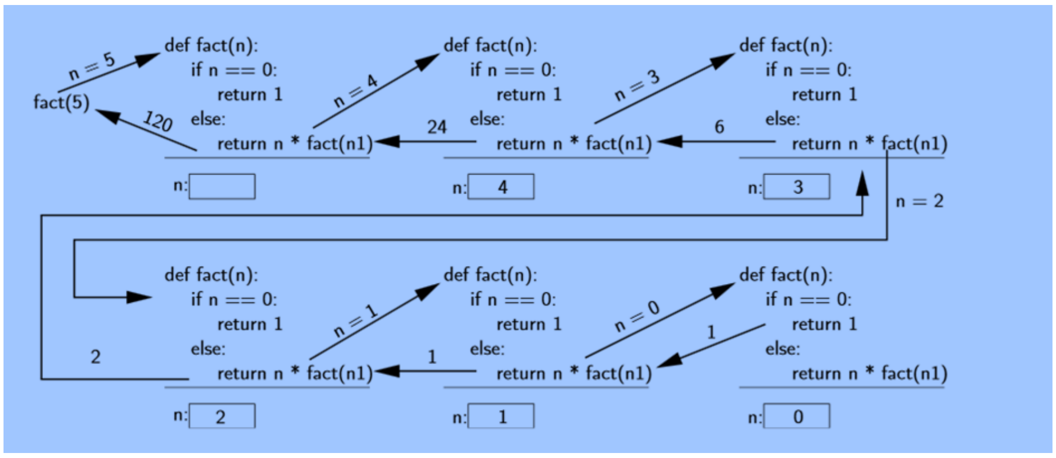


Figure 1.2.4: Recursive computation of 5!

(ii) Advantages

An important advantage of Recursion is that it saves time of programmer to a large extent. Even though problems like factorial, power or Fibonacci can be solved using loops but their recursive solutions are shorter and easier to understand. And there are algorithms that are quite easy to implement recursively but much more challenging to implement using loops.

Advantages of Recursion:

- Reduce unnecessary calling of function.
- Solving problems becomes easy when its iterative solution is very big and complex and cannot be implemented with loops.
- Extremely useful when applying the same solution

(iii) Recursive programs

1. Fibonacci series

One of the well-known problems is generating a Fibonacci series using Recursion.

A Fibonacci series looks like 0,1, 1, 2, 3, 5, 8, 13, 21and so on

Working: The next number is equal to sum of previous two numbers. The first two numbers of Fibonacci series are always 0 and. The third number becomes the sum of first 2 numbers, *i.e.,* 0 + 1 = 1. Similarly, the Fourth number is the sum of 3rd and 2nd number, *i.e.,* 1 + 1 = 3 and so on.

Thus, the Recursive definition for Fibonacci is:

$$F_k^{(n)} = \begin{cases} 0 & \text{if } 0 <= k < n-1 \\ 1 & \text{if } k = n-1 \\ \sum_{i=k-n}^{(k-1)} Fi^{n} & \text{Otherwise} \end{cases}$$

In C Programming a recursive fibonacci function will look like:

```
int fib(int n)
{
                if (n <= 1)
                              return n;
              else
return fib(n - 1) + fib(n - 2);
}
```

If n is less than or equal to 1, then return n. Otherwise return the sum of the previous two terms in the series by calling fib function twice. Once for (n-1) and next for fib (n-2).This combines results from 2 separate recursive calls. This is sometimes known as "deep" recursion. The below figure 1.2.5 demonstrates the working of recursive algorithm for Fibonacci series.

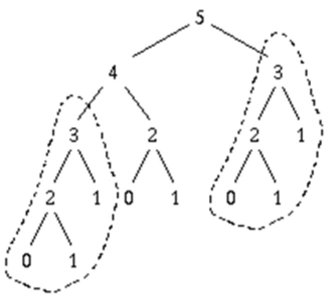


Figure 1.2.5: Recursive Algorithm

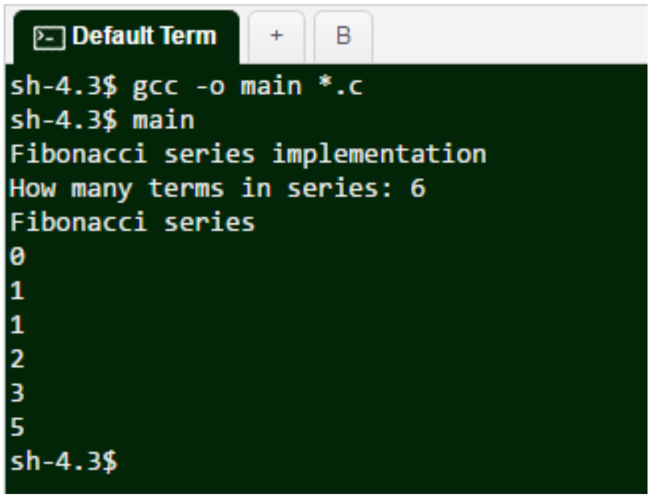
For example, the call to fib (4) repeats the calculation of fib (3) (see the circled regions of the tree). In general, when n increases by 1, we roughly double the work; that makes about 2n calls.

Following is the c program for implementation of Fibonacci series:

Program:

```
#include<stdio.h>
int fib(int n)
{
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( fib(n-1) + fib(n-2) );
}

int main()
{
   int n, j = 0, i;
   printf("Fibonacci series implementation\n");
   printf("How many terms in series: ");
   scanf("%d",&n);
   printf("Fibonacci series\n");
   for ( i = 1 ; i <= n ; i++ )
   {
      printf("%d\n",fib(j));
      j++;
   }
   return 0;
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Fibonacci series implementation
How many terms in series: 6
Fibonacci series
0
1
1
2
3
5
sh-4.3$
```

The above program uses the recursion concept to print the Fibonacci series. The program first asks the total number of terms to be displayed as output. Then it makes recursive calls to the function fib() and finds the next term in the series by adding previous two values in the series.

2. Binomial Coefficient

Binomial coefficient C (n, k) counts the number of ways to form an unordered collection of k items selected from a collection of n distinct items

For example, if you wanted to make a group of two from a group of four people, the number of ways to do this is C (4, 2).

Where, n=4 *i.e.,* 4 people and k=2 i.e. group of 2 people

There are total 6 ways to group them in an unordered manner.

Let us assume 4 people as A, B, C and D

So the 2 letter groups are: AB, AC, AD, BC, BD, and CD

Hence, C (n, k) = C (4, 2) = 6.

In general, Binomial Coefficients can be defined as:

- A binomial coefficient C (n, k) is the coefficient of X^k in the expansion of (1 + X)^n.
- A binomial coefficient C (n, k) also gives the number of ways, regardless of order, that k items can be chosen from among n items.

Problem:

This Problem of Binomial Coefficients can be implemented using Recursion. We need to write a function that takes two parameters n and k and returns the value of Binomial Coefficient C (n, k).

Recursive function:

The value of C(n, k) can recursively calculated using following standard formula for Binomial Coefficient's.

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

$$C(n, 0) = C(n, n) = 1$$

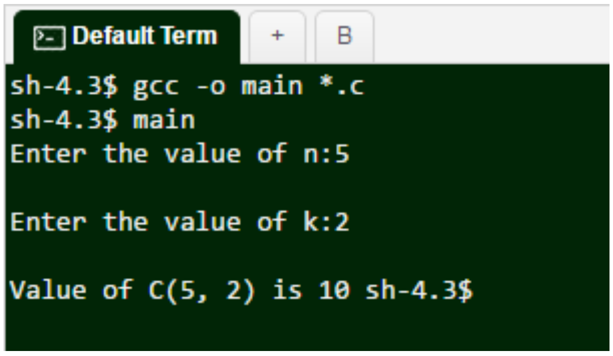
Below given program implements the calculation of Binomial Coefficients in a Recursive Manner.

Program:

//Recursive implementation of Binomial Coefficient C(n, k)

```c
#include<stdio.h>

int binomial(int n, int k)
{
    if (k==0 || k==n)     // Base Cases
        return 1;
    else
        return  binomial(n-1, k-1) + binomial(n-1, k);
}
int main()
{
    int n, k;
    printf("Enter the value of n:");
    scanf("%d",&n);
    printf("\nEnter the value of k:");
    scanf("%d",&k);
    printf("\nValue of C(%d, %d) is %d ", n, k, binomial(n, k));
    return 0;
}
```

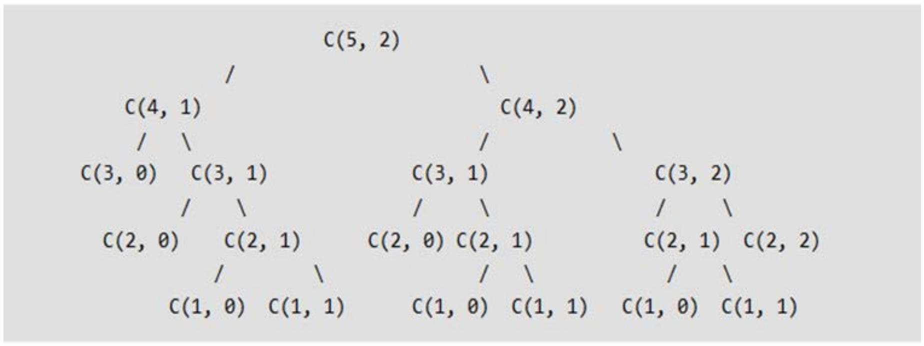
Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the value of n:5

Enter the value of k:2

Value of C(5, 2) is 10 sh-4.3$
```

It should be noted that in the above program, the binomial function is called again and again until the base cases are satisfied.

Below given figure 1.2.6 is the Recursive tree for n=5 and k=2.



```
                           C(5, 2)
                   /                       \
          C(4, 1)                           C(4, 2)
            /  \                           /          \
      C(3, 0)   C(3, 1)             C(3, 1)              C(3, 2)
                 /   \               /    \                /    \
         C(2, 0)     C(2, 1)     C(2, 0) C(2, 1)      C(2, 1)  C(2, 2)
                      /       \               /  \         /   \
               C(1, 0)  C(1, 1)     C(1, 0)  C(1, 1)  C(1, 0)  C(1, 1)
```

Figure 1.2.6: Example of DP and Recursion

3. GCD (Greatest Common Divisor)

The Greatest Common divisor of two or more integers is the largest positive integer that divides the numbers without a remainder.

For example, the GCD of 8 and 12 is 4.

Problem Definition: Given any nonnegative integers a and b, considering both are not equal to 0, calculate gcd(a, b).

Recursive Definition:

$$\text{For } a,b \geq 0,\ \gcd(a,b) = \begin{cases} a & \text{if } b=0 \\ \mathbf{gcd(b, (a \bmod b))} & \textbf{otherwise} \end{cases}$$

Input: Any Nonnegative integers a and b, both not equal to zero.

Output: The greatest common divisor of a and b.

For example:

Consider a=54 and b=24. We need to find GCD (54, 24)

Thus, the divisors of 54 are: 1, 2, 3, 6, 9, 18, 27, and 54

Similarly, the divisors of 24 are: 1, 2, 3, 4, 6, 8, 12, and 24

Thus, 1,2,3,6 are the common divisors of both 54 and 24:

The greatest number of these common divisors is 6.

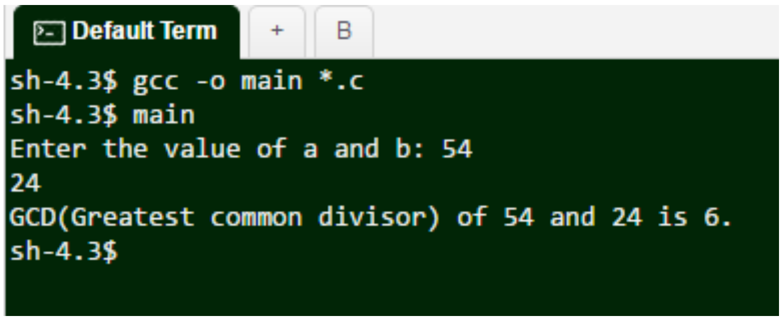
That is, the GCD (greatest common divisor) of 54 and 24 is 6.

The following program demonstrates computation of GCD using recursion:

Program:

```
/*GCD of Numbers using Recursion*/
#include <stdio.h>
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b)
            return gcd(a - b, b);
        else
            return gcd(a, b - a);
    }
    return a;
}

int main()
{
    int a, b, ans;
    printf("Enter the value of a and b: ");
    scanf("%d%d", &a, &b);
    ans = gcd(a, b);
    printf("GCD(Greatest  common  divisor)  of  %d  and  %d  is  %d.\n",  a,  b,
ans);
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the value of a and b: 54
24
GCD(Greatest common divisor) of 54 and 24 is 6.
sh-4.3$
```

Did you Know?

One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

Self-assessment Questions

12) Which Data Structure is used to perform Recursion?

a) Queue	b) Stack
c) Linked List	d) Tree

13) What is the output of the following code?

```
int doSomething(int a, int b)
{ if (b==1)
return a;
else
return a + doSomething(a,b-1);
}

doSomething(2,3);
```

a) 4	b) 2
c) 3	d) 6

14) Determine output of,

```
int rec(int num){
   return (num) ? num%10 + rec(num/10):0;
  }
       main(){
 printf("%d",rec(4567));
   }
```

a) 4 b) 12

c) 22 d) 21

15) What will be the below code output?

```
int something(int number)
  {
    if(number <= 0)
        return 1;
  else
        return number * something(number-1);
   }
   something(4);
```

a) 12 b) 24

c) 1 d) 0

16) void print(int n),

```
{
  if (n == 0)
    return;
  printf("%d", n%2);
    print(n/2);
}
```

What will be the output of print(12)?

a) 0011 b) 1100

c) 1001 d) 1000

Summary

- A pointer is a value that designates the address (i.e., the location in memory), of some value. Pointers are variables that hold a memory location.
- '&' - address of variable is used to assign address of any variable to pointer variable.
- '*' indirection operator is used to access the value contained in a particular pointer.
- Pointers store the address of any variable using & operator. We can access the value of that variable using a * operator succeeded by the variable name.
- **Memory allocation function:**
  - Malloc() - Allocates requested size of bytes and returns a pointer first byte of allocated space
  - Calloc() - Allocates space for an array elements, initializes to zero and then returns a pointer to memory
  - Free() - deallocate the previously allocated space
  - Realloc() - Change the size of previously allocated space
- Recursion is the process of repeating items in a self-similar way. In Programs, if a function makes a call to itself then it is called a recursive function. Recursion is more general than iteration. Choosing between recursion and looping involves the considerations of efficiency and elegance.

Terminal Questions

1. Explain the role of pointers in data structures.
2. What are the memory allocation functions? Explain in detail.
3. Define Recursive functions.
4. Write a note on indirection operator.

Answer Keys

Self-assessment Questions	
Question No.	Answer
1	d
2	a
3	a
4	a
5	b
6	c
7	a
8	c
9	b
10	d
11	c
12	b
13	d
14	c
15	b
16	a

Activity

1. Activity Type: Offline **Duration: 10 Minutes**

Description:

Ask all the students to get the output of the below question:

```
#include<stdio.h>
int main(){
   int i = 3;
   int *j;
   int **k;
   j=&i;
   k=&j;
   printf("%u %u %d ",k,*k,**k);
   return 0;
}
```

Prepare a presentation on pointers and dynamic memory allocation.

Bibliography

e-References

- cslibrary.stanford.edu, (2016). Stanford CS Education Library. Retrieved on 19 April 2016, from http://cslibrary.stanford.edu/106/
- doc.ic.ac.uk, (2016). Recursion. Retrieved on 19 April 2016, from http://www.doc.ic.ac.uk/~wjk/c++Intro/RobMillerL8.html

External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C' (2nd ed.).* Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth (2nd ed.).* BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C (2nd ed.).* Pearson Education

Video Links

Topic	Link
Introduction to pointers, declaring and initializing pointers and accessing variables through is pointers	https://www.youtube.com/watch?v=fAPt0Upy3ho
Memory allocation functions	https://www.youtube.com/watch?v=s4io0ir2kas
Recursion	https://www.youtube.com/watch?v=AuTjrMu-2F0

Notes: