Programming For Problem Solving

GROUP OF INSTITUTIONS

Lecture 40

POINTER

C Pointer is a special variable that stores/points the address of another variable. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Syntax of declaration of pointer:

data_type *var_name;

Example : int *p; char *p;

Where, * is used to denote that “p” is pointer variable and not a normal variable.

Declaration and initialization of pointer :

int a,*p=&a;

Dereferencing a pointer: When indirection operator (*) is used with the pointer variable, then it is known as **dereferencing a pointer.** When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Note: The size of any pointer is compiler dependent. (2 byte for 16 bit compiler) .

POINTER (CONT…)

Double Pointer: When we define a pointer to pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why second pointer is called double pointers.

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

Syntax:

```
int **ptr;
```

POINTER (CONT…)

```
// C program to demonstrate pointer and Double pointer
#include <stdio.h>
 int main()
{
    int var = 25,*ptr1,**ptr2;
     ptr1 = &var;
    ptr2 = &ptr1;
    printf("Value of var = %d\n", var );
printf("Address of var using single pointer = %u\n",ptr1 );
printf("Value of var using single pointer = %d\n",*ptr1 );
printf("Address of ptr1 using double pointer = %u\n",*ptr2 );
printf("Value of var using double pointer = %d\n", **ptr2);
  return 0;
}
```

Suppose:

Variable	var	ptr1	ptr2
Value	25	1000	2000
Address	1000	2000	5000

Output:

```
Value of var = 25
Address of var using single pointer = 1000
Value of var using single pointer = 25
Address of ptr1 using double pointer =2000
Value of var using double pointer = 25
```

TYPE OF POINTER

There are different types of pointers which are as follows −

- Null pointer
- Void pointer
- Wild pointer
- Dangling pointer

Null Pointer:

You create a null pointer by assigning the null value at the time of pointer declaration.

This method is useful when you do not assign any address to the pointer. A null pointer always contains value 0.

Example:

```
int *ptr=NULL;
printf("Value inside ptr = %d",ptr);
Output: Value inside ptr = 0
```

TYPE OF POINTER (CONT…)

void Pointer: It is a pointer that has no associated data type with it. A void pointer can hold addresses of any type and can be typecast to any type.

It is also called a generic pointer and does not have any standard data type.

It is created by using the keyword void.

Pointer arithmetic is not possible of void pointer due to its concrete size.

It can’t be used as dereference.

Example: void *p;

Wild Pointer:

Wild pointers are also called uninitialized pointers. Because they point to some arbitrary memory location and may cause a program to crash or behave badly.

Example

```
int *p; //wild pointer because p have not assigned any address
```

Dangling pointer:

Dangling pointer is a pointer pointing to a memory location that has been freed (or deleted).

TYPE OF POINTER (CONT…)

Different ways where Pointer acts as dangling pointer :

1.De-allocation of memory

```
float *p = (float *)malloc(sizeof(float));

free(p);   //now p is a dangling pointer.
```

2.Variable goes out of scope

```
{
    int *p ;
            {
                 int c;
                p=&c;
               }
     //p is dangling pointer here.
 }
```

POINTER ARITHMETIC

The operations that are allowed in pointer:

Increment and decrement

The Rule to increment the pointer is given below:

new_address= current_address + size_of(data type)

The Rule to Decrement the pointer is given below:

new_address= current_address - size_of(data type)

Addition of integer to a pointer

new_address= current_address + number * size_of(data type))

Subtraction of integer to a pointer

new_address= current_address - (number * size_of(data type))

Subtracting two pointers of the same type

Address2 - Address1 = (Subtraction of two addresses)/size of data type

Comparison of pointer

All relational operators can be used for pointer comparison of same type.

FEATURES OF POINTER

- Pointers save memory space.
- Execution time with pointers is faster because data are manipulated with the address, that is, direct access to memory location.
- Pointers can return multiple value
- An array, of any type can be accessed with the help of pointers, without considering its subscript range.
- Pointers are used for file handling.
- Pointers are used to allocate memory dynamically.
- Pointers are used to pass argument by reference

Programming For Problem Solving

Lecture-41

MEMORY ALLOCATION IN C

There are two types of memory allocation in C:

Static memory allocation (Compile time) and Dynamic memory allocation (Run time).

Static Allocation: In static memory allocation, the required amount of memory is allocated to the program in the start. Hence the memory allocated to variable is fixed and is determined by the compiler at the compile time e.g. :

▪ int a;	Here 2 bytes are allocated at compile time
▪ int a[5];	Here10 bytes are allocated at compile time

Dynamic memory allocation: Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

There are 4 library functions under "stdlib.h" for dynamic memory allocation.

malloc()

calloc()

realloc()

free()

DYNAMIC MEMORY ALLOCATION(DMA)

malloc(): The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

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.

DMA (CONT…)

calloc(): The name calloc stands for "contiguous allocation". The only 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.

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.

DMA (CONT…)

free(): Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free():

free(ptr);

This statement cause the space in memory pointer by ptr to be deallocated.

realloc():

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc():

ptr=realloc(ptr,newsize);

Here, ptr is reallocated with size of newsize.

DMA(CONT…)

malloc() V/S calloc()

malloc()	calloc()
The name malloc stands for memory allocation.	The name calloc stands for contiguous allocation.
malloc() takes one argument that is, number of bytes.	calloc() take two arguments those are: number of blocks and size of each block.

syntax of malloc():
void *malloc(size_t n);
Allocates n bytes of memory. If the allocation succeeds, a void pointer to the allocated memory is returned. Otherwise NULL is returned.
	syntax of calloc():
	void *calloc(size_t n, size_t size);
	Allocates a contiguous block of memory large enough to hold n elements of size bytes each. The allocated region is initialized to zero.

malloc is faster than calloc.	calloc takes little longer than malloc because of the extra step of initializing the allocated memory by zero.

DMA(CONT…)

STATIC V/S DYNAMIC MEMORY ALLOCATION

Static memory allocation	Dynamic memory allocation
Static Memory Allocation is done before program execution (compile time).	Dynamic Memory Allocation is done during program execution. (run time)
It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory
It is less efficient (less or more memory)	It is more efficient
Once the memory is allocated, the memory size cannot change.	Memory size can be changed.
we cannot reuse the unused memory.	The user can allocate more memory when required. Also, the user can release the memory when the user needs it.

DMA (CONT…)

// sum of n elements using dynamic memory allocation.

```
#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));
      if(ptr==NULL)
 {
    printf("Error! memory not allocated.");
     exit(0);
   }
```

DMA (CONT…)

```
 printf("Enter elements of array: ");
   for(i=0;i<n;++i)
  {
 scanf("%d",ptr+i);
    sum = sum + *(ptr+i);
   }
  printf("Sum=%d",sum);
  free(ptr);
}
```

DMA (CONT…)

```
// sorting of n integers using pointer.
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p,n,i,j,t;
    printf("enter how many numbers");
    scanf("%d",&n);
    p=(int *)malloc(n*sizeof(int));
    if(p==NULL)
    printf("memory not available");
    else
    {
      for(i=0;i<n;i++)
      {
          printf("enter value");
          scanf("%d",p+i);
      }
```

DMA (CONT…)

```
for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-1-i;j++)
        {
            if(*(p+j)>*(p+j+1))
            {
                t=*(p+j);
                *(p+j)=*(p+j+1);
                *(p+j+1)=t;
            }
        }
    }
    for(i=0;i<n;i++)
    {
        printf("%d  ",*(p+i));
    }
  }
  return 0;
}
```

Programming For Problem Solving

Lecture-42

SELF REFERENCE STRUCTURE

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

In other words, structures pointing to the same type of structures are self-referential in nature.

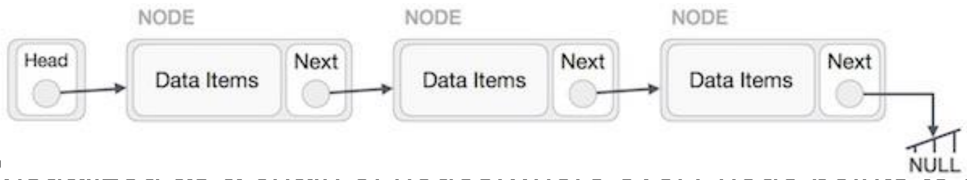
Example:

```
struct node
 {
    int data1;
    char data2;
    struct node* link;
};
```

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

LINKED LIST

A linked list is a list of elements, which are connected together via links.



Linked list can be visualized as a chain of nodes, where every node points to the next node.

Linked List contains a head node have address of first node.

Each node carries a data field(s) and a link field called next.

Each node is linked with its next node using its next link.

Last node carries a link as null to mark the end of the list.

Code for node:

```
struct node
{
int info;
struct node *next;
};
```

ADVANTAGE & DISADVANTAGES OF LINKED LIST

Advantages	Disadvantages
It can grow or shrink during execution, Hence avoid memory wastage	In a Linked list traversal is more time-consuming as compared to an array. Direct access to an element is not possible in a linked list. For example, for accessing a node at position n, one has to traverse all the nodes before it.
Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element only the address present in the next pointer needs to be updated.	More memory is required in the linked list as compared to an array. Because in a linked list, a pointer is also required to store the address of the next element
Linear data structures like stack and queues are often easily implemented using a linked list.	In a singly linked list reverse traversing is not possible, but in the case of a doubly-linked list, it can be possible as it contains a pointer to the previously connected nodes with each node. For performing this extra memory is required for the back pointer

Programming For Problem Solving

GROUP OF INSTITUTIONS

Lecture-43

CONCEPT OF FILE HANDLING

File: A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image.

FILE is inbuilt structure in c.

File Handling: File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

STEPS TO WORK ON FILE

Declaration of file pointer.

Opening of file.

Operation on file.

Closing of file.

Declaration Of File Pointer:

- FILE *f1, *f2;

Opening of file:

- File pointer = fopen(“filename”, “mode”);

STEPS TO WORK ON FILE (CONT...)

To check existence of file:

```
if(f1==NULL)
{
printf(“File doesn’t exist”);
exit(0);
}
```

Closing of file:

```
fclose(f1);
```

fclose() function is used to close a file.

FILE OPENING MODES

There are 12 main types of file opening mode:

"r"-Open file for reading and file must exist;

"w"- Open file for writing. If file does not exist it is created or if life already exist it's content is erased.

"a"-Open file for appending. It adds all information at the end of the file leaving old data untouched. If file does not exist it is created.

"r+"- Open file for reading and writing and file must exist.

"w+"- Open file for writing and reading. If file does not exist it is created or if life already exist it's content is erased.

"a+"- Open file for appending and reading. Again all new data is written at the end of the file old data leaving untouched. If file does not exist it is created.

As given above, if you want to perform operations on a binary file, then you have to append ‘b’ at the last. For example, instead of “w”, you have to use “wb”, instead of “a+” you have to use “a+b”.

TEXT FILES V/S BINARY FILES

There are three main differences between Text files and binary file from the programmers point of view:

Handling of newlines: In text mode a newline character is converted into the carriage return and line feed combination before being written to the disk. In binary mode, no such conversion takes place.

Representation of end of files: in Text mode a special character is inserted after the last character in the file to marks the end of file. As against this there is no such special character present in the binary mode. The binary file keeps track of the end of the file from the number of character present in the directory entry of the file.

Storage of numbers: Storage of numbers in text file is inefficient as numbers are stored as string of characters in Text mode.

FILE HANDLING LIBRARY FUNCTIONS

Function	Description	Syntax
fopen()	It is used to open the file	file pointer = fopen("file", "mode")
fclose()	It is used to close the file	fclose(file pointer)
fgetc() / getc()	It read single character from file	getc(file pointer) , fgetc(file pointer)
fputc() / putc()	It write a single character to file	putc(variable, file pointer), fputc(variable, file pointer)
getw()	It read single integer from file	getw(file pointer)
putw()	It write single integer to file	putw(variable, file pointer)
ftell()	It gives current location of pointer	ftell(file pointer)
fseek()	It set pointer to desired location	fseek(file pointer, n, refpos)
rewind()	It take the pointer to beginning of file	rewind(file pointer)
fscanf()	It read all type of content from file	fscanf(f, “%d”, &i)
fprintf()	It write all type of content to file	fprintf(f, “%d”, i)

fseek()

fseek() function is used to move file pointer position to the given location.

fseek(fp, long int offset, int whence)

Where

fp – file pointer

offset – Number of bytes/characters to be offset/moved from whence/the current file pointer position

whence – This is the current file pointer position from where offset is added. Below 3 constants are used to specify this.

SEEK_SET – It moves file pointer position to the beginning of the file.

SEEK_CUR – It moves file pointer position to given location.

SEEK_END – It moves file pointer position to the end of file.

Programming For Problem Solving

GROUP OF INSTITUTIONS

Lecture-44

FILE HANDLING PROGRAMS

/*WAP to input characters from keyboard & write on file*/

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
 char ch;
 FILE *f2;
 clrscr();
 f2 = fopen(“xyz.txt", “a");
```

FILE HANDLING PROGRAMS (CONT…)

```
if(f2 == NULL)
{
printf(“\nFile does not exist\n");
exit(0);
}
while((ch = getchar()) != EOF)
{
putc(ch, f2);
}
fclose(f2);
getch();
}
```

FILE HANDLING PROGRAMS (CONT…)

/*WAP to read characters from file*/

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
 char ch;
 FILE *f1;
 clrscr();
 f1 = fopen("abc.txt", "r");
```

FILE HANDLING PROGRAMS (CONT…)

```
if(f1 == NULL)
{
printf(“\nFile does not exist\n");
exit(0);
}
while((ch = getc(f1)) != EOF)
{
putchar(ch);
}
fclose(f1);
getch();
}
```

FILE HANDLING PROGRAMS (CONT…)

/* Write a c program to count the number of character in a file and copy the text of that file to another file. */

```
#include<process.h>
#include<stdio.h>
#include<conio.h>
void main()
{  char ch;
FILE *f1, *f2;
int n=0;
clrscr();
f1=fopen("abc.txt","r");
f2=fopen("xyz.txt","a");
```

FILE HANDLING PROGRAMS (CONT…)

```
if(f1==NULL||f2==NULL)
{  printf("file does not exist");
exit(0);
 }
while((ch=fgetc(f1))!=EOF)
{
n++;
fputc(ch,f2);
}
printf(“number of character in file = %d ”,n);
fclose(f1);
fclose(f2);
getch();
 }
```

FILE HANDLING PROGRAMS (CONT…)

/* WAP to read number from file and then write all ‘odd’ number to file ODD.txt & all even to file EVEN.txt */

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp,*fo,*fe;
int n;
clrscr();
 fp=fopen("DATA.TXT","r");
 fo=fopen("ODD.TXT","w");
 fe=fopen("EVEN.TXT","w");
```

FILE HANDLING PROGRAMS (CONT…)

```
 if(fp = = NULL ||  fo = = NULL || fe = =NULL)
{
printf("file can not open");
exit(0);
}
while( (n=getw(fp)) !=EOF )
{
 if(n%2==0)
putw(n,fe);
 else
 putw(n,fo);
}
fclose(fp);
fclose(fo);
fclose(fe);
 getch();
}
```

FILE HANDLING PROGRAMS (CONT…)

/* Suppose a file contains student's records with each record containing name and age of a student. WAP in C to read these records and display them in sorted order by name. */

```
#include <stdio.h>
#include <string.h>
struct student
{
int age;
char name[50];
};
int main()
{
FILE *p;
struct student s[50],t;
int i=0,n=0,j;
p=fopen("a.txt","r");
```

FILE HANDLING PROGRAMS (CONT…)

```
if(p==NULL)
{
printf("file cant open");
exit(0);
}
while(fscanf(p,"%d%s",&s[i].age,&s[i].name)!=EOF)
{
i++;
n++;
}
```

FILE HANDLING PROGRAMS (CONT…)

```
for(i=0;i<n-1;i++)
{
for(j=0;j<(n-1-i);j++)
{
    if(strcmp(s[j].name,s[j+1].name)>0)
     {
          t=s[j];
       s[j]=s[j+1];
        s[j+1]=t;
          }
}
}
for(i=0;i<n;i++)
{
printf("%d %s\n",s[i].age,s[i].name);
}
return 0;
}
```

Programming For Problem Solving

Lecture-45

PRE PROCESSOR

Before, the source code passes through the compiler .It is examined by the processor for any preprocessor directives. Preprocessor directives follow special syntax rules. They all begin with the symbol # and do not require a semicolon.

It also removes comment, white space and converts .c file into .i file and send to compiler.

These directives can be divided into 3 categories :

File Inclusion Directive

Compiler Control Directive(Conditional compilation)

Macro Substitution directive

FILE INCLUSION

It is used to include the content of one file into other program using #include pre-processor directive.

There are 2 ways to do it:

#include<stdio.h>:This method will search the file <stdio.h> in standard/specific list of directories only.

#include“stdio.h”: This method will search the file in standard as well as current directory. (first search in current directory if not find then search in standard)

CONDITIONAL COMPILATION

This method help programmer to write compatible code for two different machines.

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. The beginning of the block of program text is marked by one of three directives:

#if

#ifdef

#ifndef

Optionally, an alternative block of text can be set aside with one of two directives:

#else

#elif

The end of the block or alternative block is marked by the **#endif** directive

If the condition checked by #if , #ifdef , or #ifndef is true (nonzero), then all lines between the matching #else (or #elif) and an #endif directive, if present, are ignored.

If the condition is false (0), then the lines between the #if , #ifdef , or #ifndef directive are ignored.

CONDITIONA L COMPILATION (CONT…)

Example1:

```
#define CODE
void main()
{
printf(“a”);
#ifdef CODE
printf(“b”);
#else
printf(“c”);
#endif
printf(“d”);
}
```

Output: abd

CONDITIONA L COMPILATION (CONT…)

Example2:

```
#define CODE
void main()
{
printf(“a”);
#ifndef CODE
printf(“b”);
#else
printf(“c”);
#endif
printf(“d”);
}
```

Output: acd

CONDITIONAL COMPILATION (CONT…)

Example3:

```
#define CODE 10
void main()
{
printf(“a”);
#if CODE>20
      printf(“b”);
#else
                    #if CODE>15
                    printf(“c”);
                    #else
                    printf(“d”);
                    #endif
 #endif
printf(“e”);
}
```

Output: **ade**

CONDITIONAL COMPILATION (CONT…)

Example4:

```
#define CODE 10
void main()
{
printf(“a”);
#if CODE>20
      printf(“b”);
#elif CODE>15
                   printf(“c”);
  #else
                   printf(“d”);
 #endif
printf(“e”);
}
```

output: **ade**

Programming For Problem Solving

Lecture-46

MACRO

A macro is a fragment of code that is given a name. You can use that fragment of code in your program by using the name. Macro is defined by #define directive.

There are two types of macros:

Object-like Macros

Function-like Macros

Object-like Macros: The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

```
#define PI 3.14
```

Here, when we use PI in our program, it's replaced with 3.14.

Here PI is called macro template and 3.14 is macro expansion.

MACRO (CONT…)

Example simple Macro:

```
#include <stdio.h>
#define PI 3.1415
int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%d", &radius);
    // Notice, the use of PI
    area = PI*radius*radius;
    printf("Area=%.2f",area);
    return 0;
}
```

MACRO (CONT…)

Function like Macro(Macro with argument):

You can also define macros that works like a function call, known as function-like macros.

Example:

```
#define circleArea(r) (3.14*r*r)
```

Every time the program encounters circleArea(argument), it is replaced by (3.14*argument*argument).

Example :

```
#include <stdio.h>
#define circleArea(r) (3.14*r*r)
int main()
{
    float radius,area;
    printf("Enter the radius: ");
    scanf("%f", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);
    return 0;
}
```

MACRO V/S FUNCTION

Macro	Function
Macro is **Preprocessed**	Function is **Compiled**
No Type Checking	**Type Checking** is Done
Code Length **Increases**	**Code** Length remains **Same**
Speed of Execution is **Faster**	Speed of Execution is **Slower**
Before Compilation macro name is replaced by macro value	During function call , Transfer of Control takes place
Useful where small code appears many time	Useful where large code appears many time
Macro does not Check **Compile Errors**	Function Checks **Compile Errors**

#undef

It is used to undefined the macro which was previously defined by #define pre-processor directive.

```
#define CODE
#undef CODE
void main()
{
printf(“a”);
#ifdef CODE
printf(“b”);
#else
printf(“c”);
#endif
printf(“d”);
}
```

Output: acd

#pragma

#pragma is used to call a function before and after main function in a C program.

```
#include <stdio.h>
 void function1( );
void function2( );
 #pragma startup function1
#pragma exit function2
 int main( )
{
   printf ( "\n Now we are in main function" ) ;
   return 0;
}
```

#pragma (CONT…)

```
void function1( )

{

   printf("\nFunction1 is called before main function call");

}

 void function2( )

{

   printf ( "\nFunction2 is called just after end of main function" ) ;

}
```

Output:

```
Function1 is called before main function call

Now we are in main function

Function2 is called just after end of main function
```

Programming For Problem Solving

Lecture-47

COMMAND LINE ARGUMENT

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside.

Command line arguments are passed to the main() method.

Syntax:

```
int main(int argc, char *argv[])
{
}
```

- Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

COMMAND LINE ARGUMENT (CONT…)

/* WAP to print arguments passed by command line run as:

c:/tc/ test hello I am here */

```
#include <stdio.h>

int main(int argc, char *argv[])

{

      int i; //for loop counter

      for(i=0; i<argc; i++)

      {

       printf("%s\t",argv[i]);

      }

return 0;

}
```

Output:

```
test      hello     I      am       here
```

COMMAND LINE ARGUMENT (CONT…)

/*WAP to sum of n numbers using command line argument. run as

c:/tc/ test 4 5 7 3 6 */

```
#include <stdio.h>
int main(int argc, char *argv[])
{     int a,b,sum=0;
      int i; //for loop counter
      for(i=1; i<argc; i++)
      {
                sum = sum+atoi (argv[i]);
      }
      printf("SUM =: %d\n",sum);
      return 0;
}
```

Output:

```
SUM=25
```

COMMAND LINE ARGUMENT (CONT…)

/*WAP to find the factorial of number using command line argument*/

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
 int n, f=1, i;
clrscr();
n = atoi(argv[1]);
for (i=n; i>1; i=i-1)
f = f * i;
printf(" Factorial of %d=%d", n, f);
getch();
}
```

THANK YOU :)