Strings

Strings : strings are basically an array which is collections of characters.

<Ex>

```
#include<stdio.h>
main()
{
        char s[100]={'a','b','c','d','e'};
        int n,i;
        n=sizeof(s)/sizeof(s[0]);
        for(i=0;i<n;i++)
                printf("%c",s[i]);
        printf("\n");
}
```

now in above example, although we have supplied only 5 characters in string loop will execute 100 times.

Note that when an character array is partially initialized, remaining elements are made NULL by compiler. So in array of 100 characters in above example, compiler only initializes first 5 characters and make remaining elements NULL.

NULL ('\0') is an important character in strings. In all strings , NULL indicates end of string.

So above program can be minimized by using below code.

```
#include<stdio.h>
main()
{
        char s[100]={'a','b','c','d','e'};
        int i;
        for(i=0;s[i];i++)
                printf("%c",s[i]);
        printf("\n");
}
```

but in above program, printf() is still called 5 times. Loop terminates when NULL is found in string. But to print whole string with one printf() call, printf() with format specifier %s should be used. Print() with %s is used for printing strings on screen and stops printing only when NULL is found in string. And having NULL based operations, there is no need of passing number of elements to any function. Same way to printf() also only base address of string is supplied

```
printf("%s\n",s);   //prints all valid characters from location
                      represented by s up to NULL.
```

Characteristics of NULL character :

NULL character assignment,

```
char ch = 0;        // integer constant assignment
char ch = '\0';     // character constant assignment
```

here both assignments are same and indicated that NULL is assigned to variable ch because all characters are stored in memory by their ASCII values. And NULL character's ASCII value is 0.

<Ex>

```
#include<stdio.h>
main()
{
        char ch;
        ch=0;
        printf("%c",ch);
}

o/p > nothing gets printed on screen
```

NOTE : NULL can not be input from keyboard nor it can be printed on screen.

<Ex>

```
#include<stdio.h>
main()
{
        char s[10]={'a','b','c','d','e'};
        printf("%s\n",s);
        s[3]='\0';            //'a','b','c','NULL','e'
        printf("%s\n",s);
        s[1]=0;               //'a','NULL','c','NULL','e'
        printf("%s\n",s);
        printf("%s\n",s+1);   //points to s[1], which is NULL
        printf("%s\n",s+2);   //points to s[2] which is 'c'
        printf("%s\n",s+4);   //points to s[4] which is 'e'
}

o/p > abcde
      abc
      a

      c
      e
```

<Ex>

```
#include<stdio.h>
main()
{
```

```
        char s[10]={'a','b','c','d','e'};
        char str[10]={'s','=','%','s','\n'};
        printf(str,s);
        printf(str,"xyz");
}

o/p > s=abcde
      s=xyz
```

<Q> Even though printf's first argument is string, how come printf is able to print data correctly with its first argument as base address of array?

→ in GCC string constants are saved in text section, and represents base address of string present in text section. So when a string constant like “s=%s\n” is passed to printf, it is actually string stored in text section and in program, string constant is represented by its base address of text section. So actually printf()'s first argument is base address of string and not string itself. When base address of string present in text section is given, printf can dereference it and read string from that address and according to format specifiers present in string, prints data on console output. So thats why when we save percentage specifier's string in another array, printf() is able to print outputs correctly.

Conclusion from above discussion : printf() and scanf()'s first argument is address, not string.

We know that in memory, stack is higher and then data section and text section is present. But data section is present above text (code) section. So we can verify that string constants are stored in text section by below code.

<Ex>

```
#include<stdio.h>
main()
{
        int i;
        static int j;
        printf("i : %u\n",&i);
        printf("j : %u\n",&j);
        printf("string constant : %u\n","xyz");
}

o/p > i : 3215951964
      j : 134520868
      string constant : 134513936
```

variable i is local variable of main() which will be in stack, variable j is static variable which will be present in data section, which is much lower address than stack. String constant's address is lower than address of variable j. It is text section's address.

<Ex>

```
#include<stdio.h>
main()
{
        char s1[10]={'a','b','c','d','e'};
        char s2[10]="chintan";
        printf("s1 : %s\n",s1);
        printf("s2 : %s\n",s2);
}

o/p > s1 : abcde
      s2 : chintan
```

in above example, s1 and s2 both are of 10 characters. So when s1 is initialized with 5 characters, compiler fills NULL in remaining 5 characters, so printf() works perfectly. In case of s2, it is initialized by string constant. When an character array is initialized with string constant, whole string gets stored in array with NULL at end. So to initialize strings string constants should be used like in case of s2 instead of separate characters used to initialize s1.

Now consider following example,

<Ex>

```
#include<stdio.h>
main()
{
        char s1[5]={'a','b','c','d','e'};
        char s2[20]="chintan";
        char s3[10]="patel";
        printf("s1 : %s\n",s1);
        printf("s2 : %s\n",s2);
}

o/p > s1 : abcdepatel
      s2 : chintan
```

<Q> Why “patel” also got printed with “abcde” when only s1 string is printed using %s?

→ printf() with %s is a NULL based operation, when we pass string s1's base address to printf, it doesn't know how many characters are present in s1 or what is the size of s1. It only searches for a NULL in s1 and when found, it stops printing. But intentionally we have made string s1 of size 5 and initialized it with 5 characters so there will be no NULL present at the end of s1. In s2 and s3, NULL is guaranteed at lase because size of s2 and s3 is bigger than characters used to initialized it. So there is space for NULL at the end and as they are initialized with string constant, NULL is guaranteed at the end.

Now see base addresses of strings s1, s2 and s3 by using below statements,

```
printf("s1 is at %u\n",s1);
printf("s2 is at %u\n",s2);
printf("s3 is at %u\n",s3);

o/p > s1 is at 3215562521
      s2 is at 3215562536
      s3 is at 3215562526
```

from addresses it is clear that s1 s2 and s3 are present in contagious memory blocks and they are present in order, s1 --> s3 --> s2. Now arrays are present in which order is compiler's choice.

Now when s1 is tried to print with %s, it keeps on searching NULL, but as there is no NULL at last of s1, it also enters in array s3 as they are contagious in memory. So at the end of s3 there is NULL present. So printf() stops printing in encounter of NULL at the end of s3. And then s2 is printed as it is as NULL is present at the end of s2.

Conclusion : in such kind of strings, there should be always one extra space provided for NULL, otherwise unpredictable results may displayed when string is printed by using printf() with %s.

Now consider another example,

<Ex>

```
#include<stdio.h>
main()
{
        char s1[]={'a','b','c','d','e'};
        char s2[]="chintan";
        char *s3="patel";
        printf("%d %d %d\n",sizeof(s1),sizeof(s2),sizeof(s3));
        printf("s1 : %u &s1 : %u\n",s1,&s1);
        printf("s2 : %u &s2 : %u\n",s2,&s2);
        printf("s3 : %u &s3 : %u\n",s3,&s3);
}

o/p > 5 8 4
      s1 : 3216764975 &s1 : 3216764975
      s2 : 3216764980 &s2 : 3216764980
      s3 : 134514128 &s3 : 3216764968
```

→ if size of array is not provide then compiler automatically decides the size when array has been initialized. So in case of s1, no null will be provided at the end so its size is 5 bytes. In s2 null will be provided at its end so its size is 8 bytes. But s3 is pointer, and any pointer in c is of 4 bytes (in 32 bit GCC compiler, in 64 bit GCC it is of 8 bytes). So string will be stored in code section and its address will be supplied to s3.

→ s1, s2 and s3 will be in stack section. s1 and s2 will be containing characters but s3 will be containing text area's address.

Now in above code if we write,

```
s1[0]=s2[0];
s2[0]=s3[0];
printf("%s\n%s\n%s\n",s1,s2,s3);

o/p > cbcdephintan
      phintan
      patel
```

but if following code is written,

```
s1[0]=s2[0];
s2[0]=s3[0];
s3[0]=s1[0];
printf("%s\n%s\n%s\n",s1,s2,s3);

o/p > segmentation fault
```

its segmentation fault because s1 ad s2 are present in stack and its contents are modifiable by user. But although s3 is present in stack, string pointed by it is present in code section which is read only section. So any attempt to modify anything in read only section will result in segmentation fault.

Getting string inputs from keyboard :

<Ex>

```
#include<stdio.h>
main()
{
        char s[100];
        printf("Enter a string :");
        scanf("%s",&s);
        printf("You have entered : %s\n",s);
}

o/p > Enter a string :chintan patel
      You have entered : chintan
```

here expectation was to input “chintan patel” as whole string in s. But scanf() considers space as data separator by default. So anything separated by space will be considered two different data to scanf.

<Q> So what should be done to input strings with space?

→ there are two ways. We can make scanf to scan string up to new line is supplied. This can be achieved by following statement.

1.) `scanf(“%[^\n]s”,s);`

here `[^\n]` indicates that scanf keeps on accepting data until new line is supplied.

Similarly, `scanf("%[^.]s",s);`
here `[^.]` indicates that data will be accepted until `'.'` Is not supplied

2.) `gets(s);`

gets() function accepts one string up to new line by default.

When with scanf() and gets() string is entered and enter is hit, it puts entered string in given location but new line is not put, instead NULL is placed at the last.

Finding length of string & size of string :

<Ex>

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[100];
        printf("Enter a string :");
        gets(s);
        printf("size of string : %d\n",sizeof(s));
        printf("length of string : %d\n",strlen(s));
}

o/p > Enter a string :chintan
      size of string : 100
      length of string : 7
```

sizeof(s) – gives total size allocated in memory for s.
strlen(s) – gives total characters present in string s without NULL.

strlen() :

→ declaration :

```
size_t strlen(const char *s);
```

→ The strlen() function calculates the length of the string s, excluding the terminating null byte ('\0').

→ strlen() user defined :

```
size_t Ustrlen(const char *p)
{
        size_t count=0;
        int i;
        for(i=0;p[i];i++)
                count++;
        return count;
}
```

<Ex>

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[10];
        printf("Enter a string :");
        gets(s);
        printf("Entered string : %s\n",s);
}

o/p > Enter a string :hello friends, how are you?
      Entered string : hello friends, how are you?
      *** stack smashing detected ***: ./a.out terminated
      Aborted (core dumped)
```

if like shown above, if we enter data more than size of the string, then scanf() of gets() doesn't report error about it because size of array is now known to them. Only argument supplied to both function is starting address of string from where scanf() or gets() will start storing entered data.

If we supply more data than size of string, then scanf() or gets() will try to store it beyond allocated memory of string where it is not allowed to do so, so a stack smashing will be detected by OS and it will report the error and abort the application execution.

<Ex>

```
#include<stdio.h>
main()
{
        char str[10]="chintan";
        char *p="patel";
        str=p;
        printf("%s\n",str);
}

o/p > test.c: In function ‘main’:
      test.c:6:5: error: incompatible types when assigning to type
      ‘char[10]’ from type ‘char *’
        str=p;
           ^
```

→ if in above program, in place of str=p; , p=str; would had been written then it is completely valid statement, its because p is a pointer, it is initialized by the base address of string present in text section. But later base address of string str is assigned to it. Although we loose track of string present in text section and will never able to use it again in our program, but it is syntactically valid.

<Assignment> WAP to declare an array of 100 characters and input a string and do following
(1). input a character and search number of occurrences.

(2). reverse a string and print it.
(3). convert all lower case to upper case.

1) input a char and find no. Of occurrences

```
#include<stdio.h>
main()
{
        char str[100],ch;
        int i,cnt=0;;
        printf("Enter a string : ");
        gets(str);
        printf("Input a character to search : ");
        scanf("%c",&ch);
        for(i=0;str[i];i++)
        {
                if(str[i]==ch)
                        cnt++;
        }
        printf("character %c occurs %d times\n",ch,cnt);
}
```

```
o/p > Enter a string : chintan patel
      Input a character to search : a
      character a occurs 2 times
```

2) reverse string and print it

```
#include<stdio.h>
main()
{
        char str[100],ch;
        int i,cnt=0;
        printf("Enter a string : ");
        gets(str);
        for(i=0;str[i];i++);       //find length of string
        cnt=i;
        printf("%d\n",cnt);
        for(i=0;i<=cnt/2;i++)
        {
                ch=str[i];
                str[i]=str[cnt-i-1];
                str[cnt-i-1]=ch;
        }
        printf("reversed string : %s\n",str);
}
```

3) convert all lower case to upper case

```
#include<stdio.h>
main()
{
        char str[100];
```

```
        int i;
        printf("Enter a string : ");
        gets(str);
        for(i=0;str[i];i++)
        {
                if((str[i]>='a')&&(str[i]<='z'))
                        str[i]-=32;
        }
        printf("string : %s\n",str);
}

o/p > Enter a string : ChInTan PatEl
      string : CHINTAN PATEL
```

String Copy :

for copying one string into another, there are two ways.
1). copying member by member of one string into another
2). by using library function strcpy()

1) member by member transfer

```
#include<stdio.h>
main()
{
        char s1[100],s2[100];
        int i;
        printf("Enter string for s1 : ");
        gets(s1);
        for(i=0;s1[i];i++)
                s2[i]=s1[i];
        s2[i]=0;
        printf("s2 : %s\n",s2);
}
```

2) using built in function strcpy()

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100];
        printf("Enter string s1 : ");
        gets(s1);
        strcpy(s2,s1);
        printf("s2 : %s\n",s2);
}
```

Function strcpy() declaration is present in string.h header file. So in order to use it string.h header file should be included in our program.

→ `strcpy(s2,s1);` // copies all character from s1 into s2 including the null character

strcpy() :

→ declaration :

```
char *strcpy(char *dest, const char *src);
```

<Ex>

```
strcpy(s1,"abcdefgh");   // s1 : abcdefgh
strcpy(s2,s1+3);         // s2 : defgh
strcpy(s1,s2+1);         // s1 : efgh
```

<Ex>

```
strcpy(s1,"abc");              // s1 : abc
strcpy(s2,"01234");            // s2 : 01234
strcpy(s1+strlen(s1),s2);      // s1 : abc01234
strcpy(s2+strlen(s2),s1);      // s2 : 01234abc01234
```

<Ex>

```
strcpy(s1,"abcdefgh");   // s1 : abcdefgh\0
strcpy(s1,s1+2);         // s1 : cdefgh\0h\0
strcpy(s1+1,s1+2);       // s1 : cefgh\0\0h\0
printf("s1 : %s\n",s1);  // s1 : cefgh
```

<Ex>

```
strcpy(s1,"abcdefgh");
strcpy(s1+4,s1+3);
```

→ in this kind of operation, unpredicted behavior may be seen, strcpy() stops copying elements only after it copies null from source. But in above example, null is not copied from source to destination. So it doesn't stop until it reaches out of the array size and results in segmentation fault.

→ return : return a pointer to the destination string

→ strcpy() should not be used when source and destination address are present in overlapping memory area. e.g. are in same string.

→ strcpy() user defined :

```
char *Ustrcpy(char *dest,const char *src)
{
        int i;
        for(i=0;src[i];i++)
                dest[i]=src[i];
        dest[i]=0;
        return dest;
}
```

<Q> What is `size_t`?

→ This is an unsigned integer type used to represent the sizes of objects. The result of the sizeof operator is of this type. On systems using the GNU C Library, this will be `unsigned int` or `unsigned long int`.
Usage : size_t is preferred to declare any arguments or variables that holds sizes of objects

strncpy() :

→ declaration :
```
char *strncpy(char *dest, const char *src, size_t n);
```

<Ex>
```
strcpy(s1,"abcdefgh");
strcpy(s2,"012345678");
strncpy(s2+2,s1+3,3);    // s2 : 01def5678
```

→ If there is no null byte among the first n bytes of source, the string placed in destination will not be null-terminated. If the length of source is less than n, strncpy() writes additional null bytes to destination to ensure that a total of n bytes are written.

→ One valid (and intended) use of strncpy() is to copy a C string to a fixed length buffer while ensuring both that the buffer is not over flowed and that unused bytes in the target buffer are zeroed out.

→ If there is no terminating null byte in the first n bytes of src, strncpy() produces an unterminated string in dest. You can force termination using something like the following:

```
strncpy(dest, str, n);
if (n > 0)
        dest[n - 1]= '\0';
```

→ return : return a pointer to the destination string

→ strncpy() user defined :
```
char * Ustrncpy(char *dest, const char *src, size_t n)
{
       int i;

       for (i=0;i<n && src[i];i++)
              dest[i]=src[i];
       for (;i<n;i++)
              dest[i]='\0';
       return dest;
}
```

memmove() :

→ declaration :
```
void *memmove(void *dest, const void *src, size_t n);
```

→ memmove requires third argument as how many bytes to be copied, same as in strncpy().

→ for copying strings in overlapping areas, memmove() function is used.

→ using memmove to copy string, bytes in source are first copied into a temporary array that does not overlap source or destination, and the bytes are then copied from the temporary array to destination.

<Ex>

```
strcpy(s1,"abcdefgh");
memmove(s1+4,s1+3,4);
printf("s1 : %s\n",s1);

o/p > s1 : abcddefg
```

→ memmove() user defined :

```
void *Umemmove(void *dest,const void *src,size_t n)
{
        char temp[n+1];
        int i;
        for(i=0;i<n;i++)
                temp[i]=((char *)src)[i];
        for(i=0;i<n;i++)
                ((char *)dest)[i]=temp[i];
        return dest;
}
```

→ return : returns a pointer to destination

strcat() :

→ declaration :

```
char *strcat(char *dest, const char *src);
```

→ The strcat() function appends the src string to the dest string, overwriting the terminating null byte ('\0') at the end of dest, and then adds a terminating null byte. The strings may not overlap, and the dest string must have enough space for the result. If dest is not large enough, program behavior is unpredictable. As with strcat(), the resulting string in dest is always null terminated.

<Ex>

```
strcpy(s1,"abcdefgh");
strcat(s1,"xyz");              // s1 : abcdefghxyz
```

→ in above example, equivalent statement like `strcat(s1,"xyz");` can be also implemented using strcpy as shown below

```
strcpy(s1+strlen(s1),"xyz");
```

<Ex>

```
strcpy(s1,”xyz”);
srtcat(“abc”,xyz”);
strcat(s1,”123”);
```

→ no compile time error is reported in this program because strcat()'s arguments are addresses. So addresses of strings of text section is passed to strcat(). So strcat() will try to modify data in text section which is prohibited so segmentation fault will be reported.

<Ex>

```
strcpy(s1,”abc”);
strcat(s1+1,”123”);      // s1 : abc123
```

<Ex>

```
strcpy(s1,”abc”;
strcat(s1+5,”123”);      // s1 : abc
```

→ will search for null from s1+5 location from which null is not guaranteed to be present, we can get segmentation fault if null is not found in the user space memory.

→ strcat() user defined :

```
char *Ustrcat(char *dest, const char *src)
{
        int len=strlen(dest);
        int i;
        for (i=0;src[i];i++)
                dest[len+i]=src[i];
        dest[len+i]='\0';
        return dest;
}
```

→ return : pointer to the destination string

strncat() :

→ declaration :

```
char *strncat(char *dest, const char *src, size_t n);
```

→ The strncat() function is similar to strcat(), except that it will use at most n bytes from source; and source does not need to be null-terminated if it contains n or more bytes. If source contains n or more bytes, strncat() writes n+1 bytes to destination (n from source plus the terminating null byte). Therefore, the size of destination must be at least strlen(destination)+n+1.

→ strncat() user defined :

```
char *strncat(char *dest, const char *src, size_t n)
```

```
{
        int len=strlen(dest);
        int i;
        for (i=0;i<n && src[i];i++)
                dest[len+i]=src[i];
        dest[len+i]='\0';
               return dest;
}
```

→ return : returns pointer to the resulting destination string.

<Assignment> Input 3 strings and store in s1,s2,s3. Copy all the 3 strings into s4, one after another separated by space.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[10],s2[10],s3[10],s4[10];
        printf("Enter s1 : ");
        gets(s1);
        printf("Enter s2 : ");
        gets(s2);
        printf("Enter s3 : ");
        gets(s3);
        strcpy(s4,s1);
        strcat(s4," ");
        strcat(s4,s2);
        strcat(s4," ");
        strcat(s4,s3);
        printf("String s4 : %s\n",s4);
}
```

```
o/p > Enter s1 : abc
      Enter s2 : xyz
      Enter s3 : 123
      String s4 : abc xyz 123
```

<Assignment> input 2 strings s1 and s2. Insert s2 in given position in s1.
1). using temp array
2). without using temp array

1) using temp array

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],s2[20],temp[20];
        int pos;
```

```
        printf("Enter s1 : ");
        gets(s1);
        printf("Enter s2 : ");
        gets(s2);
        printf("Enter position : ");
        scanf("%d",&pos);
        strcpy(temp,s1+pos);
        strcpy(s1+pos,s2);
        strcat(s1,temp);
        printf("modified s1 : %s\n",s1);
}
```

2) without using temp array

```
        memmove(s1+pos+strlen(s2),s1+pos,strlen(s1+pos)+1);
        memmove(s1+pos,s2,strlen(s2));
        printf("modified s1 : %s\n",s1);
```

```
o/p > Enter s1 : chintan
      Enter s2 : patel
      Enter position : 3
      modified s1 : chipatelntan
```

<Assignment> WAP to input a string and remove all non alpha bates.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],s2[20];
        int i,len;
        printf("Enter s1 : ");
        gets(s1);
        for(i=0;s1[i];i++)
        {
                if(((s1[i]>='a')&&(s1[i]<='z'))||
((s1[i]>='A')&&(s1[i]<='Z')))
                        continue;
                else
                {
                        memmove(s1+i,s1+i+1,strlen(s1+i));
                        i--;
                }
        }
        printf("modified s1 : %s\n",s1);
}
```

```
o/p > Enter s1 : c7h&i.n,t5a) n
      modified s1 : chintan
```

<Assignment> WAP to input a string and a character and remove all occurrences of the character from the string.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],ch;
        int i;
        printf("Enter s1 : ");
        gets(s1);
        printf("Enter character : ");
        scanf("%c",&ch);
        for(i=0;s1[i];i++)
        {
                if(s1[i]==ch)
                {
                        memmove(s1+i,s1+i+1,strlen(s1+i+1)+1);
                        i--;
                }
        }
        printf("modified s1 : %s\n",s1);
}

o/p > Enter s1 : embedded
      Enter character : e
      modified s1 : mbddd
```

→ instead of checking every character in string one by one in our string to search for a particular character in a string, there is an inbuilt function available : strchr()

strchr() :

→ declaration :

```
char *strchr(const char *s, int c);
```

→ The strchr() function returns a pointer to the first occurrence of the character c in the string s. Here "character" means "byte"; these functions do not work with wide or multi byte characters.

→ The terminating null byte is considered part of the string, so that if c is specified as '\0', these functions return a pointer to the terminator.

<Ex>

```
char s[20]="chintan",ch='i';
if(strchr(s,ch)==NULL)
        printf("Not found\n");
```

```
else
        printf("Found at index %u\n",strchr(s,ch)-s);

o/p > Found at index 2
```

→ but here calling strchr is called every time whenever location of the character 'i' in string “chintan” is needed, instead of that, we can collect return of strchr in a character pointer and use that pointer when needed as shown below.

<Ex>

```
char s[20]="chintan",ch='i',*p;
p=strchr(s,ch);
if(p==NULL)
        printf("Not found\n");
else
        printf("Found at index %u\n",p-s);

o/p > Found at index 2
```

→ strchr() user defined :

```
char *Ustrchr(const char *s,int c)
{
        int i;
        for(i=0;s[i];i++)
        {
                if(s[i]==c)
                        return s+i;
        }
        return NULL;
}
```

→ Note that we are passing character to be searched to strchr() but in function data passing, character's ASCII values are passed so characters are promoted to integer. Thats why in formal arguments of strchr(), int is used instead of char.

→ return : The strchr() function return a pointer to the first occurrence of matched character or NULL if the character is not found.

<Assignment> WAP to input a string and a character and remove all occurrences of the character from the string.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],ch,*p;
        int i;
        printf("Enter s1 : ");
```

```
        gets(s1);
        printf("Enter character : ");
        scanf("%c",&ch);
        while(p=strchr(s1,p))
                memmove(p,p+1,strlen(p+1)+1);
        printf("modified s1 : %s\n",s1);
}
```

<Assignment> Remove extra repetitions of all characters in a given string.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[20],*p;
        int i;
        printf("Enter string : ");
        gets(s);
        for(i=0;s[i];i++)
        {
                while(p=strchr(s+i+1,s[i]))
                        memmove(p,p+1,strlen(p+1)+1);
        }
        puts(s);
}
```

```
o/p > Enter string : embedded
      Modified string : embd
```

<Assignment> WAP to input a string and find out how many characters are repeated.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[20],*p,temp[20];
        int i,cnt=0;
        printf("Enter string : ");
        gets(s);
        strcpy(temp,s);
        for(i=0;temp[i];i++)
        {
                if(strchr(temp+i+1,temp[i]))
                {
                        cnt++;
                        while(p=strchr(temp+i+1,temp[i]))
                                memmove(p,p+1,strlen(p+1)+1);
                }
        }
        printf("No. of characters repeating : %d\n",cnt);
}
```

```
o/p > Enter string : embedded
      No. of characters repeating : 2
```

strrchr() :

→ declaration :

```
char *strrchr(const char *s, int c);
```

→ The strrchr() function returns a pointer to the last occurrence of the character c in the string s.

→ strrchr() user defined :

```
char *Ustrrchr(const char *s,int c)
{
        int i,l=-1;
        for(i=0;s[i];i++)
        {
                if(s[i]==c)
                        l=i;
        }
        if(l>=0)
                return s+l;
        else
                return NULL;
}
```

→ return : The strchr() function return a pointer to the last occurrence of matched character or NULL if the character is not found.

strstr() :

→ declaration :

```
char *strstr(const char *haystack, const char *needle);
```

→ The strstr() function finds the first occurrence of the substring needle in the string haystack. The terminating null bytes ('\0') are not compared.

<Ex>

```
char s1[20]="abcabcdefbcdebcd",s2[20]="bcd",*p;
int i;
p=strstr(s1,s2);
if(p)
        printf("Found at %u\n",p-s1);
else
        printf("Not found\n");
```

```
o/p > Found at 4
```

<Ex>

```
char s1[20]="abcabcdefbcdebcd",s2[20]="bcd",*p;
int i,cnt=0;
p=s1;
while(p=strstr(p,s2))
{
        cnt++;
        p++;
}
printf("String s2 occures %d times in s1\n",cnt);

o/p > String s2 occures 3 times in s1
```

→ strstr() user defined :

```
char *Ustrstr(const char *haystack,const char *needle)
{
        int i,j;
        for(i=0;haystack[i];i++)
        {
                if(haystack[i]==needle[0])
                {
                        for(j=1;needle[j];j++)
                        {
                                if(haystack[i+j]!=needle[j])
                                        break;
                        }
                        if(needle[j]=='\0')
                                return haystack+i;
                }
        }
        return NULL;
}
```

→ return : return a pointer to the beginning of the substring, or NULL if the substring is not found.

<Assignment> remove all occurrences of s2 from s1.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        while(p=strstr(s1,s2))
                memmove(p,p+strlen(s2),strlen(p+strlen(s2))+1);
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintanpatelchintan
      Enter string s2 : patel
      s1 : chintanchintan
```

<Assignment> Reverse s2 in s1

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p,ch;
        int i,j;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        while(p=strstr(s1,s2))
        {
                j=strlen(s2)-1;
                for(i=0;i<j;i++,j--)
                {
                        ch=*(p+i);
                        *(p+i)=*(p+j);
                        *(p+j)=ch;
                }
        }
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintanpatel chintanpatel
      Enter string s2 : patel
      s1 : chintanletap chintanletap
```

<Assignment> Remove extra consecutive blank spaces.

```
#include<stdio.h>
#include<string.h>
main()
{
        char s[100],*p,*q,ch=32;
        printf("Enter string : ");
        gets(s);
        p=s;
        while(p=strchr(p,ch))
        {
                q=++p;
                while(*p==ch)
                        p++;
                memmove(q,p,strlen(p)+1);
        }
        printf("s : %s\n",s);
}
```

```
o/p > Enter string : chintan     patel     patel
      s : chintan patel patel
```

<Assignment> Hide s2 in s1

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p;
        int i;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        while(p=strstr(s1,s2))
        {
                j=strlen(s2);
                for(i=0;i<j;i++)
                        s1[(p-s1)+i]='*';
        }
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintanpatelchintan
      Enter string s2 : patel
      s1 : chintan*****chintan
```

→ this assignment can also be done by built in function memset().

<Assignment> Hide s2 in s1

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100],*p;
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        p=s1;
        while(p=strstr(p,s2))
        {
                memset(p,'*',strlen(s2));
                p++;
        }
        printf("s1 : %s\n",s1);
}
```

```
o/p > Enter string s1 : chintan patel chintan
```

```
      Enter string s2 : patel
      s1 : chintan ***** chintan
```

memset() :

→ declaration :

```
void *memset(void *s, int c, size_t n);
```

→ The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.

→ memset() user defined :

```
void *Umemset(void *s,int c,size_t n)
{
        while(n--)
        {
                *(char *)s=c;
                ((char *)s)++;
        }
        return s;
}
```

→ return : returns pointer to memory area s.

strcmp() :

→ declaration :

```
int strcmp(const char *s1, const char *s2);
```

<Ex>

```
#include<stdio.h>
#include<string.h>
main()
{
        char s1[100],s2[100];
        printf("Enter string s1 : ");
        gets(s1);
        printf("Enter string s2 : ");
        gets(s2);
        if(strcmp(s1,s2)==0)
                printf("Equal\n");
        else
                printf("Not Equal\n");
}
```

```
o/p > Enter string s1 : chintan
      Enter string s2 : patel
      Not Equal
```

→ The strcmp() function compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

→ strcmp() user defined :

```
int Ustrcmp(const char *s1,const char *s2)
{
        while(*s1 && *s2)
        {
                if(*s1 != *s2)
                        break;
                s1++;
                s2++;
        }
        if(*s1 == *s2)
                return 0;
        else
        {
                if(*s1 > *s2)
                        return 1;
                else
                        return -1;
        }
}
```

→ return : returns an integer less than, equal to or greater than zero, if s1 is found respectively to be less than, to match or to be greater than s2.

strncpy() :

→ declaration :

```
int strncmp(const char *s1, const char *s2, size_t n);
```

→ The strncmp() function is similar, except it compares the only first (at most) n bytes of s1 and s2.

→ return : returns an integer less than, equal to or greater than zero, if fisrt n bytes of s1 are found respectively to be less than, to match or to be greater than first n bytes of s2.

<Assignment> reverse given string

```
#include<stdio.h>
#include<string.h>
void strrev(char *p)
{
        int n=strlen(p);
        char *q,temp;
        for(q=p+n-1;p<q;p++,q--)
```

```
        {
                temp=*p;
                *p=*q;
                *q=temp;
        }
}
main()
{
        char s[100];
        printf("Enter string : ");
        gets(s);
        strrev(s);
        printf("s : %s\n",s);
}

o/p > Enter string : chintan
      s : natnihc
```

<Assignment> implement your own strrstr()

```
#include<stdio.h>
char * Ustrrstr(char s1[], char s2[])
{
        int i,j;
        char *p=NULL;
        for(i=0;s1[i];i++,j=1)
        {
                if(s1[i]==s2[0])
                {
                        for(j=1;s2[j];j++)
                        {
                                if(s1[i+j]!=s2[j])
                                        break;
                        }
                        if(s2[j]=='\0')
                                p=s1+i;
                }
        }
        if(p)
                return p;
        else
                return NULL;
}
main()
{
        char s1[20],s2[20],*p;
        puts("enter s1:");
        gets(s1);
        puts("enter s2:");
        gets(s2);
        p=Ustrrstr(s1,s2);
```

```
        if(p)
        {
                printf("found at index=%d\n",p-s1);
        }
        else
                printf("not found\n");
}
```