MODULE 5

STRINGS

STRING TAXONOMY – Hierarchical description of different types strings and manipulation of strings.

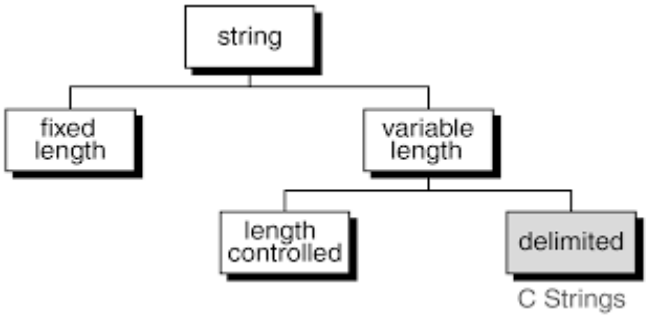
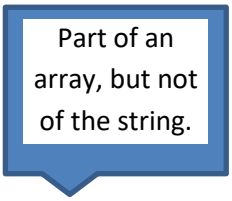


Figure: String Taxonomy

- **String** is an array of Characters and terminated by NULL character which is denoted by the escape sequence '\0'.
- **Fixed-length strings** –
  - ✓ When storing a string in a fixed length format, you need to specify an appropriate size for the string variable.
  - ✓ If the size is too small, then you will not be able to store all the elements in the string.
  - ✓ On the other hand, if the string size is large, then unnecessarily memory space will be wasted.
- **Variable-length strings** –
  - ✓ In variable length format, the string can be expanded or contracted to accommodate the elements in it.
  - ✓ For example, if you declare a string variable to store the name of a student, If a student has a long name of say 20 characters, then the string can be expanded to accommodate 20 characters, On the other hand, a student name has only 5 characters, then the string variable can be contracted to store only 5 characters.
  - ✓ However, to use a variable-length string format you need a technique to indicate the end of elements that are a part of the string.
  - ✓ This can be done either by using length-controlled string or a delimiter.
- **Length-controlled strings** –
  - ✓ In a length-controlled string, you need to specify the number of characters in the string
  - ✓ This count is used by string manipulation functions to determine the actual length of the string variable.

- **Delimited strings** –
  - ✓ In this format, the string is ended with a delimiter.
  - ✓ The delimiter is then used to identify the end of the string.
  - ✓ For example, in English language every sentence is ended with a full-stop (.). Similarly, in C we can use any character such as comma, semicolon, colon, dash, null character, etc. as the delimiter of a string,
  - ✓ However, null character is the most commonly used string delimiter in the C language.

char str[10];
gets (str);
If the user enters HELLO, then array of characters can be given



H	E	L	L	O	\0				

Although the array has 10 locations, only the first 6 locations will be treated as a string.

Figure: Delimited String

OPERATIONS ON STRINGS

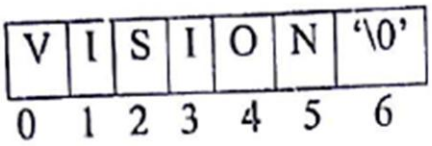
1. Finding the Length of a String
2. Converting Characters of a String into Upper Case
3. Converting Characters of a String Into Lower Case
4. Concatenating Two Strings to Form New String
5. Appending a String to Another String
6. Comparing Two Strings
7. Reversing a String
8. Extracting a Substring from Left of the string
9. Extracting a Substring from Right of the String
10. Extracting a Substring from the Middle of a String
11. Inserting a String in Another String
12. Indexing
13. Deleting a String from the Main String
14. Replacing a Pattern with Another Pattern in a String

Module -4

Strings and Pointers

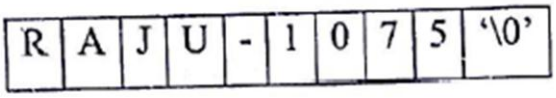
Strings

Introduction to strings: A string is an array of characters terminated by null character which is denoted by the escape sequence '\0'. Consider a string "VISION" . This string is stored in the form of an array as shown below:



The sequence of characters enclosed within two double quotes is called string constant. Ex: "SWAMI", "PERSEVERANCE", "TGS-123", "HASSAN".

The literal string or string constant is stored in consecutive bytes in memory and the compiler places the null character at end. Figure below shows the storage of a literal string " RAJU -1075":



Declaring, Initializing, Printing and Reading Strings

a. **Declaring String Variables:** In C string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is :

```
char string_name[size];
```

The size determines the number of characters in the string name including the null character.

Examples :
```
char city[10];
char name[30];
```

Null character: The null character denoted as '\0' marks end of the string. In some cases you must explicitly insert a null character to terminate a string. In literal string the null character is inserted automatically.

b. **Initializing String Variables:** Like numeric arrays character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms

```
char  city[9] = "New York" ;
char  city[9] = { 'N','E','W',' ','Y','O','R','K','\0'};
```

The declaration **char st1[ ] = {'G','O','O','D','\0'};** defines the array string as a five elements array automatically even though the size is not given.

One can declare the size much larger than the string size in the initializer i.e. the statement **char str[10] = "GOOD"** ; is permitted . In this case the computer creates a character array of size 10, places the value "GOOD" in it terminates with the null character and initializes all other remaining elements to NULL. The storage will look like

G	O	O	D	\0	\0	\0	\0	\0	\0
0	1	2	3	4	5	6	7	8	9

The following declaration is illegal: Here

```
char   str2[3] = "GOOD" ;
```

This will result in a compile time error. Also note that we cannot separate the initialization from declaration. i.e.

```
char   str3[5];
str3 = "GOOD";
```

Similarly, `char s1[4] = "abc";`
`char s2[4];`
`s2 = s1;`
is not allowed. An array name cannot be used as the left operand of an assignment operator.

Difference between Array of characters and string : Unlike array of characters the string is created as a **unit** and is terminated by the NULL character.

c. **Printing a string:** String is printed using a printf statement. As long as the string is terminated by a null character. The printf function prints all the character upto but not including the null character.

Example:
```
char first[11] = "ANDY";
printf(" The name is %s\n",first);
```

Example : Program to store the string "India is great" in the array country and display the string.

```
main()
{
    char country[15] = "India is Great";
    printf("%s",country);
}
```

d. **Reading Strings using scanf:** The conversion specification for reading a string using scanf is %s just as in printf. One must *not use ampersand (&)* symbol while reading the string.

Example :

```
char    name[20];
printf("\n Enter the name\n");
scanf("%s",name);
printf("\n The entered name is \n");
printf("%s",name);
```

Note : *The scanf functions stops reading at the first white space character.If the input for the above program segment is MK Gandhi, the output will be MK.*

String Manipulation Functions From the Standard Library String operations

C language provides number of built in string handling functions. All the string manipulation functions are defined in the standard library **string.h** . The standard library **string.h** contains many functions for string manipulation. Some of the string handling or manipulation functions used in C are:

1. **strlen( )**
2. **strcpy( )**
3. **strncpy( )**
4. **strcmp( )**
5. **strncmp( )**
6. **strcan( )**
7. **strlwr( )**
8. **strupr( )**
9. **strrev( )**

1. strlen( ) : This function is used to find the length of the string in bytes .

The general form of a call to strlen is the following :

```
length = strlen(str);
```

The parameter to strlen , str is a string . The value it returns is an integer representing the current length of str in bytes excluding the null character.

Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
 char s1[20];
 int length;

 printf("Enter the string: ");
 scanf("%s", s1);

 length = strlen(s1);

 printf("The length of the given string is %d", length);
} //End of main
```

OUTPUT

```
Enter the string: NAGARAJ
The length of the given string is 7
```

2. strcpy( ) : This function copies the string from one variable to another variable. The strcpy() function will copy the entire string including the terminating null character to the new location .

General form of a call to strcpy :

```
strcpy(dest,source) ;
```

here the characters in source string will get copied to destination string.

Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
 char s1[20], s2[20];
 printf("Enter first string: ");
 scanf("%s", s1);
 strcpy(s2, s1);
 printf("\nFirst string is = %s", s1);
 printf("\nSecond string is = %s", s2);
} //End of main
```

OUTPUT

```
Enter first string: PROGRAMMING
First string is = PROGRAMMING
Second string is = PROGRAMMING
```

3. strncpy() : This function copies the string from one variable to another variable, but only up to the specified length say n . The general form of a call to the strncpy function is the following :

```
strncpy(dest,source,n) ;
```

Where, dest –> Destination string
source -> source string
n-> is the number of characters

```
#include<stdio.h>
#include<string.h>
void main( )
{
 char s1[20], s2[20];
 printf("Enter first string: ");
 scanf("%s", s1);
 strncpy(s2, s1, 7);
 printf("\nFirst string is = %s", s1);
 printf("\nSecond string is = %s", s2);
} //End of main
```

OUTPUT

```
Enter first string: PROGRAMMING
First string is = PROGRAMMING
Second string is = PROGRAM
```

4. **strcmp( ) :** It is used to compare one string with another and it is case sensitive . The general form of the call to **strcmp** is the following where either parameter may be a string literal or variable :

```
result = strcmp(first, second);
```

The function strcmp returns an integer determined by the relationship between the strings pointed to by the two parameters. The result may take any of the following value :

Result > 0, if first > second
Result = 0, if first = second
Result < 0, if first < second

```
#include<stdio.h>
#include<string.h>
void main( )
{
  char s1[20], s2[20];
  int n;
 printf("Enter first string: ");
 scanf("%s", s1);
printf("Enter second string: ");
 scanf("%s", s2);
  n = strcmp(s1, s2);
if (n==0)
     printf("\nBoth strings are equal);
else
     printf("\nBoth strings are not equal);
} //End of main
```

OUTPUT 1

```
Enter first string: PROGRAM
Enter second string: NAGARAJ
Both strings are not equal
```

OUTPUT 2

```
Enter first string: PROGRAM
Enter second string: PROGRAM
Both strings are equal
```

5. **strncmp( ) :** It allows us to compare a block of **n** characters from one string with those in another.

General form of a call to strncmp : The general form of a call to the strncmp function is as follows;

```
result = strncmp(firststring,secondstring,n);
```

where firststring and second string are pointers to strings and n is an integer.

Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
  char s1[20], s2[20];
 printf("Enter first string: ");
 scanf("%s", s1);
printf("Enter second string: ");
 scanf("%s", s2);
  n = strcnmp(s1, s2, 7);
if (n==0)
      printf("\nBoth strings are equal up to 7
                characters);
else
      printf("\nBoth strings are not equal up to
                7 characters);
} //End of main
```

OUTPUT 1

Enter first string: PROGRAM

Enter second string: PROGRAMMING

Both strings are equal up to 7 characters

OUTPUT 2

Enter first string: PROGRAM

Enter second string: NAGARAJ

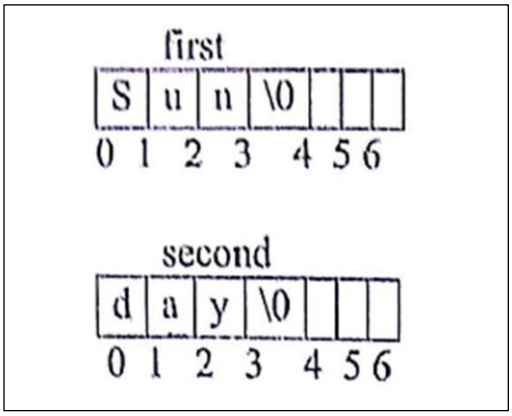
Both strings are not equal up to 7 characters

6. **strcat( ):** This function is used to join (concatenate) two strings . Thc resulting string has only one null character at the end. The general form of the call to the strcat function is the following, where both first and second are pointers to strings :

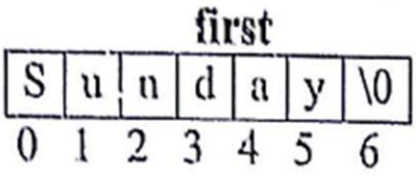
```
strcat(first, second);
```

Example:

```
char   first[7] = "Sun";
char   second[7]="day";
```



After calling **strcat(first, second),** the **first string** takes the value as follows:



Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
 char s1[20], s2[20];
 printf("Enter first string: ");
 scanf("%s", s1);
printf("Enter second string: ");
 scanf("%s", s2);
 strcat(s1, s2);
 printf("\nThe concatenated string is
                %s", s1);
} //End of main
```

OUTPUT

Enter first string: PROGRAM

Enter second string: ING

The concatenated string is PROGRAMMING

7. **strlwr( ) :** This function converts uppercase characters to lowercase .

The general syntax is **strlwr(str) :**

Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
  char s1[20];
 printf("Enter first string in lower case : \n");
 scanf("%s", s1);
 strupr(s1);
 printf("Entered string in upper case is : \n", s1);
} //End of main
```

OUTPUT

Enter first string in lower case:

program

Entered string in upper case is :
PROGRAM

8. **strupr( )** : This function converts lowercase characters to uppercase .

The general syntax is **strupr(str)**

Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
 char s1[20];
 printf("Enter first string in upper case : \n");
 scanf("%s", s1);
 strlwr(s1);
 printf("Entered string in lower case is : \n", s1);
} //End of main
```

OUTPUT

```
Enter first string in upper case:
PROGRAM
Entered string in lower case is : program
```

9. **strrev()** : This function reverses a given string . The general syntax is

strrev(str) .

Example:

```
#include<stdio.h>
#include<string.h>
void main( )
{
 char s1[20];
 printf("Enter the string  : \n");
 scanf("%s", s1);
 strrev(s1);
 printf("\nReverse of the  string is : ", s1);
} //End of main
```

OUTPUT

```
Enter the string :
RAMA
Reverse of the  string is : AMAR
```

String Input/Output Functions

The **scanf** and **printf** functions are used to process the individual units like characters, integers, strings separated by white space characters. The individual units are called **tokens.** The scanf and printf are called *token oriented* Input and output functions. In C language the line oriented *Input/ output* is done using **gets** and **puts** functions.

Reading and Writing using gets( ) and puts( ) : The functions gets() and puts() are called line oriented I/O functions and can be used to process entire line. The gets( ) function can be used to read entire line including white spaces. The puts() function can be used to print the line.

Printing a string using puts() : The general form of puts function is as given below :

```
puts (str) ;
```

The puts function takes a string or a pointer to string as a parameter and prints the string on the screen. In addition to printing a string it also prints the newline character giving the effect of printing an entire line.

Reading a string: The general form of gets function is as given below:

```
gets(str);
```

Example:

```
#include<stdio.h>
void main( )
{
 char s1[20];
 printf("Enter the string  : \n");
 gets(s1);

 printf("\nThe  entered string is : ");
 puts(s1);
} //End of main
```

OUTPUT

```
Enter the string :
NAGARAJ
The  entered string is : NAGARAJ
```

Problems with gets( )

1. It does not check whether there is enough space for the input string.
2. When reading a string using gets( ), we have to make sure that the values entered are shorter than the variable into which we are storing data.

Array of Strings : A string is an array of characters . An array of strings is an array of arrays of characters. To create an array of strings we should use two dimensional array as shown below :

```
char   a[row][col] ;
```

a- Name of the array
row- Number of strings
col – Maximum length of each string

Suppose it is required to store the names of 5 students. In such case we can have the following declarations.

Suppose it is required to store the names of 5 students. In such case we can have the following declarations.

```
char a[5][11] = {
                  "DURYODHANA",
                  "RAVANA",
                  "KUMBAKARANA",
                  "KEECHAKA",
                  "BAKASURA"
                };
```

The above strings are stored in a two dimensional arrays as follows :

a =

	0	1	2	3	4	5	6	7	8	9	10
0	D	U	R	Y	O	D	H	A	N	A	\0
1	R	A	V	A	N	A	\0				
2	K	U	M	B	A	K	A	R	N	A	\0
3	K	E	E	C	H	A	K	A	\0		
4	B	A	K	A	S	U	R	A	\0		

Miscellaneous String And Character Functions

CHARACTER MANIPULATION FUNCTIONS

Illustrates some character functions contained in ctype.h.

1. isalnum(intc)
2. isalpha(intc)
3. iscntrl(int c)
4. isdigit(int c)
5. isgraph()
6. isprint(int c)
7. islower(int c)
8. isupper(int c)
9. ispunct(int c)
10. isspace(int c)
11. isxdigit(int c)
12. tolower(int c)
13. toupper(int c)

STRING MANIPULATION FUNCTIONS

1. strcat Function - String concatenate
2. strncat Function - string n concatenate
3. strchr Function - string character occurrence
4. strrchr Function - string last occurrence
5. strcmp Function - string compare
6. strncmp Function - string n compare
7. strcpy Function - string copy
8. strncpy Function - string n copy
9. strlen Function - string length
10. strstr Function
11. strspn Function
12. strcspn Function
13. strpbrk Function
14. strtol Function
15. strtod Function
16. atoi()Function
17. atof()Function
18. atol()Function

```
    j++;
  }
  if(str[j]=='\0')
    copy_loop=k;
  new_text[n] = text[copy_loop];
  i++;
  copy_loop++;
  n++;
 }
new_str[n]='\0';
printf("\n The new string is: ");
puts(new_text);
getch();
return 0;
}
```

Output

```
Enter the main text: Hello, how are you?
Enter the string to be deleted: , how are you?
The new string is: Hello
```

13.4.14 Replacing a Pattern with Another Pattern in a String

Replacement operation is used to replace the pattern P1 by another pattern P2. This is done by writing, REPLACE(text, pattern1, pattern2)

For example, ("AAABBBCCC", "BBB", "X") = AAAXCCC ("AAABBBCCC", "X", "YYY")= AAABBBCC.

In the second example, there is no change as 'X' does not appear in the text. Figure 13.19 shows an algorithm to replace a pattern P1 with another pattern P2 in the text.

```
Step 1: [INITIALIZE] SET POS = INDEX(TEXT,P1)
Step 2: SET TEXT = DELETE(TEXT,POS,LENGTH(P1))
Step 3: INSERT(TEXT, POS,P2)
Step 4: EXIT
```

Figure 13.19 Algorithm to replace a pattern P_1 with another pattern P_2 in the text

The algorithm is very simple, where we first find the position POS, at which the pattern occurs in the text, then delete the existing pattern from that position, and insert a new pattern there. String matching refers to finding occurrences of a pattern string within another string.

15. Write a program to replace a pattern with another pattern in the text.

```
#include <stdio.h>
#include <conio.h>
int main()
{
  char str[200], pat[20], new_str[200],
  rep_pat[100];
  int i=0, j=0, k, n=0, copy_loop=0,
  rep_index=0;
  clrscr();
  printf("\n Enter the string: ");
  gets(str);
  printf("\n Enter the pattern to be replaced:
   ");
  gets(pat);
  printf("\n Enter the replacing pattern: ");
  gets(rep_pat);
  while(str[i]!='\0')
  {
  j=0,k=i;
    while(str[k]==pat[j] && pat[j]!='\0')
    {
      k++;
      j++;
    }
    if(pat[j]=='\0')
    {
       copy_loop=k;
       while(rep_pat[rep_index] !='\0')
       {
        new_str[n] = rep_pat[rep_index];
        rep_index++;
        n++;
       }
    }
   new_str[n] = str[copy_loop];
   i++;
   copy_loop++;
   n++;
   }
  new_str[n]='\0';
  printf("\n The new string is: ");
  puts(new_str);
  getch();
  return 0;
}
```

Output

```
Enter the string: How ARE you?
Enter the pattern to be replaced: ARE
Enter the replacing pattern: are
The new string is : How are you?
```

13.5 MISCELLANEOUS STRING AND CHARACTER FUNCTIONS

In this section, we will discuss some character and string manipulation functions that are part of header files—ctype.h, string.h, and stdlib.h.

13.5.1 Character Manipulation Functions

Table 13.2 illustrates some character functions contained in ctype.h.

Table 13.2 Functions in ctype.h

Function	Usage	Example
isalnum(int c)	Checks whether character c is an alphanumeric character	isalpha('A');
isalpha(int c)	Checks whether character c is an alphabetic character	isalpha('z');
iscntrl(int c)	Checks whether character c is a control character	scanf("%d", &c);iscntrl(c);
isdigit(int c)	Checks whether character c is a digit	isdigit(3);
isgraph()	Checks whether character c is a graphic or printing character. The function excludes the white space character	isgraph('!');
isprint(int c)	Checks whether character c is a printing character. The function includes the white space character	isprint('@');
islower(int c)	Checks whether the character c is in lower case	islower('k');
isupper(int c)	Checks whether the character c is in upper case	isupper('K');
ispunct(int c)	Checks whether the character c is a punctuation mark	ispunct('?');
isspace(int c)	Checks whether the character c is a white space character	isspace(' ');
isxdigit(int c)	Checks whether the character c is a hexadecimal digit	isxdigit('F');
tolower(int c)	Converts the character c to lower case	tolower('K') returns k
toupper(int c)	Converts the character c to upper case	tolower('k') returns K

13.5.2 String Manipulation Functions

In this section we will look at some commonly used string functions present in the `string.h` and `stdlib.h` header files.

`strcat` *Function*

Syntax:

```
char *strcat(char *str1, const char *str2);
```

Programming Tip: Before using string copy and concatenating functions, ensure that the destination string has enough space to store all the elements so that memory overwriting does not take place.

The `strcat` function appends the string pointed to by `str2` to the end of the string pointed to by `str1`. The terminating null character of `str1` is overwritten. The process stops when the terminating null character of `str2` is copied. The argument `str1` is returned. Note that `str1` should be big enough to store the contents of `str2`.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[50] = "Programming";
  char str2[] = "In C";
  strcat(str1, str2);
  printf("\n str1: %s", str1);
  return 0;
}
```

Output

```
str1: Programming In C
```

`strncat` *Function*

Syntax:

```
char *strncat(char *str1, const char *str2,
size_t n);
```

This function appends the string pointed to by `str2` to the end of the string pointed to by `str1` up to `n` characters long. The terminating null character of `str1` is overwritten. Copying stops when `n` characters are copied or the terminating null character of `str2` is copied. A terminating null character is appended to `str1` before returning to the calling function.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[50] = "Programming";
  char str2[] = "In C";
  strncat(str1, str2, 2);
  printf("\n str1: %s", str1);
  return 0;
}
```

Output

```
str1: Programming In
```

`strchr` *Function*

Syntax:

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

The `strchr ( )` function searches for the first occurrence of the character `c` (an unsigned char) in the string pointed to by the argument `str`. The function returns a pointer pointing to the first matching character, or `NULL` if no match is found.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str[50] = "Programming In C";
  char *pos;
```

```
pos = strchr(str, 'n');
if(pos)
  printf("\n n is found in str at position
  %d", pos);
else
  printf("\n n is not present in the
  string");
return 0;
}
```

Output

```
n is found in str at position 9
```

strrchr *Function*

Syntax:

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

The strrchr ( ) function searches for the first occurrence of the character c (an unsigned char) beginning at the rear end and working towards the front in the string pointed to by the argument str, i.e., the function searches for the last occurrence of the character c and returns a pointer pointing to the last matching character, or NULL if no match is found.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str[50] = "Programming In C";
  char *pos;
  pos = strrchr(str, 'n');
  if(pos)
    printf("\n The last position of n is:
    %d", pos);
  else
    printf("\n n is not present in the
    string");
  return 0;
}
```

Output

```
The last position of n is: 13
```

strcmp *Function*

Syntax:

```
int strcmp(const char *str1, const char
*str2);
```

The strcmp function compares the string pointed to by str1 to the string pointed to by str2. The function returns zero if the strings are equal. Otherwise, it returns a value less than zero or greater than zero if str1 is less than or greater than str2 respectively.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[10] = "HELLO";
  char str2[10] = "HEY";
  if(strcmp(str1,str2)==0)
    printf("\n The two strings are
    identical");
  else
    printf("\n The two strings are not
    identical");
  return 0;
}
```

Output

```
The two strings are not identical
```

strncmp *Function*

Syntax:

```
int strncmp(const char *str1, const char
*str2, size_t n);
```

This function compares at most the first n bytes of str1 and str2. The process stops comparing after the null character is encountered. The function returns zero if the first n bytes of the strings are equal. Otherwise, it returns a value less than zero or greater than zero if str1 is less than or greater than str2, respectively.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[10] = "HELLO";
  char str2[10] = "HEY";
  if(strncmp(str1,str2,2)==0)
    printf("\n The two strings are
   identical");
  else
    printf("\n The two strings are not
   identical");
  return 0;
}
```

Output

```
The two strings are identical
```

strcpy *Function*

Syntax:

```
char *strcpy(char *st r1, const char *str2);
```

This function copies the string pointed to by str2 to str1 including the null character of str2. It returns the argument str1. Here str1 should be big enough to store the contents of str2.

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

```
#include <string.h>
int main()
{
  char str1[10], str2[10] = "HELLO";
  strcpy(str1,str2);
  printf("\n str1: %s", str1);
  return 0;
}
```

Output

```
HELLO
```

strncpy *Function*

Syntax:

```
char *strncpy(char *str1, const char *str2,
size_t n);
```

This function copies up to n characters from the string pointed to by str2 to str1. Copying stops when n characters are copied. However, if the null character in str2 is reached then the null character is continually copied to str1 until n characters have been copied. Finally, a null character is appended to str1. However, if n is zero or negative then nothing is copied.

Programming Tip:
Do not use string functions on a character array that is not terminated with a null character.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[10], str2[10] = "HELLO";
  strncpy(str1,str2, 2);
  printf("\n str1: %s", str1);
  return 0;
}
```

Output

```
HE
```

Note

To copy the string str2 in str1, a better way is to write

```
strncpy(str1, str2, sizeof(str1)-1);
```

This would enforce the copying of only that many characters for which str1 has space to accommodate.

strlen *Function*

Syntax:

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

This function calculates the length of the string str up to but not including the null character, i.e., the function returns the number of characters in the string.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str[] = "HELLO";
  printf("\n Length of str is: %d",
  strlen(str));
  return 0;
}
```

Output

```
Length of str is: 5
```

strstr *Function*

Syntax:

```
char *strstr(const char *str1, const char
*str2);
```

This function is used to find the first occurrence of string str2 (not including the terminating null character) in the string str1. It returns a pointer to the first occurrence of str2 in str1. If no match is found, then a null pointer is returned.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[] = "HAPPY BIRTHDAY TO YOU";
  char str2[] = "DAY";
  char *ptr;
  ptr = strstr(str1, str2);
  if(ptr)
    printf("\n Substring Found");
  else
    printf("\n Substring Not Found");
  return 0;
}
```

Output

```
Substring Found
```

strspn *Function*

Syntax:

```
size_t strspn(const char *str1, const char *str2);
```

The function returns the index of the first character in str1 that doesn't match any character in str2.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[] = "HAPPY BIRTHDAY TO YOU";
  char str2[] = "HAPPY BIRTHDAY JOE";
  printf("\n The position of first character in
  str2 that does not match with that in str1
  is %d", strspn(str1,str2));
```

```
    return 0;
}
```

Output

```
The position of first character in str2 that
does not match with that in str1 is 15
```

strcspn *Function*

Syntax:

```
size_t strcspn(const char *str1, const char
*str2);
```

The function returns the index of the first character in str1 that matches any of the characters in str2.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[] = "PROGRAMMING IN C";
  char str2[] = "IN";
  printf("\n The position of first character in
  str2 that matches with that in str1 is %d",
  strcspn(str1,str2));
  return 0;
}
```

Output

```
The position of first character in str2 that
matches with that in str1 is 8
```

strpbrk *Function*

Syntax:

```
char *strpbrk(const char *str1, const char
*str2);
```

The function strpbrk() returns a pointer to the first occurrence in str1 of any character in str2, or NULL if none are present. The only difference between strpbrk() and strcspn is that strcspn() returns the index of the character and strpbrk() returns a pointer to the first occurrence of a character in str2.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[] = "PROGRAMMING IN C";
  char str2[] = "AB";
  char *ptr = strpbrk(str1,str2));
  if(ptr==NULL)
    printf("\n No character matches in the two
    strings");
  else
    printf("\n Character in str2 matches with
    that in str1");
  return 0;
}
```

Output

```
No character matches in the two strings
```

strtok *Function*

Syntax:

```
char *strtok( char *str1, const char
  *delimiter );
```

The strtok() function is used to isolate sequential tokens in a null-terminated string, str. These tokens are separated in the string using delimiters. The first time that strtok is called, str should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. However, the delimiter must be supplied each time, though it may change between calls.

The strtok() function returns a pointer to the beginning of each subsequent token in the string, after replacing the token itself with a null character. When all tokens are left, a null pointer is returned.

```
#include <stdio.h>
#include <string.h>
main ()
{
  char str[] = "Hello, to, the, world of,
   programming";
  char delim[] = ",";
  char result[20];
  result = strtok(str, delim);
  while(result!=NULL)
  {
    printf("\n %s", result);
    result = strtok(NULL, delim);
  }
  getch();
  return 0;
}
```

Output

```
Hello
to
the
world of
programming
```

strtol *Function*

Syntax:

```
long strtol(const char *str, char **end, int
base);
```

The strtol function converts the string pointed by str to a long value. The function skips leading white space characters and stops when it encounters the first non-numeric character. strtol stores the address of the first

invalid character in str in *end. If there were no digits at all, then the strtol function will store the original value of str in *end. You may pass NULL instead of *end if you do not want to store the invalid characters anywhere.

Finally, the third argument base specifies whether the number is in hexadecimal, octal, binary, or decimal representation.

```
#include <stdio.h>
#include <stdlib.h>
main ()
{
  long num;
num = strtol("12345 Decimal Value", NULL, 10);
printf("%ld", num);
num = strtol("65432 Octal Value", NULL, 8);
printf("%ld", num);
num = strtol("10110101 Binary Value", NULL, 2);
printf("%ld", num);
num = strtol("A7CB4 Hexadecimal Value", NULL,
   16);
printf("%ld", num);
getch();
return 0;
}
```

Output

```
12345
27418
181
687284
```

strtod *Function*

Syntax:

```
double strtod(const char *str, char **end);
```

The function accepts a string str that has an optional plus ('+') or minus sign ('-') followed by either:

- a decimal number containing a sequence of decimal digits optionally consisting of a decimal point, or
- a hexadecimal number consisting of a "0X" or "0x" followed by a sequence of hexadecimal digits optionally containing a decimal point.

In both cases, the number may be optionally followed by an exponent ('E' or 'e' for decimal constants or a 'P' or 'p' for hexadecimal constants), followed by an optional plus or minus sign, followed by a sequence of decimal digits. For decimal and hexadecimal constants, the exponent indicates the power of 10 and 2, respectively, by which the number should be scaled.

```
#include <stdio.h>
#include <stdlib.h>
main ()
{
  double num;
  num = strtod("123.345abcdefg", NULL);
  printf("%lf", num);
  getch();
  return 0;
}
```

Output

```
123.345000
```

atoi() *Function*

Till now you must have understood that the value 1 is an integer and '1' is a character. So there is a huge difference when we write the two statements given below

```
int i=1;      // here i =1
int i='1';    /* here i =49, the ASCII value
   of character 1*/
```

Similarly, 123 is an integer number but '123' is a string of digits. What if you want to operate some integer operations on the string '123'? For this, C provides a function atoi that converts a given string into its corresponding integer.

The atoi() function converts a given string passed to it as an argument into an integer. The atoi() function returns that integer to the calling function. However, the string should start with a number. atoi() will stop reading from the string as soon as it encounters a non-numerical character. atoi() is included in the stdlib.h file. So before using this function in your program, you must include this header file. The syntax of atoi() can be given as,

```
int atoi(const char *str);
```

```
Example   i = atoi("123.456");
          RESULT: i = 123.
```

atof() *Function*

The atof() function converts the string that it accepts as an argument into a double value and then returns that value to the calling function. However, the string must start with a valid number. One point to remember is that the string can be terminated with any non-numerical character, other than "E" or "e". The syntax of atof() can be given as,

```
double atof(const char *str);
```

```
Example   x = atof("12.39 is the answer");
          RESULT: x = 12.39
```

atol() *Function*

The atol() function converts the string into a long int value. The atol function returns the converted long value to the calling function. Like atoi, the atol() function will read from a string until it finds any character that should not be in a long. Its syntax can be given as,

```
long atol(const char *str);
```

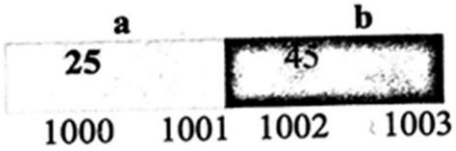
```
Example   x = atol("12345.6789");
          RESULT: x = 12345L.
```

POINTERS

Pointers and address: Consider the following declaration of two variables **a** and **b**:

```
int a = 25;
int b = 45;
```

The compiler allocates two bytes of memory starting from the address **1000** and stores the value **25** at that location and gives **a** as the name for that location. Similarly for second variable **b** as illustrated in the figure below :



The value of **a** and **b** can be accessed and printed as shown below:

```
printf("\n Value of a =%d\n",a);
printf("\n Value of b =%d\n",b);
```

In order to access the address of the variables one can use the address operator **&** as below:

&a -- -→ to get the address of variable **a**
&b ----→ to get the address of variable **b**

The addresses of **a** and **b** can be printed as shown below:

```
printf(" Address of a = %d \n",&a);
printf("Address of b = %d\n",&b);
```

In order to access the values using the address one must use ***dereferencing operator*** or pointer denoted by the symbol *. By prefixing the *operator to the address of variable, we can access the value stored in that address as below:

```
printf("\n  Value of  a = %d\n",*&a);
printf("\n Value of   b = %d\n",*&b);
```

Note: The operator * is known as *indirection operator or dereference operator* and the operator **&** is known as *address operator* or *reference operator*.

Example: **A program to print the values using variables and their addresses.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int  a =100;
        int  b = 450;

        printf("\n Value of a = %d \n",a);
        printf("\n   Value of b = %d \n",b);

          /* Accessing the address of variables */

        printf("\n Address of a = %d\n",&a);
        printf("\n Address of b = %d\n",&b);

          /* Accessing the data using the dereferencing operator */

        printf("\n The value of a =%d \n", *&a);
        printf("\n Value of b = %d\n",  *&b);

}
```

Pointer definition

A pointer is a variable that can hold the address of another variable.

The steps to be followed to use pointers:

a. Declare a data variable	Ex: int x;
b. Declare a pointer variable	Ex: int *p;
c. Initialize a pointer variable	Ex: p = &x;
d. Access data using pointer variable.	Ex: y= *p

Declaration of a pointer

Pointer variables should be declared before they are used.

The syntax to declare a pointer variable is:

```
Data_type  *identifier ;
```

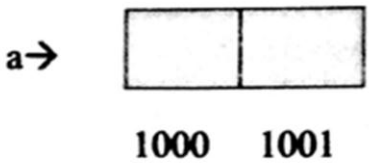
Example :

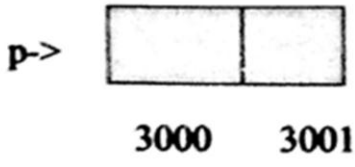
```
int     *pi;   /* pi is a pointer to int */
float   *pf;   /* pf is a pointer to float */
char    *pc;  /* pc is a pointer to character */
double  *pd; /* pd is a pointer to  double */
FILE    *fp;  /* fp is a pointer to FILE */
```

Initialization of Pointer variables: It is the process of assigning an address to the pointer variable . Consider the following declaration statements :

```
int   a;
int  *p ;
```

After declaration the variable **a** gets allocated in some memory location (address) say 1000 and the pointer **p** gets the address 3000 as illustrated below.

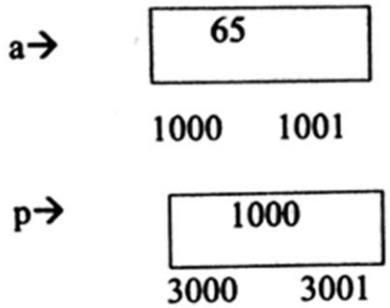




The pointer **p** can be initialized as follows:

```
a = 65;
p = &a;
```

After initialization the variable **a** holds the value **65** and the variable **p** holds the value of address of **a** as illustrated below :



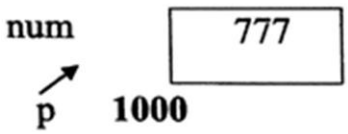
Note: ***A pointer can also be declared and initialized in the same statement as follows:***

```
int   x;
int *P = &x ;
```

Accessing Variables Through Pointers : The contents of variable can be accessed by de referencing a pointer. If a pointer p is pointing to a variable ***num***, then its value can be accessed by just saying *p. Here * operator acts as the dereference operator.

[**Indirection** :*The process of accessing the value of the variables indirectly by pointer pointing to the address of variable is called Indirection.*]

For example consider a variable **num** whose value is 777 and the pointer pointing to its address is **p** as given in figure below:



The value 777 can be accessed in 2 methods:

i) printf (" %d" , num); Output: 777 using variable

ii) printf (" %d" , *p); Output: 777 using variable

Example: **Program to Access the value of the variable using pointers.**

```
#include<stdio.h>
#include<conio.h>
main() { int   a,b,v;
         int   *p,*q;
            clrscr( );
         a=30;
         b=10;
         p = &a;
         q = &b;
         v = *p + *q;
            printf("\n The result is %d\n",v);
            getch();
       }
```

Output

The result is 40

MODULE 5

STRUCTURES UNIONAND ENUMERATED DATA TYPE

STRUCTURES

Definition of Structures: A structure is a collection of one or more variables of similar or dissimilar data types, grouped together under a single name.i.e. A structure is a derived data type containing multiple variables of **homogeneous or heterogeneous data types**. The structure is defined using the keyword **struct** followed by an identifier. This identifier is called **struct_tag**. The syntax and example is as follows:

Syntax for Defining Structure:

```
struct struct_tag
{
    type  var_1;
    type  var_2;
    ------------
    ------------
    type  var_n;
};
```

Here,

- The word ***struct*** is a keyword in C.
- ***struct_tag*** identifies the structure later in the program.
- ***type*** is the data type.
- ***var_1, var_2, …… var_n*** are the **members** of structures.

Example:

```
struct  student
{     char   name[20];
      int    roll_number;
      float  average_marks;
};
```

- **Declaring Structure Variables:** The syntax of declaring structure variables is shown below :

struct struct_tag v1,v2,v3,------vn;

Example: struct student s1,s2,s3;

- **Structure Initialization:**Assigning the values to the structure member fields is known as structure initialization. The syntax of initializing structure variables is similar to that of arrays i.e. all the elements will be enclosed within braces i.e. '{' and '}' and are separated by commas . The syntax is as shown below:

struct struct_tag variable = { v1,v2,v3------- vn};

Example: **struct student cse = { "Raj" ,21, 75.3 };**

The complete structure definition, declaration and initialization are as shown below.

Structure Definition :

```
struct employee
  { char name[20];
    int  salary;
    int  id;
  };
```

Structure Declaration

```
struct employee emp1,emp2;
```

Structure Initialization :

```
struct employee emp1 = { " Ramu",20000,1000};
struct employee emp2 = { "ThyagaRaju",10050,200};
```

- **Accessing Members of Structures (** *Using The Dot (.) Operator* **) :**

The member of structure is identified and accessed using the dot operator (.). The dot operator connects the member name to the name of its containing structure. The general syntax is :

struct_variable.membername;

Example: **Consider the structure definition and initialization shown below.**

```
struct employee
{    char  name[10];
     float  salary ;
     int  id ;
};
```

Struct employee E1 = { "Ramu" , 52000, 878 };

The members of structure variable E1 can be accessed using dot operator as follows.

E1.name ---> can access the string " Ramu"

E1. Salary ---> can access the value 52000

E1.id ---> can access the value 878

Displaying the various members of Structures: The values of members of structure variable can be displayed using the **printf** and **dot operator** as illustrated below:

```
printf("%s\n",E1.name);        prints the name of the employee
printf("%f\n",E1.salary);      prints the salary
printf("%d\n",E1.id);          prints the employee id
```

Reading the values for various members of structures : The values of members of structure variables can be read using **scanf or gets or getc and dot operator** as illustrated below :

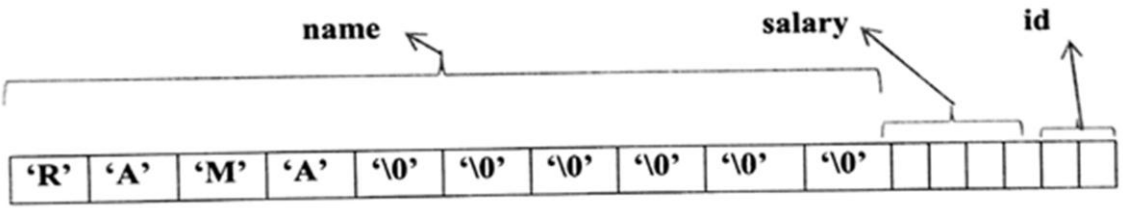
We can read the name of the *employee* , *salary* and *id* as follows.

```
gets(E1.name);              reads the employee name
scanf("%f",&E1.salary);     reads the salary of employee
scanf("%d",&E1.id);         reads the employee id
```

- **Memory Representation of Structure:**

When structure is declared the sum of memory required for each members will get allocated contiguously. For the structure employee the total amount of memory of 16 Bytes will get allocated. (10 Bytes for name , 4 Bytes for Salary and 2 Bytes for id.)

The memory representation for the structure employee E1 is shown below :



Programming Example: To Read and Write a structure called **Car** with members like name, model and Price.

```
#include<stdio.h>
#include<conio.h>
        struct Car
        {
                char    name[10];
                int     Model;
                float   price ;
        };

 void main()
 {
        struct Car c1;

          clrscr();
          printf("\n Enter Car name  :");
          gets(c1.name);
          printf("\n Enter Model No:");
          scanf("%d",&c1.Model) ;
          printf("\n Enter price of the Car in lakhs:");
          scanf("%f",&c1.price);
          printf("\n Car name is %s:",c1.name);
          printf("\n Car  Model  is %d:",c1.Model);
          printf("\n Car  price  is :%f lakhs",c1.price);
          getch();
   }
```

UNDERSTANDING THE COMPUTER'S MEMORY

- Every computer has a primary memory.
- All data and programs need to be placed in the primary memory for execution.
- RAM (Random Access Memory), which is a part of the primary memory, is a collection of memory locations (often known as cells) and each location has a specific address.
- Each memory location is capable of storing | byte of data (though new computers are able to store 2 bytes of data but in this book we have been talking about locations storing | byte of data).
- Therefore, a char type data needs just | memory location, an int type data needs 2 memory locations. Similarly, float and double type data need 4 and 8 memory locations, respectively.

In general, the computer has three areas of memory each of which is used for a specific task. These areas of memory include—

- ✓ **stack**
- ✓ **heap**
- ✓ **global memory**

➢ **Stack :**

- A fixed size of memory called system stack is allocated by the system and is filled as needed from the bottom to the top, one element at a time.
- These elements can be removed from the top to the bottom by removing one element at a time, i.e., the last clement added to the stack is removed first.
- When a program has used the variables or data stored in the stack, it can be discarded to enable the stack to be used by other programs to store their data.
- System stack is the section of memory that is allocated for automatic variables within functions.

➢ **Heap** :

- It is a contiguous block of memory that is available for use by programs when the need arises.
- A fixed size heap is allocated by the system and is used by the system in a random fashion.
- The addresses of the memory locations in heap that are not currently allocated to any program for use are stored in a free list.
- When a program requests a block of memory the dynamic allocation technique takes a block from the heap and assigns it to the program.
- When the program has finished using the block, it returns the memory block to the heap and the addresses of the memory locations in that block are added to the free list
- Compared to heaps, stacks are faster but smaller and expensive.

- When a program begins execution with the main() function, all variables declared within main() are allocated space on the stack.
- Moreover, all the parameters passed to a called function are stored on the stack.

- **Global memory** :
  - The block of code that is the main() program (along with other functions in the program) is stored in the global memory.
  - The memory in the global area is allocated randomly to store the code of different functions in the program in such a way that one function is not contiguous to another function.
  - Besides the function code, all global variables declared in the program are stored in the global memory area.
- **Other memory layouts** :
  - C provides some more memory areas such as text segment, BSS, and shared library segment.
  - The text segment is used to store the machine instructions corresponding to the compiled program. This is generally a read-only memory segment.
  - BSS (Block Started by Symbol) is used to store uninitialized global variables.
  - Shared library segment contains the executable image of shared libraries that are being used by the program.