UNIT-4

Arrays Functions and Strings in C

Asst. Prof. S.S Wadnere
Computer Engineering Department
SNJB KBJ College of Engineering, Chandwad

Contents: **(4 hours)**

Arrays in 'C': Concept, declaration, initialization, assessing elements, operations, multidimensional array

Functions in 'C': definition, function call, call by value and call by reference, return statement, standard library functions and user defined functions, passing array as function parameter.

Strings in 'C': Concept, declaration, initialization and string manipulation functions, library functions.

Motivation Array

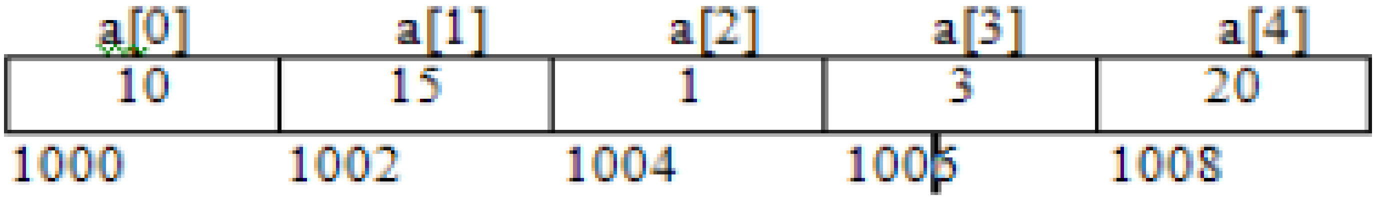
We wish to ***store*** percentage marks obtained by 100 students. In such a case we have two options to store these marks in memory:

1. Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student’s marks.
2. Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Introducing Arrays

An array

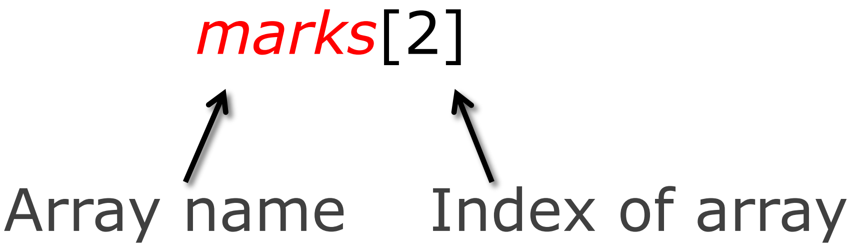
- a single name for a collection of data values
- all of the same data type
- All the data items of an array are stored in series of memory locations in RAM.



a[0]	a[1]	a[2]	a[3]	a[4]
10	15	1	3	20
1000	1002	1004	1006	1008

An *element* of an array is accessed using the **array name** and an **index or subscript.**

Ex.



marks[2]

Array name

Index of array

index always start with **0** and increment by **1**, so a[2] is the third element

The name of the array is the **address of the first** element and the subscript (index) is the **offset**

Definition and Initialization

- Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type.

An array is defined using a declaration statement.

Syntax

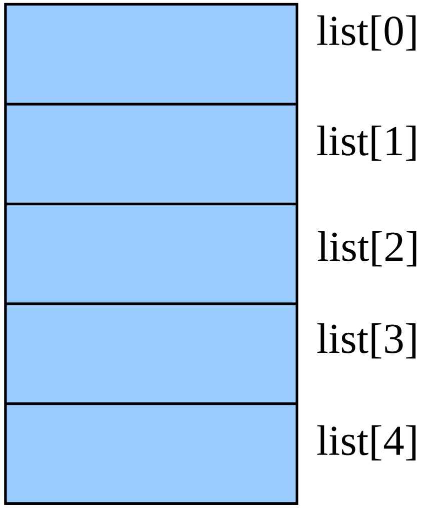
data_type array_name[size];

- allocates memory for size elements
- subscript of first element is 0
- subscript of last element is size-1
- size **must be a constant**

Example

```
int list[5];
```

Allocates memory for 5 integer variables



list[0]

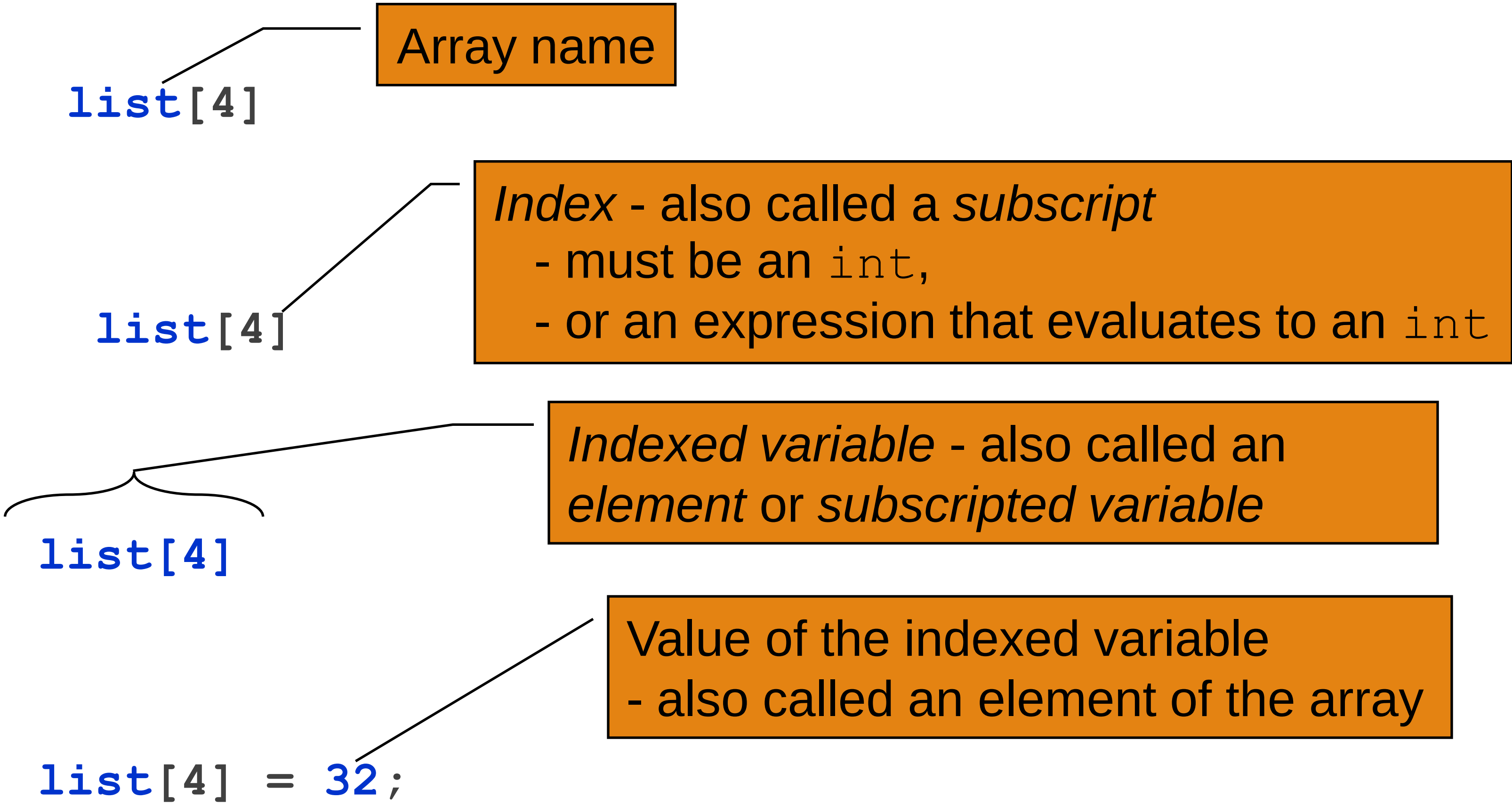
list[1]

list[2]

list[3]

list[4]

Some Array Terminology



```
list[4]
```

Array name

```
list[4]
```

Index - also called a *subscript*
- must be an `int`,
- or an expression that evaluates to an `int`

```
list[4]
```

Indexed variable - also called an *element* or *subscripted variable*

```
list[4] = 32;
```

Value of the indexed variable
- also called an element of the array

Initializing Arrays

Arrays can be initialized at the time they are declared.

***Examples*:**

```
double taxrate[3] ={0.15, 0.25, 0.3};

char list[5] = {‘h’, ’e’, ’l’, ’l’, ’o’};

double vector[100] = {0.0}; /*assigns  zero to all
                              100 elements */

int s[] = {5,0,-5};            /*the size of s is 3*/
```

Assigning values to an array

```
void main()
{
  int A[10];
  A[2]=1000;
  A[3]=1;
  scanf(“%d”, &A[5]);
}
```

A[0] = xxx
A[1] = xxx
A[2] = 1000
A[3] = 1
A[4] = xxx
A[5] = xxx
A[6] = xxx
A[7] = xxx
A[8] = xxx
A[9] = xxx

Assigning and reading values from an array

```
void main()
 {
    int A[10],i;
    for(i=0;i<10;i++)
    scanf(“%d”, &A[i]);

    for(i=0;i<10;i++)
    printf(“%d ”, A[i]);
 }
```

If the user enters 1 to 10. Output will be 1 2 3 4 5 6 7 8 9 10

A[0] = 1

A[1] = 2

A[2] = 3

A[3] = 4

A[4] = 5

A[5] = 6

A[6] = 7

A[7] = 8

A[8] = 9

A[9] = 10

Assigning and reading values from an array

```
void main()
{
 int A[10],sum=0,i;
  for(i=0;i<10;i++)
  scanf(“%d”, &A[i]);
  for(i=0;i<10;i++)
  printf(“%d ”, A[i]);
  for(i=0;i<10;i++)
  sum = sum + A[i];
}
Output:1 2 3 4 5 6 7 8 9 10
Sum = 55
```

A[0] = 1
A[1] = 2
A[2] = 3
A[3] = 4
A[4] = 5
A[5] = 6
A[6] = 7
A[7] = 8
A[8] = 9
A[9] = 10

Search Element in Array

```
int a[20], x, i;
for (i=0; i<20; i++)
        scanf(“%d”,&data[i]);
        printf("Enter elements to search: \n");
        scanf("%d", &x);

  for(i = 0; i < n; i++)
  {
       if(a[i]==x)
       {
           printf("%d is present at location %d\n",a[i],i+1);
           return 0;
       }
   }
 printf("%d is not present\n", x);
```

Multidimensional Arrays

Arrays with more than one index

- number of dimensions = number of indexes

general form:

Data type Array name[size1][size2]...[sizeN];

A 2-D array corresponds to a table or grid

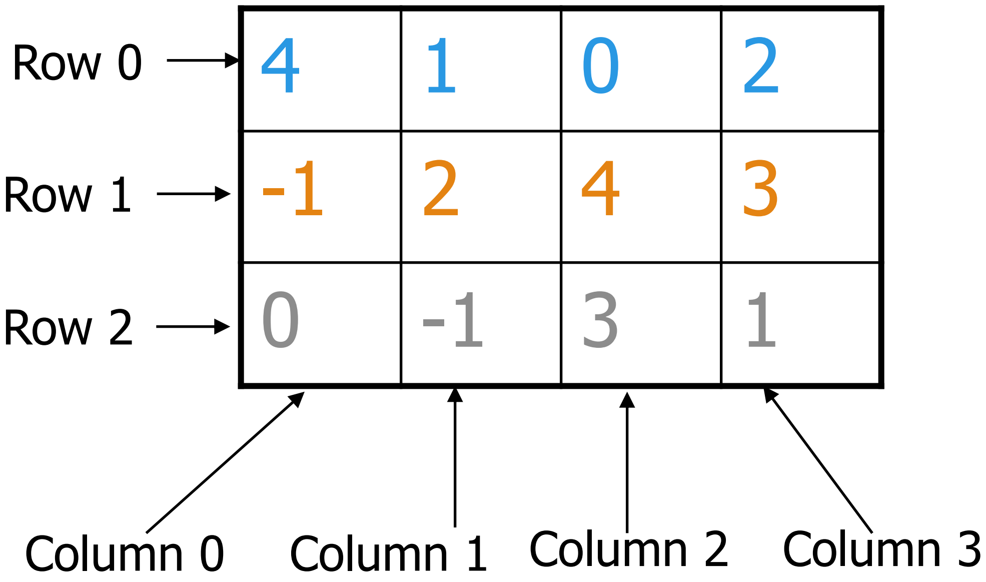
- One dimension is the **row**
- The other dimension is the **column**

Arrays with more than two dimensions are a simple extension of two-dimensional (2-D) arrays

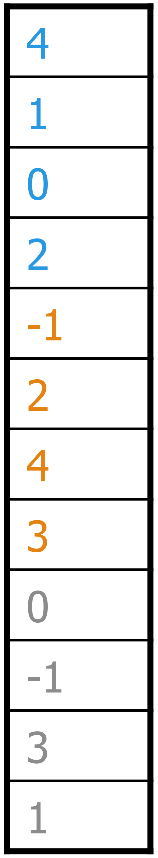
2 Dimensional-array

Set of numbers arranged with rows and columns.

```
int matrix[3][4];
```



	Column 0	Column 1	Column 2	Column 3
Row 0	4	1	0	2
Row 1	-1	2	4	3
Row 2	0	-1	3	1



4
1
0
2
-1
2
4
3
0
-1
3
1

in memory

Accessing Array Elements

```
int matrix[3][4];
```

- Row numbers are from 0 to 2
- Column numbers are from 0 to 3
- matrix has **12 integer** elements
- matrix[0][0] element in first row, first column
- matrix[2][3] element in last row, last column

Initialization

```
int x[4][4] = {   {2, 3, 7, 2},
                  {7, 4, 5, 9},
                  {5, 1, 6, -3},
                  {2, 5, -1, 3}};
int x[][4] = {    {2, 3, 7, 2},
                  {7, 4, 5, 9},
                  {5, 1, 6, -3},
                  {2, 5, -1, 3}};
```

Initialization(cont….)

```
int i, j, matrix[3][4];
for (i=0; i<3; i++)
  for (j=0; j<4; j++)
     matrix[i][j] = i;
```

j

i	0	1	2	3
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2

```
matrix[i][j] = j;
```

j

i	0	1	2	3
0	0	1	2	3
1	0	1	2	3
2	0	1	2	3

Max in 2D

Find the maximum of *int matrix[3][4]*

```
int max = matrix[0][0];
for (i=0; i<3; i++)
  for (j=0; j<4; j++)
    if (matrix[i][j] > max)
          max = matrix[i][j];
```

	0	1	2	3
0	0	1	0	2
1	-1	2	4	3
2	0	-1	3	1

Functions in C

A function is a group of statements that together perform a task. You can divide up your code into separate functions. logically the division is such that each function performs a specific task.

Motivation Behind Functions

1) Structured Programming
   - Construct a program from smaller components called **module**.
   - Each piece more **manageable** than the original program
   - To help make the program more understandable
2) Software reusability
   - Use existing functions as **building blocks** for new programs
3) Avoid code repetition

Motivation Behind Functions

```
 ata type function()
main()
{
Function( parameter passing);
}

Data type (parameters)
{
Return value
}
```

Function Definition

Function definition format

```
return-value-type  function-name( parameter-list )
{
   declarations and statements
}
```

Function-name: any valid identifier

Return-value-type: data type of the result (default **int**)
void – indicates that the function returns nothing

Parameter-list: comma separated list.

A type must be listed explicitly for each parameter unless, the parameter is of type **int**

Function Definition (continued..)

Declarations and statements: function body (block)

Variables can be declared inside blocks (can be nested)

Functions can not be defined inside other functions

Returning control

If nothing returned: **return** or until reaches right brace

If something returned: **return** *expression***;**

```
int absolute(int x)
{
    if (x >= 0)
    return x;
       else
    return -x;
}
```

Function calls

Syntax

function call

function-name**(arguments)**

Performs operations or manipulations returns results

Function call analogy:

Worker gets information, does task, returns result

```
int x,y=-10;
    x=absolute(y);
```

Flow of Control

- First statement executed in any program is the first statement in the function main( )
- When another function called
  - Control passed to first statement in that function’s body
  - program proceeds through sequence of statements within the function
- When last statement of function executed
  - control returns to where function was called
  - control given to next command after the call

Flow of Control

```
void main ( )
 {  . . .
   print_summary (rpt_total);
   revenue = rpt_total * .72675;
   . . .
 }

  void print_summary (int total)
   {  . . .
     cout << . . .
   }
```

- first statement of main
- function call, jumps to first statement of that function
- proceeds through function
- returns to next statement after call

Parameters

Function definition syntax:

```
float circleArea (float r)
{
        statements
}
```

Parameters in the declaration : formal parameters

Call (invocation of the function)

```
float area,radius=5.7;
area = circleArea (radius);
```

Parameters in the call: actual parameters

Passing Argument to Function

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

- ✓In C Programming we have different ways of parameter passing schemes such as
  1. Call by Value
  2. Call by Reference.

Call by Value

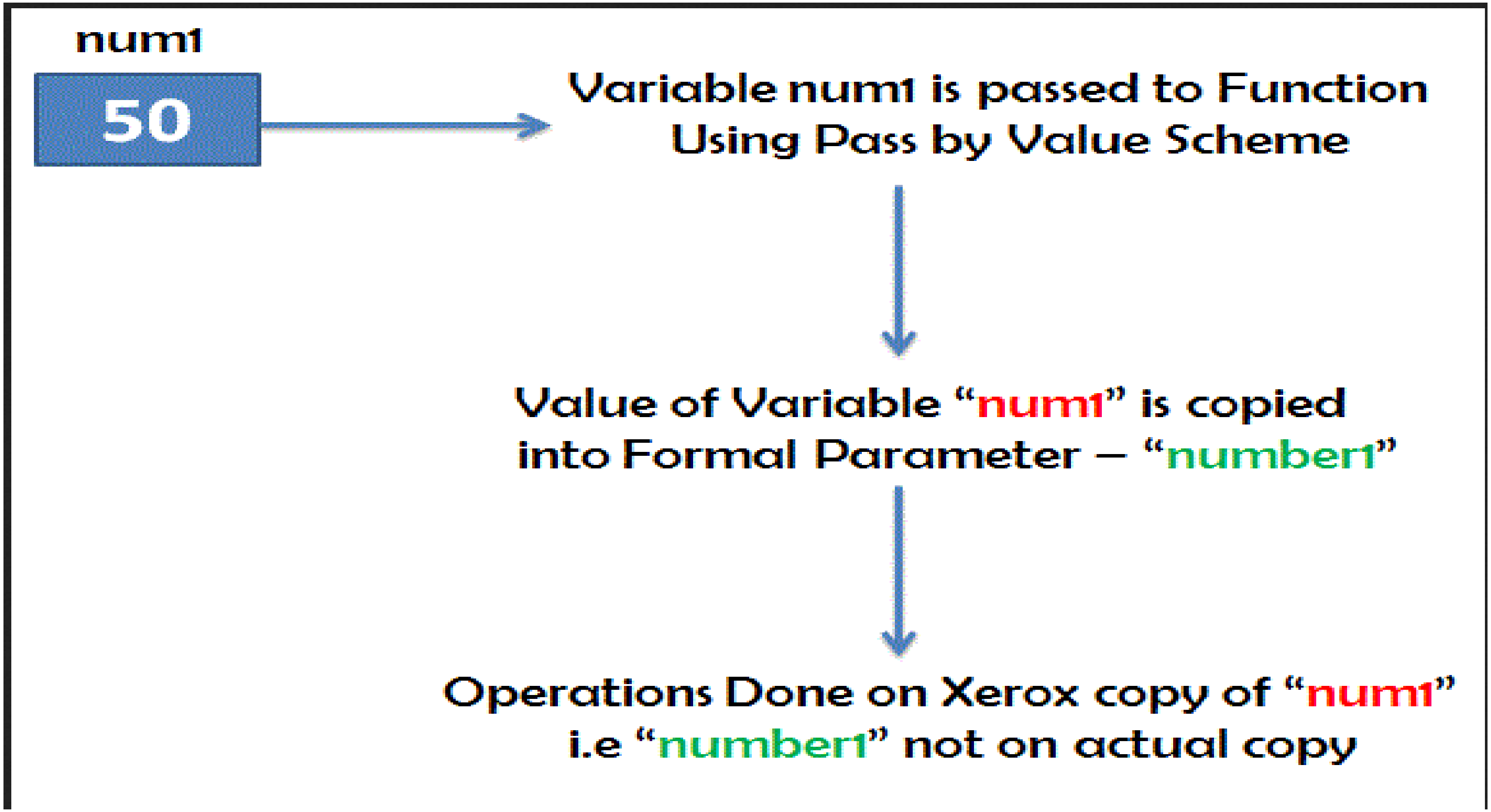
While Passing Parameters using call by value , **xerox copy of original parameter is created** and passed to the called function.

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

```
#include<stdio.h>
void interchange(int number1,int number2)
{
  int temp;
  temp = number1;
  number1 = number2;
  number2 = temp;
 }
int main() {
  int num1=50,num2=70;
  interchange(num1,num2);
  printf("\nNumber 1 : %d",num1);
  printf("\nNumber 2 : %d",num2);
  return(0);
}
```

Output : Number 1 : 50 Number 2 : 70

Call by Value



Variable num1 is passed to Function
Using Pass by Value Scheme

Value of Variable "num1" is copied
into Formal Parameter – "number1"

Operations Done on Xerox copy of "num1"
i.e "number1" not on actual copy

Call by Address

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

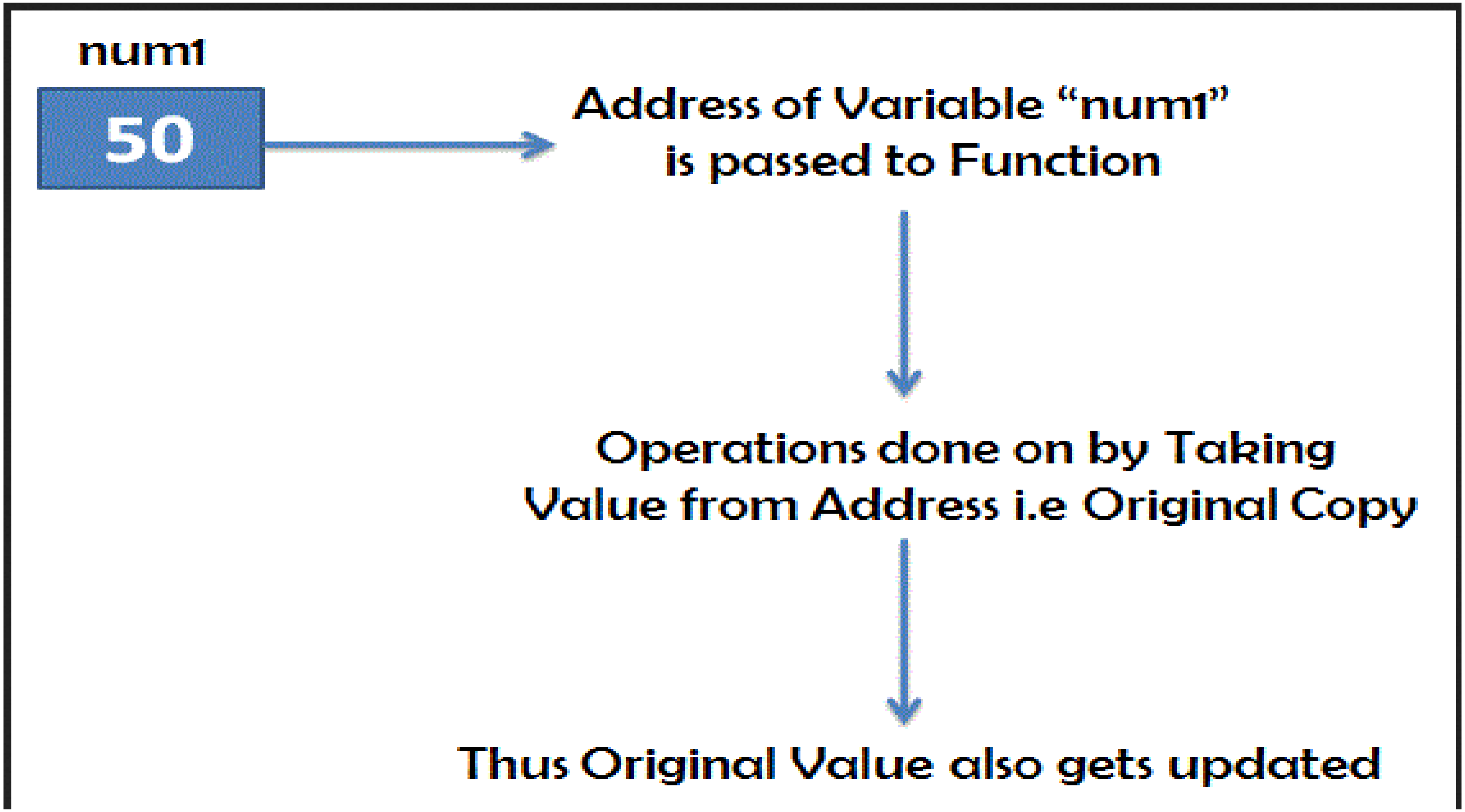
Any updates made inside the called function **will modify the original copy** since we are directly modifying the content of the exact memory location.

```
#include<stdio.h>
void interchange(int *num1,int *num2) {
int temp;
temp = *num1;
*num1 = *num2; *num2 = temp;
}

 int main() {
 int num1=50,num2=70;
interchange(&num1,&num2);
printf("\nNumber 1 : %d",num1);
printf("\nNumber 2 : %d",num2);
return(0);
}
```

Output : Number 1 : 70 Number 2 : 50

Call by Address



num1

50

Address of Variable "num1"
is passed to Function

Operations done on by Taking
Value from Address i.e Original Copy

Thus Original Value also gets updated

The return Statement

The **return** statement terminates the execution of a function and returns control to the calling function.

Execution resumes in the calling function at the point immediately following the call.

A **return** statement can also return a value to the calling function.Syntax:

```
    return expression;
```

Example

```
int main()
{
    int x = 25,y;
    y = sq( x );
    print_val( x, y );
    return 0;
}

int sq( int s ) {
    return( s * s );
}

void print_val( int a, int b) {
    printf( “a = %d, b = %d\n", a, b );
    return;
}
```

Types of Function

We have two type of functions :

1. **User-defined Function** : Function created by user. We just talked about user defined function in the beginning.

2. **Built-in Function** : Function created by C , and ready to use.

Standard (Predefined) Functions

Predefined functions

- Part of the C language
- Provided in function libraries

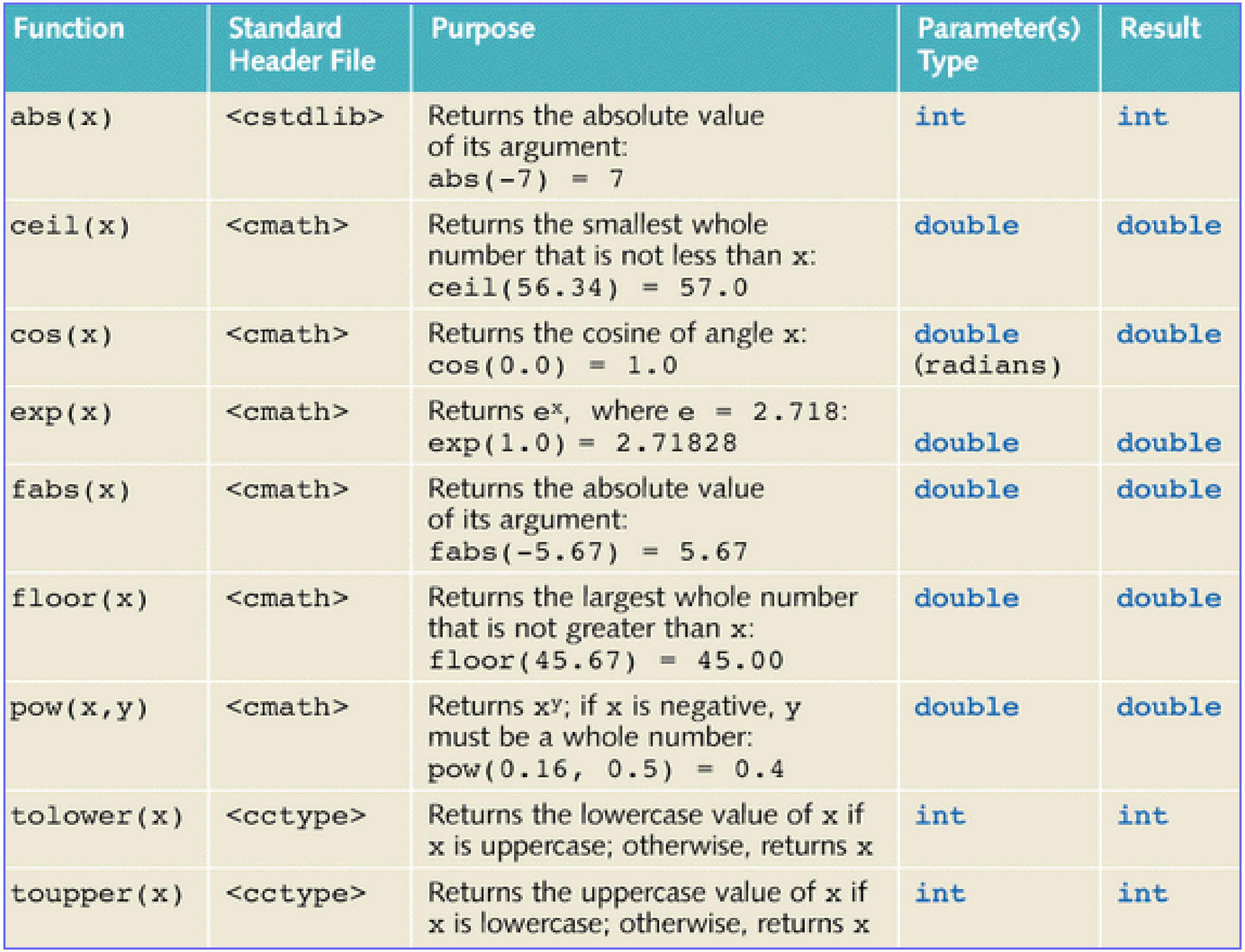
Make sure to use the required **`#include`** file

Examples:

abs(x), sin(x), log(x), pow(x, n)

Need to include “math.h” header file

Predefined Functions



Function	Standard Header File	Purpose	Parameter(s) Type	Result
abs(x)	<cstdlib>	Returns the absolute value of its argument: abs(-7) = 7	int	int
ceil(x)	<cmath>	Returns the smallest whole number that is not less than x: ceil(56.34) = 57.0	double	double
cos(x)	<cmath>	Returns the cosine of angle x: cos(0.0) = 1.0	double (radians)	double
exp(x)	<cmath>	Returns e^x, where e = 2.718: exp(1.0) = 2.71828	double	double
fabs(x)	<cmath>	Returns the absolute value of its argument: fabs(-5.67) = 5.67	double	double
floor(x)	<cmath>	Returns the largest whole number that is not greater than x: floor(45.67) = 45.00	double	double
pow(x,y)	<cmath>	Returns x^y; if x is negative, y must be a whole number: pow(0.16, 0.5) = 0.4	double	double
tolower(x)	<cctype>	Returns the lowercase value of x if x is uppercase; otherwise, returns x	int	int
toupper(x)	<cctype>	Returns the uppercase value of x if x is lowercase; otherwise, returns x	int	int

Passing Arrays as Function Parameter

- Arrays are always pass by **reference**
- Modifications to the array are reflected to main program
- **The array name is the *address* of the first element**
- The maximum *size* of the array must be specified at the time the array is declared.
- The *actual number* of array elements that are used will vary, so the ***actual size* of the array is usually passed as another argument to the function**

Example

```
main()
{
  int a[2]={3, 5};
  int c;
  c = sum_arr(a, 2)
}
int sum_arr(int b[], int n)
{
   int i, sum=0;
   for(i=0; i < n; i++)
        sum = sum + b[i];
   return(sum);
}
```

a[0]=3
a[1]=5
c=? 8
b=
n=2
i=0 1 2
sum=0 3 8

Strings in C

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

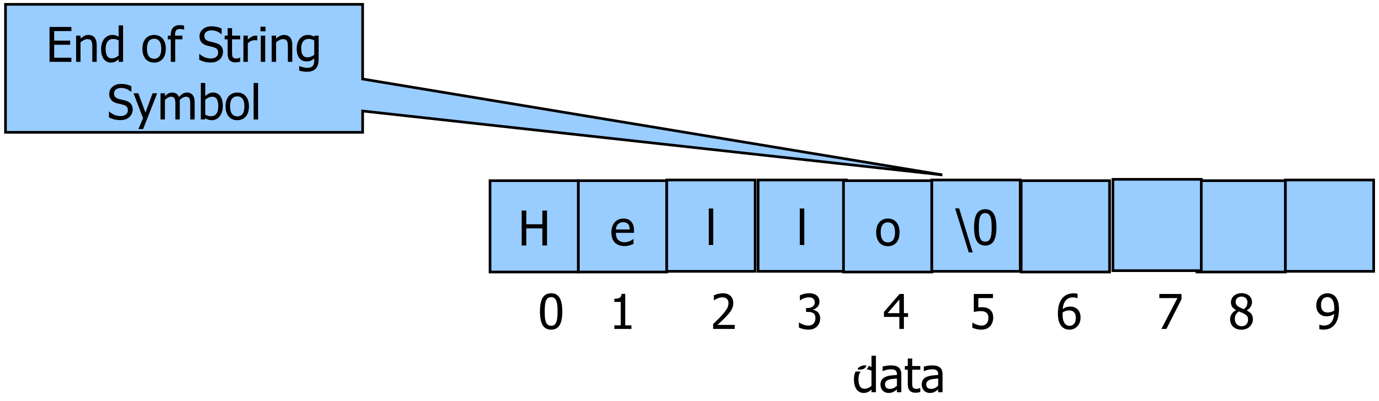
Strings

- □ No explicit type, strings are maintained as arrays of characters

```
char data[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; // or

char data[10] = “Hello”;
```

- □ End of string is indicated by a *delimiter*, the zero character ‘\0’
- □ String literal (string constant) - written in double quotes **"Hello**“



End of String Symbol

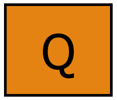
H	e	l	l	o	\0				
0	1	2	3	4	5	6	7	8	9

data

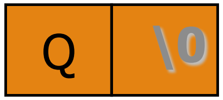
Distinction Between Characters and Strings

The representation of a char (e.g., ‘Q’) and a string (e.g., “Q”) is essentially different.

- A string is an array of characters ended with the null character.



Character ‘Q’



String “Q”

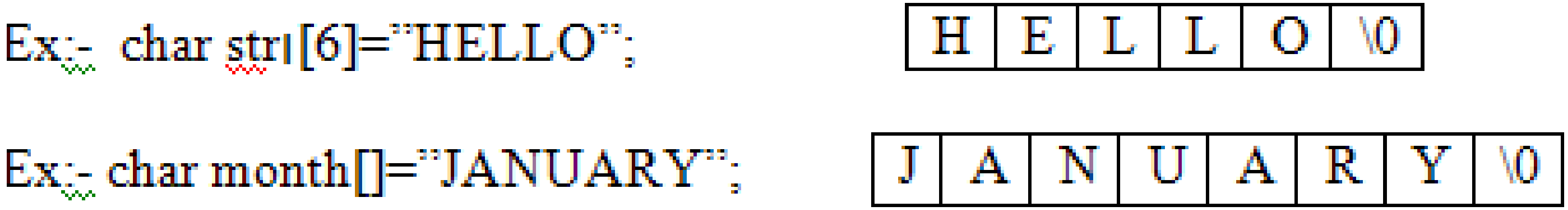
String Declaration

- ✓Declare as a character array
  - ✓**char color[] = "blue";**
- ✓Remember that strings represented as character arrays end with **'\0`**
- ✓**color** has **5** elements

String Initialization

C does not support string as a build in data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name.

Ex. char str4[6] = {‘H’,’e’,’l’,’l’,’o’,’\0’};



Ex:- char str1[6]="HELLO";

H	E	L	L	O	\0

Ex:- char month[]="JANUARY";

J	A	N	U	A	R	Y	\0

String Input from Terminal

(a) Formatted input function:-

scanf can be used with %s format

scanf("%s",name);

(b) Unformatted input functions:-

(1) **getchar()**:- It is used to read a single character from keyboard. Using this function repeatedly we may read entire line of text

Ex:- **ch=getchar();**

(2) **gets()**:- It is more convenient method of reading a string of text including blank spaces.

Ex:- **gets(line);**

Example

```
int main( ) {
char str[100];
int n,i;
printf( "Enter string :");
gets( str );   // OR
scanf(“%s”,str);  // OR
printf(“Enter no. of chars”);
scanf(“%d”,&n);
printf(“Enter string: “);
        for(i=0;i<n;i++)
        str[i]=getchar();
str[i]=‘\0’;
return 0;
}
```

Writing strings on to the screen

(a) Using formatted output functions

printf with %s format specifier we can print strings in different formats on to screen.

Ex:- char name[10];

printf(“%s”,name);

(b) Using unformatted output functions:-

1.**putchar()**:- It is used to print a character on the screen.

Ex:- putchar(ch);

2.puts():- It is used to print strings including blank spaces.

Ex:- char line[15]=”Welcome to lab”; puts(line);

Example

```
int main( ) {
char str[]=“Hello World”;
int i;

printf( “String is:");
puts( str );
```

OR

```
printf(“%s”,str);
        for(i=0;str[i]!=‘\0’;i++)
        putchar(str[i]);

return 0;
}
```

Exercise

Write a function to count the number of characters in a string.

Idea: count the number of characters before \0

H	e	l	l	o	\0				

Solution

```
void count_letters()
{
  int i=0;
  char mystring[10];

  printf(“Enter String(max size 9)”);
  scanf(“%s”,mystring);

  while (mystring[i] != '\0')
    i = i + 1;
  printf(“No of characters are:%d”,i);
}
```

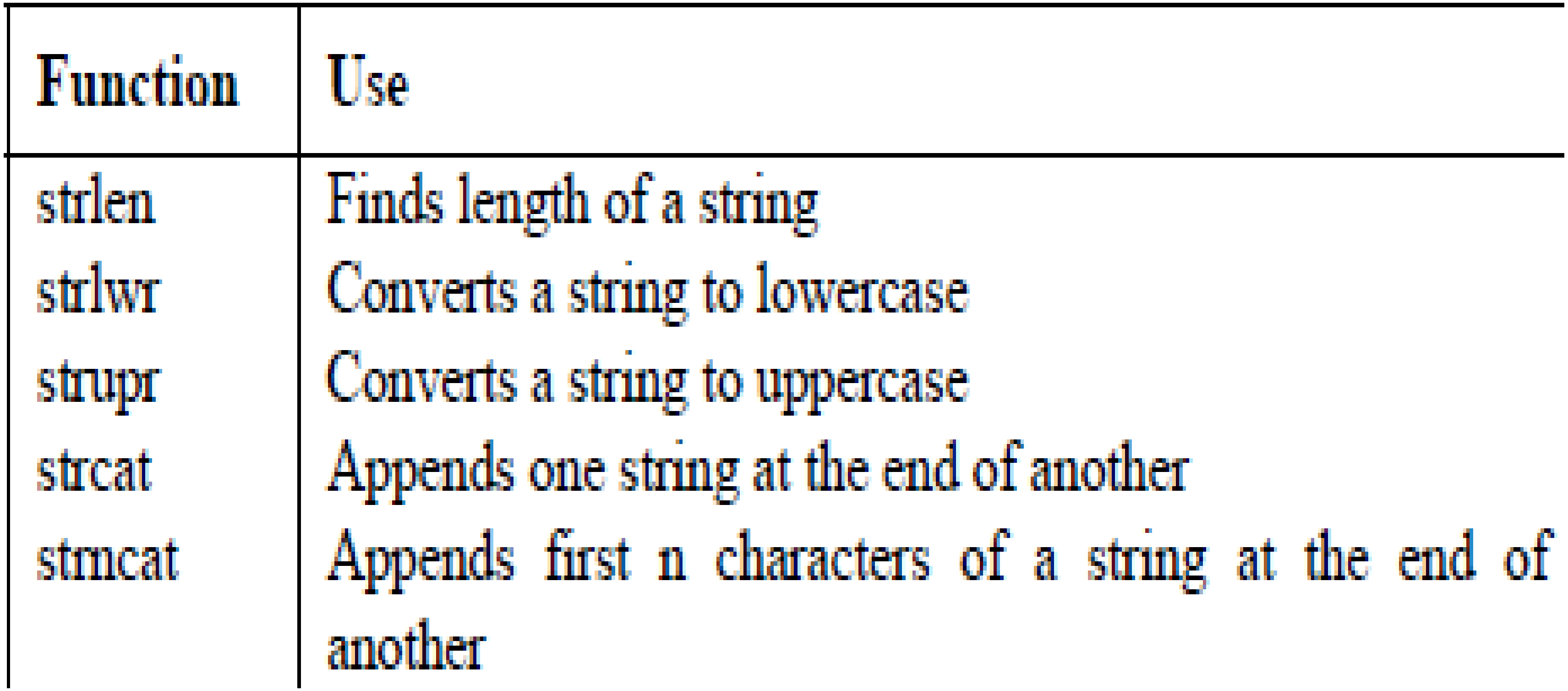
String Library Functions

The string can not be copied by the assignment operator ‘=’.

- e..g, str = “Test String”” is <u>not valid.</u>

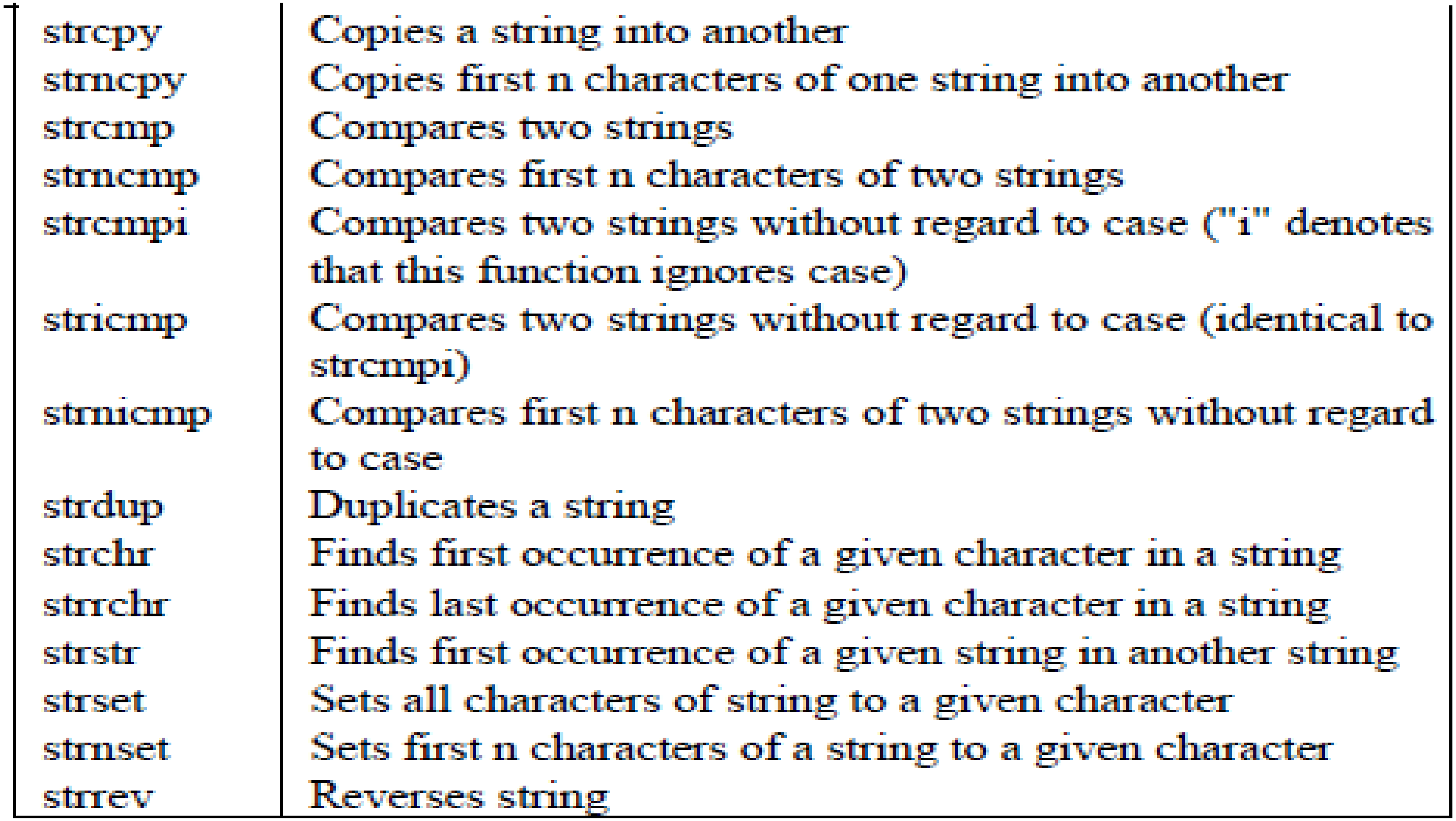
C provides string manipulating functions in the “**string.h**” library.

String Library Functions



Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another

String Library Functions



strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

strcpy

Strcpy (destinationstring, sourcestring)

Copies sourcestring into destinationstring

For example

```
strcpy(str, “hello world”);
```

assigns “hello world” to the string str

Example with strcpy

```
#include <stdio.h>
#include <string.h>
 main()
     {
     char x[] = “Example with strcpy”;
             char y[25];
             printf(“The string in array x is %s \n “, x);
             strcpy(y,x);
             printf(“The string in array y is %s \n “, y);
         }
```

O/P: The string in array y is Example with strcpy

strcat

strcat(destination, source)

OR

strcat(string1,string2)

appends source string to right hand side of destination string

For example if str had value “a big ”

strcat(str, “hello world”); appends “hello world” to the string “a big ” to get

“ a big hello world”

Example with strcat

```
#include <stdio.h>
#include <string.h>
 main(){
     char x[] = “Example with strcat”;
       char y[]= “which stands for string concatenation”;
          printf(“The string in array x is %s \n “, x);
          strcat(x,y);
           printf(“The string in array x is %s \n “, x);
            }
```

```
O/P: The string in array x is Example with strcat
            The string in array x is Example with strcatwhich stands
        for string concatenation
```

strcmp

strcmp(stringa, stringb)

- Compares stringa and stringb alphabetically
- The characters are compared against the ASCII table.
- “thrill” < “throw” since ‘i’ < ‘o’;
- “joy” < joyous“;
- Note lowercase characters are greater than Uppercase

Relationship	Returned Value	Example
`str1 < str2`	Negative	“Hello”< “Hi”
`str1 = str2`	0	“Hi” = “Hi”
`str1 > str2`	Positive	“Hi” > “Hello”

Example with strcmp

```
#include <stdio.h>
#include <string.h>
 main()
     {
           char x[] = “cat”;
           char y[]= “cat”;
            if (strcmp(x,y) == 0)
            printf(“The string in array x  %s is equal to
 that in %s \n “, x,y);
```

strlen

strlen(str) returns length of string excluding null character

strlen(“tttt”) = 4 not 5 since \0 not counted

Example with strlen

```
#include <stdio.h>
#include <string.h>
 main()
    {
      int i, count;
    char x[] = “tommy tucket took a tiny ticket ”;
       count = 0;
      for (i = 0; i < strlen(x);i++)
         {
           if (x[i] == ‘t’) count++;
          }
     printf(“The number of t’s in   %s is %d \n “, x,count);
    }
```