# Session 13 Pointers

## Objectives

**At the end of this session, you will be able to:**

- *Explain what a pointer is and where it is used*
- *Explain how to use pointer variables and pointer operators*
- *Assign values to pointers*
- *Explain pointer arithmetic*
- *Explain pointer comparisons*
- *Explain how pointers can be passed as arguments to functions*
- *Explain pointers and single dimensional arrays*
- *Explain Pointer and multidimensional arrays*
- *Explain how allocation of memory takes place*

## Introduction

A pointer provides a way of accessing a variable without referring to the variable directly. It provides a symbolic way of using addresses. This session deals with the concept of pointer and its usage in C. Also, we shall discuss a few concepts which are associated with pointers.

## 13.1 What is a Pointer?

A **Pointer** is a variable, which contains the address of a memory location of another variable, rather than the stored value of that variable. If one variable contains the address of another variable, the first variable is said to **point** to the second. A pointer provides an indirect way of accessing the value of a data item. Consider two variables **`var1`** and **`var2`** such that **`var1`** has a value of `500` and is stored in the memory location `1000`. If **`var2`** is declared as a pointer to the variable **`var1`**, the representation will be as follows:

| Memory location | Value stored | Variable name |
|---|---|---|
| 1000 | 500 | var1 |
| 1001 | | |
| 1002 | | |
| . | | |
| . | | |
| 1108 | 1000 | var2 |

Here, **`var2`** contains the value `1000`, which is nothing but the address of the variable **`var1`**.

Pointers can point to variables of other fundamental data type variables like `int`, `char`, or `double` or data aggregates like `arrays`.

## 13.1.1 Why are Pointers used?

Few of the situations where pointers can be used are:

- To return more than one value from a function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them (or to parts of them), instead of moving the arrays themselves
- To allocate memory and access it (dynamic memory allocation)

## 13.2 Pointer Variables

If a variable is to be used as a pointer, it must be declared. A pointer declaration consists of a base type, an `*`, and a variable name. The general syntax for declaring a pointer variable is:

```
type *name;
```

where `type` is any valid data type, and `name` is the name of the pointer variable. The declaration tells the compiler that `name` is used to store the address of a value corresponding to the data type `type`. In the declaration statement, `*` indicates that a pointer variable is being declared.

In the above example of **`var1`** and **`var2`**, since **var2** is a pointer, which holds the value of an `int` type variable **`var1`**, it will be declared as:

```
int *var2;
```

Now, **var2** can be used in a program to indirectly access the value of **var1**. Remember, **var2** is not of type `int` but is a pointer to a variable of type `int`.

The base type of the pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point anywhere in the memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly.

## 13.3 The Pointer Operators

There are two special operators which are used with pointers: `*` and `&`. The `&` operator is a unary operator and it returns the memory address of the operand. For example,

```
var2 = &var1;
```

places the memory address of **var1** into **var2**. This address is the computer's internal location of the variable **var1** and has nothing to do with the value of **var1**. The `&` operator can be thought of as returning "the address of". Therefore, the above assignment means "**var2** receives the address of **var1**". Referring back, the value of **var1** is `500` and it uses the memory location `1000` to store this value. After the above assignment, **var2** will have the value `1000`.

The second pointer operator, `*`, is the complement of `&`. It is a unary operator and returns the value contained in the memory location pointed to by the pointer variable's value.

Consider the previous example, where **var1** has the value `500` and is stored in the memory location `1000`, after the statement

```
var2 = &var1;
```

**var1** contains the value `1000`, and after the assignment

```
temp = *var2;
```

**temp** will contain `500` and not `1000`. The `*` operator can be thought of as "at the address".

Both `*` and `&` have a higher precedence than all other arithmetic operators except the unary minus. They share the same precedence as the unary minus.

The following program prints the value of an integer variable, its address, which is stored in a pointer variable, and also the address of the pointer variable.

**Example 1:**

```
#include <stdio.h>
void main()
{
    int var = 500, *ptr_var;
    /* var is declared as an integer and ptr_var as a pointer pointing to an
integer */
    ptr_var = &var; /*stores address of var in ptr_var*/
    /*Prints value of variable (var) and address where var is stored */
    printf("The value %d is stored at address %u:",var,&var);
   /*Prints value stored in ptr variable (ptr_var) and address where ptr_var
is stored */
    printf("\nThe value %u is stored at address: %u", ptr_var, &ptr_var);
    /* Prints value of variable (var) and address where var is stored, using
pointer to variable */
    printf("\nThe value %d is stored at address:%u", *ptr_var, ptr_var);
}
```

A sample output for the above will be:

```
The value 500 is stored at address: 65500
The value 65500 is stored at address: 65502
The value 500 is stored at address: 65500
```

In the above, **`ptr_var`** contains the address 65500, which is a memory location where the value of **`var`** is stored. The contents of this memory location (65500) can be obtained by using `*`, as **`*ptr_var`**. Now **`*ptr_var`** represents the value 500, which is the value of **`var`**. Since **`ptr_var`** is also a variable, its address can be printed by prefixing it with `&`. In the above case, **`ptr_var`** is stored at location 65502. The `%u` conversion specifier prints the arguments as unsigned integers.

Recollect that an integer occupies 2 bytes of memory. Hence, value of **`var`** is stored at 65500 and the compiler allots the next memory allocation 65502 to **`ptr_var`**. Similarly, a floating point number will require four bytes and a double precision number may require eight bytes. Pointer variables store an integer value. For most programs using pointers, pointer types can be considered to be 16-bit values that occupy 2 bytes.

Note that the following two statements give the same output.

```
printf("The value is %d", var);
printf("The value is %d", *(&var));
```

## Assigning Values to Pointers

Values can be assigned to pointers through the `&` operator. The assignment statement will be:

```
ptr_var = &var;
```

where the address of **`var`** is stored in the variable **`ptr_var`**. It is also possible to assign values to pointers through another pointer variable pointing to a data item of the same type.

```
ptr_var = &var;
ptr_var2 = ptr_var;
```

A NULL value can also be assigned to a pointer using zero as follows:

```
ptr_var = 0;
```

Variables can be assigned value through their pointers as well.

```
*ptr_var = 10;
```

will assign `10` to the variable **`var`** if **`ptr_var`** points to **`var`**.

In general, expressions involving pointers follow the same rules as other C expressions. It is very important to assign values to pointer variables before using them; else they could be pointing to any unpredictable values.

## Pointer Arithmetic

Addition and subtraction are the only operations, which can be performed on pointers. The following example demonstrates this:

```
int var, *ptr_var;
ptr_var = &var;
var = 500;
```

In the above example, let us assume that **`var`** is stored at the address `1000`. Then, **`ptr_var`** has the value `1000` stored in it. Since integers are 2 bytes long, after the expression:

```
ptr_var++ ;
```

**`ptr_var`** will contain `1002` and NOT `1001`. This means that **`ptr_var`** is now pointing to the integer stored at the address `1002`. Each time **`ptr_var`** is incremented, it will point to the next integer and since integers are 2 bytes long, **ptr_var** will be incremented by 2. The same is true for decrements also.

Here are few more examples.

| | |
|---|---|
| `++ptr_var` or `ptr_var++` | points to next integer after **var** |
| `--ptr_var` or `ptr_var--` | points to integer previous to **var** |
| `ptr_var + i` | points to the $i^{th}$ integer after **var** |
| `ptr_var – i` | points to the $i^{th}$ integer before **var** |
| `++*ptr_var` or `(*ptr_var)++` | Will increment **var** by 1 |
| `*ptr_var++` | Will fetch the value of the next integer after **var** |

Each time a pointer is incremented, it points to the memory location of the next element of its base type. Each time it is decremented, it points to the location of the previous element. With pointers to characters, this appears normal, because generally characters occupy 1 byte per character. However, all other pointers will increase or decrease depending on the length of the data type they are pointing to.

As seen in the above examples, in addition to the increment and decrement operators, integers can be added and subtracted to or from pointers. Besides addition and subtraction of a pointer and an integer, none of the other arithmetic operations can be performed on pointers. To be specific, pointers cannot be multiplied or divided. Also, float or double type cannot be added or subtracted to or from pointers.

**Pointer Comparisons**

Two pointers can be compared in a relational expression. However, this is possible only if both these variables are pointing to variables of the same type. Consider that **ptr_a** and **ptr_b** are two pointer variables, which point to data elements **a** and **b**. In this case, the following comparisons are possible:

| | |
|---|---|
| `ptr_a < ptr_b` | Returns true provided **a** is stored before **b** |
| `ptr_a > ptr_b` | Returns true provided **a** is stored after **b** |
| `ptr_a <= ptr_b` | Returns true provided **a** is stored before **b** or **ptr_a** and **ptr_b** point to the same location |
| `ptr_a >= ptr_b` | Returns true provided **a** is stored after **b** or **ptr_a** and **ptr_b** point to the same location |
| `ptr_a == ptr_b` | Returns true provided both pointers **ptr_a** and **ptr_b** points to the same data element |
| `ptr_a != ptr_b` | Returns true provided both pointers **ptr_a** and **ptr_b** point to different data elements but of the same type |
| `ptr_a == NULL` | Returns true if **ptr_a** is assigned NULL value (zero) |

Also, if **ptr_begin** and **ptr_end** point to members of the same array then,

```
ptr_end - ptr_begin
```

will give the difference in bytes between the storage locations to which they point.

## 13.4 Pointers and Single-dimensional Arrays

An array name is truly a pointer to the first element in that array. Therefore, if **`ary`** is a single-dimensional array, the address of the first array element can be expressed as either **`&ary[0]`** or simply as **`ary`**. Similarly, the address of the second array element can be written as **`&ary[1]`** or as **`ary+1`**, and so on. In general, the address of the `(i + 1)` th array element can be expressed as either **`& ary[i]`** or as **`(ary + i)`**. Thus, the address of an array element can be expressed in two ways:

- By writing the actual array element preceded by the ampersand sign (&)

- By writing an expression in which the subscript is added to the array name

Remember that in the expression **`(ary + i), ary`** represents an address, whereas **`i`** represents an integer quantity. Moreover, **`ary`** is the name of an array whose elements can be both integers, characters, floating point, and so on (of course, all elements have to be of the same type). Therefore, the above expression is not a mere addition; it is actually specifying an address, which is a certain number of memory cells beyond the first. The expression **`(ary + i)`** is in true sense, a symbolic representation for an address specification rather than an arithmetic expression.

As said before, the number of memory cells associated with an array element will depend on the data type of the array as well as the computer's architecture. However, the programmer can specify only the address of the first array element that is the name of the array (**`ary`** in this case) and the number of array elements beyond the first, that is, a value for the subscript. The value of **`i`** is sometimes referred to as an **offset** when used in this manner.

The expressions **`&ary[i]`** and **`(ary + i)`** both represent the address of the ith element of ary, and so it is only logical that **`ary[i]`** and **`*(ary + i)`** both represent the contents of that address, that is, the value of the **`i`** th element of **`ary`**. Both terms are interchangeable and can be used in any particular application as desired by the programmer.

The following program shows the relationship between array elements and their addresses.

**Example 2:**

```
#include<stdio.h>
void main()
{
    static int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    for (i = 0; i < 10; i ++)
    {
        printf("\n i = %d , ary[i] = %d , *(ary+i)= %d ", i, ary[i], *(ary
+ i));
        printf("&ary[i] = %X , ary + i = %X", &ary[i], ary+i);
        /* %X gives unsigned hexadecimal */
    }
}
```

The above program defines a single dimensional, 10-element integer array **ary**, whose elements are assigned the values `1`, `2`, `..10`. The `for` loop is used to display the value and the corresponding address of each array element. Note that the value of each element is specified in two different ways, as **`ary[i]`** and as **`*(ary + i)`**, to illustrate their equivalence. Similarly, the address of each array element is also displayed in two ways. The output of the program will be as follows:

```
i=0 ary[i]=1  *(ary+i)=1  &ary[i]=194  ary+i = 194
i=1 ary[i]=2  *(ary+i)=2  &ary[i]=196  ary+i = 196
i=2 ary[i]=3  *(ary+i)=3  &ary[i]=198  ary+i = 198
i=3 ary[i]=4  *(ary+i)=4  &ary[i]=19A  ary+i = 19A
i=4 ary[i]=5  *(ary+i)=5  &ary[i]=19C  ary+i = 19C
i=5 ary[i]=6  *(ary+i)=6  &ary[i]=19E  ary+i = 19E
i=6 ary[i]=7  *(ary+i)=7  &ary[i]=1A0  ary+i = 1A0
i=7 ary[i]=8  *(ary+i)=8  &ary[i]=1A2  ary+i = 1A2
i=8 ary[i]=9  *(ary+i)=9  &ary[i]=1A4  ary+i = 1A4
i=9 ary[i]=10 *(ary+i)=10 &ary[i]=1A6  ary+i = 1A6
```

This output clearly shows the difference between **`ary[i]`**, which represents the value of the **`i`**th array element, and **`&ary[i]`**, which represents its address.

When assigning a value to an array element such as **`ary[i]`**, the left side of the assignment statement can be written as either **`ary[i]`** or as **`*(ary + i)`**. Thus, a value may be assigned directly to an array element or it may be assigned to the memory area whose address is that of the array element. It is sometimes necessary to assign an address to an identifier. In such situations, a pointer must appear on the left side of the assignment statement. It is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as **`ary, (ary + i)`** and **`&ary[i]`** cannot appear on the left side of an assignment statement. Moreover, the address of an array cannot be arbitrarily altered, so expressions such as **`ary++`** are not allowed. The reason for this is: **ary** is the address of the array **`ary`**. When the array is declared, the linker has decided where this array will go, for example, say an address `1002`. Once it is given this address, it stays there. Trying to increment this address has no meaning, its like saying

```
x = 5++;
```

Since a constant cannot be incremented, the compiler will flag an error.

In case of the array **ary**, **ary** is also known as a **Pointer Constant**. Remember, **(ary + 1)** does not move the array **ary** to the **(ary + 1)** th position, it just points to that position, whereas **ary++** actually tries to move **ary** by 1 position.

The address of one element cannot be assigned to some other array element, though the value of one array element can be assigned to another through pointers.

```
&ary[2] =  &ary[3];   /* not allowed */
ary[2]  =  ary[3];    /* allowed */
```

Recall that the `scanf()` function required that variables of the basic data types be preceded by ampersands (`&`), whereas array names were exempted from this requirement. This will be easy to understand now. The `scanf()` requires that the address of the data items, being entered into the computer's memory, be specified. As said before, the ampersand (`&`) actually gives the address of the variable and so it is required that an ampersand precede a single valued variable. Ampersands are not required with array names because array names themselves represent addresses. However, if a single element of an array is to be read, it will require an ampersand to precede it.

```
scanf("d", *ary) /* For first element of array */
scanf("%d", &ary[2]) /* For an array element */
```

## 13.4.1 Pointers and Multidimensional Arrays

The way a single dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript), a multidimensional array can also be represented with an equivalent pointer notation. This is because a multidimensional array is actually a collection of single dimensional arrays. For example, a two-dimensional array can be defined as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array declaration can be written as:

```
data_type (*ptr_var)[expr 2];
```

instead of

```
data_type array[expr 1][expr 2];
```

This concept can be generalized to higher dimensional arrays, that is,

```
data_type (*ptr_var)[exp 2] .... [exp N];
```

can be written instead of

```
data_type array[exp 1][exp 2] ... [exp N];
```

In these declarations, **data_type** refers to the data type of the array, **ptr_var** is the name of the pointer variable, array is the corresponding array name, and **exp 1**, **exp 2**, **exp 3**, ... **exp N** are positive valued integer expressions that indicate the maximum number of array elements associated with each subscript.

Note the parentheses that surround the array name and the preceding asterisk in the pointer version of each declaration. These parentheses must be present; else the definition would represent an array of pointers rather than a pointer to a group of arrays.

For example, if **ary** is a two dimensional array having 10 rows and 20 columns, it can be declared as

```
int (*ary)[20];
```

instead of

```
int ary[10][20];
```

In the first definition, **ary** is defined to be a pointer to a group of contiguous, single-dimensional, 20-element integer arrays. Thus, **ary** points to the first element of the array, which is actually the first row (row 0) of the original two-dimensional array. Similarly, **(ary + 1)** points to the second row of the original two-dimensional array, and so on.

A three-dimensional floating-point array **fl_ary** can be defined as:

```
float (*fl_ary)[20][30];
```

rather than

```
float fl_ary[10][20][30];
```

In the first declaration, **fl_ary** is defined as a group of contiguous, two-dimensional, 20 x 30 floating point arrays. Hence, **fl_ary** points to the first 20 x 30 array, **(fl_ary + 1)** points to the second 20 x 30 array, and so on.

In the two-dimensional array **ary**, the item in row 4 and column 9 can be accessed using the statement:

```
ary[3][8];
```

or

```
*(*(ary + 3) + 8);
```

The first form is the usual way in which an array is referred to. In the second form, **(ary + 3)** is a pointer to the row 4. Therefore, the object of this pointer, **`*(ary + 3)`**, refers to the entire row. Since row 3 is a one-dimensional array, **`*(ary + 3)`** is actually a pointer to the first element in row 3, `8` is then added to this pointer. Hence, **`*(*(ary + 3) + 8)`** is a pointer to element 8 (the 9th element) in row 4. The object of this pointer, **`*(*(ary + 3) + 8)`**, therefore refers to the item in column 9 of row 4, which is **`ary [3][8]`**.

There are different ways to define arrays, and different ways to process the individual array elements. The choice of one method over another generally depends on the user's preference. However, in applications involving numerical arrays, it is often easier to define the arrays in the conventional manner.

**Pointers and Strings**

Strings are nothing but single-dimensional arrays, and as arrays and pointers are closely related, it is only natural that strings too will be closely related to pointers. Consider the case of the function `strchr()`. This function takes as arguments a string and a character to be searched for in that string, that is,

```
ptr_str = strchr(strl, 'a');
```

the pointer variable **`ptr_str`** will be assigned the address of the first occurrence of the character '`a`' in the string **`str`**. This is not the position in the string, from 0 to the end of the string but the address, from where the string starts to the end of the string.

The following program uses `strchr()` in a program which allows a user to enter a string and a character to be searched for. The program prints out the address of the start of the string, the address of the character, and the character's position relative to the start of the string (0 if it is the first character, 1 if it is the second and so on). This relative position is the difference between the two addresses, the address of start of the string and the address where the character's first occurrence is found.

**Example 3:**

```
#include <stdio.h>
#include <string.h>
void main ()
{
    char a, str[81], *ptr;
    printf("\nEnter a sentence:");
    gets(str);
    printf("\nEnter character to search for:");
    a = getche();
    ptr = strchr(str,a);
    /* return pointer to char */
    printf( "\nString starts at address: %u",str);
    printf("\nFirst occurrence of the character is at address: %u", ptr);
    printf("\n Position of first occurrence (starting from 0)is: %d", ptr-
str);
}
```

A sample run will be as follows:

```
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address: 65437.
Position of first occurrence (starting from 0) is: 17
```

In the declaration statement, a pointer variable **`ptr`** is set aside to hold the address returned by `strchr()`, since this is an address of a character (**`ptr`** is of type `char`).

The function `strchr()` does not need to be declared if the include file `string.h` is included.

## 13.5 Allocating Memory

Till this point of time it has been established that an array name is actually a pointer to the first element of the array. Also, it is possible to define the array as a pointer variable rather than the conventional array. However, if an array is declared conventionally, it results in a fixed block of memory being reserved at the beginning of the program execution, whereas this does not occur if the array is represented as a pointer variable. As a result, the use of a pointer variable to represent an array requires some sort of initial memory assignment before the array elements are processed. Such memory allocations are generally done using the `malloc()` library function.

Consider an example. A single dimensional integer array **`ary`** having 20 elements can be defined as:

```
int *ary;
```

instead of

```
int ary[20];
```

However, **`ary`** will not be automatically assigned a memory block when it is defined as a pointer variable, though a block of memory enough to store 10 integer quantities will be reserved in advance if **`ary`** is defined as an array. If **`ary`** is defined as a pointer, sufficient memory can be assigned as follows:

```
ary = malloc(20 *sizeof(int));
```

This will reserve a block of memory whose size (in bytes) is equivalent to the size of an integer. Here, a block of memory for 20 integers is allocated. The number 20 assigns 20 bytes (one for each integer) and this is multiplied by `sizeof(int)`, which will return 2, if the computer uses 2 bytes to store an integer. If a computer uses 1 byte to store an integer, the `sizeof()` function is not required. However, it is preferable to use this always as it facilitates the portability of code. The function `malloc()` returns a pointer which is the address location of the starting point of the memory allocated. If enough memory space does not exist, `malloc()` returns a NULL. The allocation of memory in this manner, that is, **as and when required in a program** is known as **Dynamic memory allocation.**

Before proceeding further, let us discuss the concept of **Dynamic Memory allocation**. A C program can store information in the main memory of the computer in two primary ways. The first method involves global and local variables – including arrays. In the case of global and static variables, the storage is fixed throughout the program's run time. These variables require that the programmer knows the amount of memory needed for every situation in advance. The second way in which information can be stored is through C's **Dynamic Allocation System**. In this method, storage for information is allocated from the pool of free memory as and when needed.

The `malloc()` function is one of the most commonly used functions which permit allocation of memory from the pool of free memory. The parameter for `malloc()` is an integer that specifies the number of bytes needed.

As another example, consider a two-dimensional character array **`ch_ary`** having 10 rows and 20 columns. The definition and allocation of memory in this case would be as follows:

```
char (*ch_ary)[20];
ch_ary = (char*)malloc(10*20*sizeof(char));
```

As said earlier, `malloc()` returns a pointer to type void. However, since **`ch_ary`** is a pointer to type `char`, type casting is necessary. In the above statement, `(char*)` casts `malloc()` so as to return a pointer to type `char`.

However, if the declaration of array has to include the assignment of initial values then an array has to be defined in the conventional manner rather than as a pointer variable as in:

```
int ary[10] = {1,2,3,4,5,6,7,8,9,10};
```

or

```
int ary[] = {1,2,3,4,5,6,7,8,9,10};
```

The following example creates a single dimensional array dynamically and sorts the array in ascending order. It uses pointers and the `malloc()` function to assign memory.

**Example 4:**

```
#include<stdio.h>
#include<malloc.h>
void main()
{
    int *p,n,i,j,temp;
    printf("\n Enter number of elements in the array :");
    scanf("%d",&n);
    p=(int*)malloc(n*sizeof(int));
    for(i=0;i<n;++i)
    {
        printf("\nEnter element no. %d:",i+1);
        scanf("%d",p+i);
    }
    for(i=0;i<n-1;++i)
        for(j=i+1;j<n;++j)
            if(*(p+i)>*(p+j))
            {
                temp=*(p+i);
                *(p+i)=*(p+j);
                *(p+j)=temp;
            }
    for(i=0;i<n;++i)
        printf("%d\n",*(p+i));
}
```

Note the `malloc()` statement,

```
p = (int*)malloc(n*sizeof(int));
```

Here, `p` is declared as a pointer to an array and assigned an amount of memory using `malloc()`.

Data is read, using `scanf()`.

```
scanf("%d",p+i);
```

In `scanf()`, the pointer variable is used to store data into the array.

Sorted array elements are displayed using `printf()`.

```
printf("%d\n",*(p+i));
```

Note the asterisk in this case. This is because the value stored in that particular location has to be displayed. Without the asterisk, the `printf()` will display the address where the marks are stored and not the marks stored.

➢ **free()**

This function can be used to de-allocate (frees) memory when it is no longer needed.

The general format of `free()` function:

```
voidfree( void *ptr );
```

The `free()` function de-allocates the space pointed to by **`ptr`**, freeing it up for future use. **`ptr`** must have been used in a previous call to `malloc()`, `calloc()`, or `realloc()`. `calloc()` and `realloc()` have been discussed later.

The example given below will ask you how many integers you'd like to store in an array. It'll then allocate the memory dynamically using `malloc()` and store a certain number of integers, print them out, then releases the used memory using **free**.

**Example 5:**

```
#include <stdio.h>
#include <stdlib.h> /* required for the malloc and free functions */
int main()
{
    int number;
    int *ptr;
    int i;
    printf("How many ints would you like store? ");
    scanf("%d", &number);
    ptr = (int *) malloc (number*sizeof(int)); /* allocate memory */
    if(ptr!=NULL)
    {
        for(i=0 ; i<number ; i++)
        {
            *(ptr+i) = i;
        }
        for(i=number ; i>0 ; i--)
        {
            printf(”%d\n”,*(ptr+(i-1))); /* print out in reverse order
*/
        }
        free(ptr); /* free allocated memory */
        return 0;
    }
    else
    {
        printf(”\nMemory allocation failed - not enough memory.\n”);
        return 1;
    }
}
```

Output if entered 3:

```
How many ints would you like store? 3
2
1
0
```

➢ **calloc()**

`calloc` is similar to `malloc`, but the main difference is that the values stored in the allocated memory space is zero by default. With `malloc`, the allocated memory could have any value.

`calloc` requires two arguments. The first is the number of variables you’d like to allocate memory for. The second is the size of each variable.

```
void *calloc( size_t num, size_t size );
```

Like `malloc`, `calloc` will return a void pointer if the memory allocation was successful, else it will return a NULL pointer.

The example given below shows you how to call `calloc` and reference the allocated memory using an array index. The initial value of the allocated memory is printed out in the `for` loop.

**Example 6:**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float *calloc1, *calloc2;
    int i;
    calloc1 = (float *) calloc(3, sizeof(float));
    calloc2 = (float *)calloc(3, sizeof(float));
    if(calloc1!=NULL && calloc2!=NULL)
    {
        for(i=0 ; i<3 ; i++)
        {
            printf(”calloc1[%d] holds %05.5f ”, i, calloc1[i]);
            printf(”\ncalloc2[%d] holds %05.5f ”, i, *(calloc2+i));
        }
        free(calloc1);
        free(calloc2);
        return 0;
    }
    else
    {
        printf(”Not enough memory\n”);
        return 1;
    }
}
```

**Output:**

```
calloc1[0]   holds   0.00000
calloc2[0]   holds   0.00000
calloc1[1]   holds   0.00000
calloc2[1]   holds   0.00000
calloc1[2]   holds   0.00000
calloc2[2]   holds   0.00000
```

On all machines, the `calloc1` and `calloc2` arrays should hold zeros. `calloc` is especially useful when you're using multi-dimensional arrays. Here is another example to demonstrate the use of `calloc()` function.

**Example 7:**

```
/* This program gets the number of elements, allocates spaces for the
elements, gets a value for each element, sum the values of the elements,
and print the number of the elements and the sum.
*/
#include <stdio.h>
#include <stdlib.h>
main()
{
    int *a, i, n, sum = 0;
    printf ( "\n%s%s", "An array will be created dynamically. \n\n",
"Input an array size n followed by integers : " );
    scanf( "%d", &n); /* get the number of elements */
    a = (int *) calloc (n, sizeof(int) ); /* allocate space */
    /* get a value for each element */
    for( i = 0; i < n; i++ )
    {
        printf("Enter %d values : ",n);
        scanf( "%d", a + i );
    }
    /* sum the values */
    for(i = 0; i < n; i++ )
        sum += a[i];
    free(a); /* free the space */
    /* print the number and the sum */
    printf ( "\n%s%7d\n%s%7d\n\n", "Number of elements: ", n, "Sum of
the elements: ", sum );
}
```

➢ **realloc()**

Suppose you've allocated a certain number of bytes for an array but later find that you want to add values to it. You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using `realloc`, without losing your data.

`realloc()` takes two arguments. The first is the pointer referencing the memory. The second is the total number of bytes you want to reallocate.

```
void *realloc( void *ptr, size_t size );
```

Passing zero as the second argument is the equivalent of calling **free**.

Once again, `realloc` returns a void pointer if successful, else a NULL pointer is returned.

This example uses `calloc` to allocate enough memory for an `int` array of five elements. Then `realloc` is called to extend the array to hold seven elements.

**Example 8:**

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr;
    int i;
    ptr = (int *)calloc(5, sizeof(int *));
    if(ptr!=NULL)
    {
        *ptr = 1;
        *(ptr+1) = 2;
        ptr[2] = 4;
        ptr[3] = 8;
        ptr[4] = 16;
        /* ptr[5] = 32; wouldn’t assign anything */
        ptr = (int *)realloc(ptr, 7*sizeof(int));
        if(ptr!=NULL)
        {
            printf(“Now allocating more memory... \n”);
            ptr[5] = 32; /* now it’s legal! */
            ptr[6] = 64;
            for(i=0;i<7;i++)
            {
                printf(”ptr[%d] holds %d\n”, i, ptr[i]);
            }
            realloc(ptr,0); /* same as free(ptr); - just fancier! */
            return 0;
        }
        else
        {
            printf(”Not enough memory - realloc failed.\n”);
            return 1;
        }
    }
    else
    {
        printf(”Not enough memory - calloc failed.\n”);
        return 1;
    }
}
```

**Output :**

```
Now allocating more memory...

ptr[0] holds  1
ptr[1] holds  2
ptr[2] holds  4
ptr[3] holds  8
ptr[4] holds  16
ptr[5] holds  32
ptr[6] holds  64
```

Notice the two different methods that used when initializing the `array:  ptr[2] = 4`; is the equivalent to `*(ptr+2) = 4;` (just easier to read!).

Before using **`realloc`**, assigning a value to **`ptr[5]`** wouldn't cause a compile error. The program would still run, but **`ptr[5]`** wouldn't hold the value you assigned.

## Summary

- A pointer provides a way of accessing a variable without referring to the variable directly.
- A pointer is a variable, which contains the address of a memory location of another variable, rather than its stored value.
- A pointer declaration consists of a base type, an *, and the variable name.
- There are two special operators which are used with pointers: * and &.
- The & operator returns the memory address of the operand.
- The second operator, *, is the complement of &. It returns the value contained in the memory location pointed to by the pointer variable's value.
- Addition and subtraction are the only operations, which can be performed on pointers.
- Two pointers can be compared in a relational expression only if both these variables are pointing to variable(s) of the same type.
- Pointers are passed to a function as arguments, enabling data items within the called routine of the program to access variables whose scope does not extend beyond the calling function.
- An array name is truly a pointer to the first element in that array.
- A pointer constant is an address; a pointer variable is a place to store addresses.
- Memory can be allocated as and when needed by using the malloc(), calloc(), and realloc() functions. Allocating memory in this way is known as Dynamic Memory Allocation.

## Check Your Progress

1. A __________ provides a way of accessing a variable without referring to the variable directly.

| | |
|---|---|
| A. Array | B. Pointer |
| C. Structure | D. None of the above |

2. Pointers cannot point to arrays. (T/F)

3. The ___________ of the pointer defines what type of variables the pointer can point to.

| | |
|---|---|
| A. Type | B. Size |
| C. Content | D. None of the above |

4. The two special operators used with pointers are ____ and _____.

| | |
|---|---|
| A. ^ and % | B. ; and ? |
| C. * and & | D. None of the above |

5. _________ and ___________ are the only operations, which can be performed on pointers.

| | |
|---|---|
| A. Addition, Subtraction | B. Multiplication, Division |
| C. Division, Addition | D. None of the above |

6. Two pointers can be compared only if both these variables are pointing to variables of different types. **(T/F)**

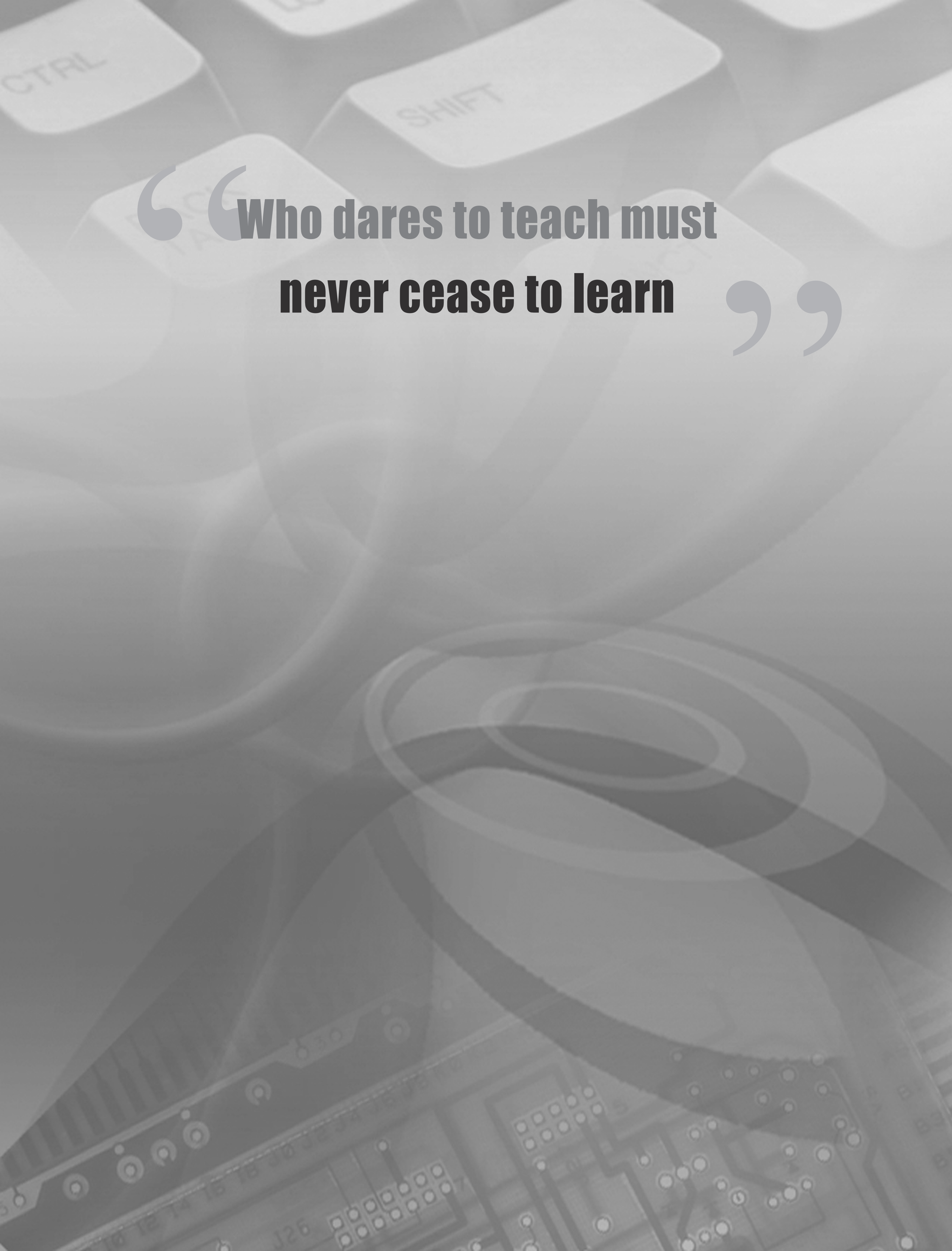
7. The allocation of memory in this manner, that is, as and when required in a program is known as ___________ .

| | |
|---|---|
| A. Dynamic Memory Allocation | B. Static Memory Allocation |
| C. Content Memory Allocation | D. None of the above |

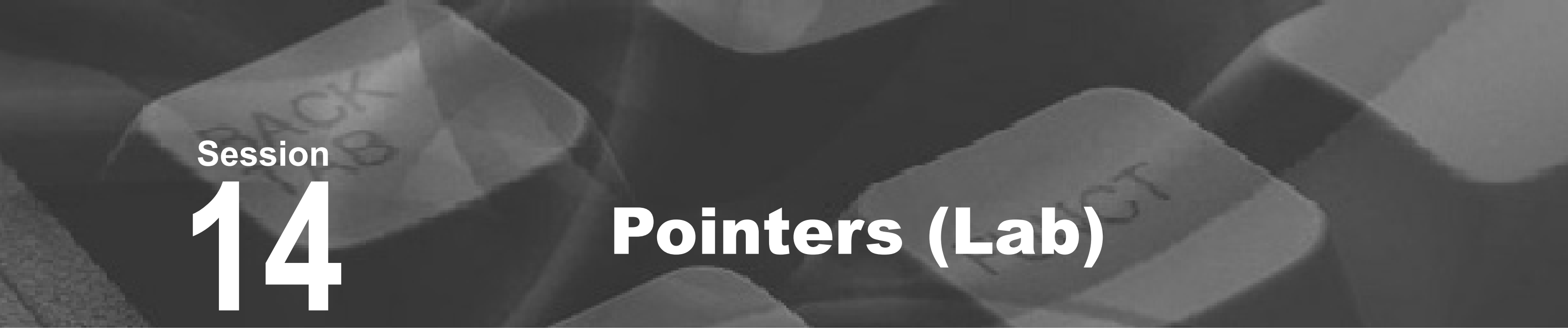
## Try It Yourself

1. Write a program to accept a string and find out if it is a palindrome.

2. Write a program using pointer to strings that accepts the name of an animal and a bird and returns the names in plural.



“Who dares to teach must never cease to learn”



# Session 14 Pointers (Lab)

**Objectives**

**At the end of this session, you will be able to:**

- *Use Pointers*
- *Use Pointers with Arrays*

The steps given in this session are detailed comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

**Part I – For the first 1 Hour and 30 Minutes:**

## 14.1 Pointers

Pointer variables in C can hold the address of a variable of any basic data type. That is, pointers can be of integer or char data type. An integer pointer variable will hold the address of an integer variable. A character pointer will hold the address of a character variable.

### 14.1.1 Counting the number of vowels in a string using pointers

Pointers can be used instead of subscripts to traverse through elements in an array. For example, a string pointer can be used to point to the starting memory address of a word. Hence such a pointer could be used to read the characters in that word. To demonstrate this, let us write a C program to count the number of vowels in a word using pointers. The steps are listed below:

**1. Declare a character pointer variable.**

The code will be,

```
char *ptr;
```

**2. Declare a character array and accept value for the same.**

The code will be,

```
char word[10];
printf("\n Enter a word : ");
scanf("%s", word);
```

**3. Assign the character pointer to the string.**

The code will be,

```
ptr = &word[0];
```

The address of the first character of the character array, word, will be stored in the pointer variable, ptr. In other words, the pointer ptr will be point to the first character in the character array word.

**4. Traverse through the characters in the word to find out whether they are vowels or not. In case a vowel is found, increment the count of vowels.**

The code for the same is,

```
int i, vowcnt;
for(i=0;i<strlen(word);i++)
{
    if((*ptr=='a')||(*ptr=='e')||(*ptr=='i')||(*ptr=='o')||(*ptr=='u')|
|(*ptr=='A')||(*ptr=='E')|| (*ptr =='I')||(*ptr=='O')||(*ptr=='U'))
         vowcnt++;
    ptr++;
}
```

**5. Display the word and the number of vowels in the word.**

The code for the same will be,

```
printf("\n The word is : %s \n The number of vowels in the word is :
%d ", word,vowcnt);
```

Let us look at the complete program.

1. **Invoke the editor in which you can type the C program.**

2. **Create a new file.**

3. **Type the following code :**

```
void main()
{
    char *ptr;
    char word[10];
    int i, vowcnt=0;
    printf("\n Enter a word : ");
    scanf("%s",word);
    ptr = &word[0];
    for(i=0;i<strlen(word);i++)
    {
        if((*ptr=='a')||(*ptr=='e')||(*ptr=='i')||(*ptr=='o')||
        (*ptr=='u')|| (*ptr=='A')||(*ptr=='E')||(*ptr=='I')||
        (*ptr=='O')||(*ptr=='U'))
            vowcnt++;
        ptr++;
    }
    printf("\n The word is : %s \n The number of vowels in the word
is : %d ", word,vowcnt);
}
```

To see the output, follow these steps:

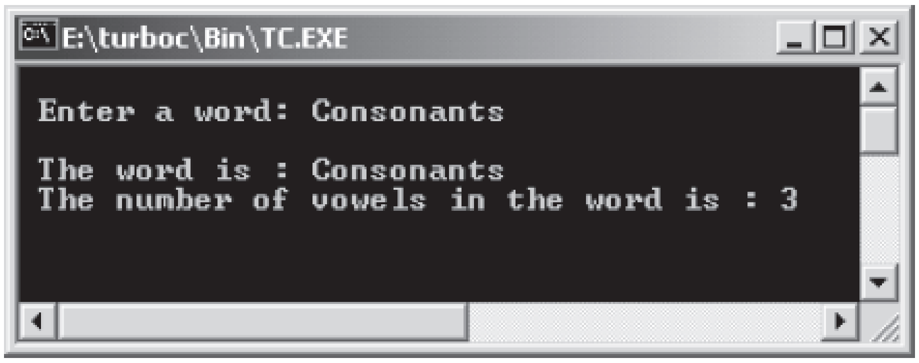
4. **Save the file with the name `pointerI.C`.**

5. **Compile the file, `pointerI.C`.**

6. **Execute the program, `pointerI.C`.**

7. **Return to the editor.**

The sample output of the above program will be as shown in Figure 14.1.



```
Enter a word: Consonants

The word is : Consonants
The number of vowels in the word is : 3
```

Figure 14.1: Output of pointerI.C

## 14.1.2 Sort an array in alphabetical order using pointers

Pointers could be used to swap the contents of two memory addresses. To demonstrate this, let us write a C program to sort a set of strings in alphabetical order.

There are many ways of solving this program. Let us use an array of character pointers to understand the use of array of pointers.

To do this programmatically,

**1. Declare an array of character pointers to hold 5 strings.**

The code for the same is,

```
char *ptr[5];
```

The array looks as depicted in figure 14.2.

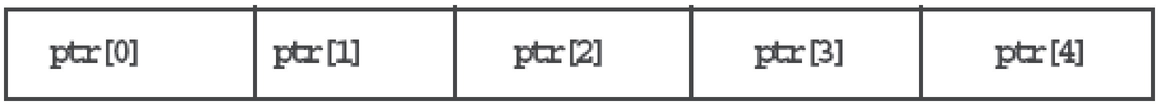


Figure 14.2: Character Pointer Array

**2. Accept 5 strings and assign the pointers in the pointer array to the strings.**

The code is,

```
int i;
char cpyptr1[5][10];
for (i=0;i<5;i++)
{
    printf("\n Enter a string : ");
    scanf("%s",cpyptr1[i]);
    ptr[i]=cpyptr1[i];
}
```

**3. Preserve the array of strings before sorting.**

To do this, we need to create a copy of the array of strings. The code for the same will be,

```
char cpyptr2[5][10];
for (i=0;i<5;i++)
    strcpy(cpyptr2[i],cpyptr1ptr[i]);
```

Here, the `strcpy()` function is used to copy the strings into another array.

**4. Sort the array of strings in alphabetical order.**

The code is,

```
char *temp;
for(i=0;i<4;i++)
{
    for(j=i+1;j<5;i++)
    {
        if (strcmp(ptr[i],ptr[j])>0)
        {
            temp=ptr[i];
            ptr[i]=ptr[j];
            ptr[j]=temp;
        }
    }
}
```

**5. Display the original and the sorted strings.**

The code for the same will be,

```
print("\n The Original list is ");
for(i=0;i<5;i++)
    printf("\n%s",cpyptr2[i]);
printf("\n The Sorted list is ");
for(i=0;i<5;i++)
    printf("\n%s",ptr[i]);
```

Let us look at the complete program.

**1. Create a new file.**

**2. Type the following code :**

```
void main()
{
    char *ptr[5];
    int i;
    int j;
    char cpyptr1[5][10],cpyptr2[5][10];
    char *temp;
    for (i=0;i<5;i++)
    {
        printf("\n Enter a string : ");
        scanf("%s",cpyptr1[i]);
        ptr[i]=cpyptr1[i];
    }
    for (i=0;i<5;i++)
        strcpy(cpyptr2[i],cpyptr1[i]);
    for(i=0;i<4;i++)
    {
        for(j=i+1;j<5;j++)
        {
            if (strcmp(ptr[i],ptr[j])>0)
            {
                temp=ptr[i];
```

```
                    ptr[i]=ptr[j];
                    ptr[j]=temp;
                }
            }
        }
        printf("\n The Original list is ");
        for(i=0;i<5;i++)
            printf("\n%s",cpyptr2[i]);
        printf("\n The Sorted list is ");
        for(i=0;i<5;i++)
            printf("\n%s",ptr[i]);
}
```

To see the output, follow these steps:

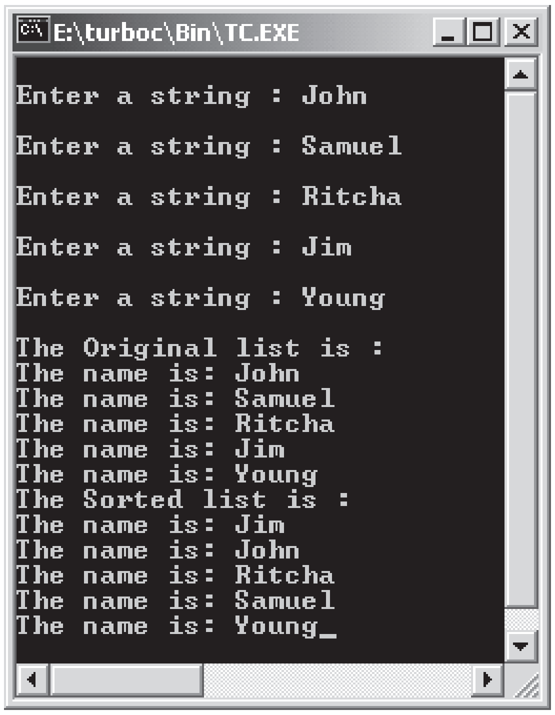
3. **Save the file with the name `pointII.C`.**

4. **Compile the file, `pointII.C`.**

5. **Execute the program, `pointII.C`.**

6. **Return to the editor.**

The sample output of the above program will be as shown in Figure 14.3.



E:\turboc\Bin\TC.EXE

```
Enter a string : John

Enter a string : Samuel

Enter a string : Ritcha

Enter a string : Jim

Enter a string : Young

The Original list is :
The name is: John
The name is: Samuel
The name is: Ritcha
The name is: Jim
The name is: Young
The Sorted list is :
The name is: Jim
The name is: John
The name is: Ritcha
The name is: Samuel
The name is: Young_
```

Figure 14.3: Output of pointII.C

# Session 14

## Pointers (Lab)

**Part II – For the next 30 Minutes:**

1. Write a C Program to concatenate two strings using pointers.

   To do this,

   a. Declare three string variables

   b. Declare three character pointers

   c. Accept the values of two strings

   d. Make the three pointers to point to the three strings variables respectively. The third string will not have any value right now

   e. Loop through the first string and copy the contents of that string to the third string. Use the pointer variables to copy the values

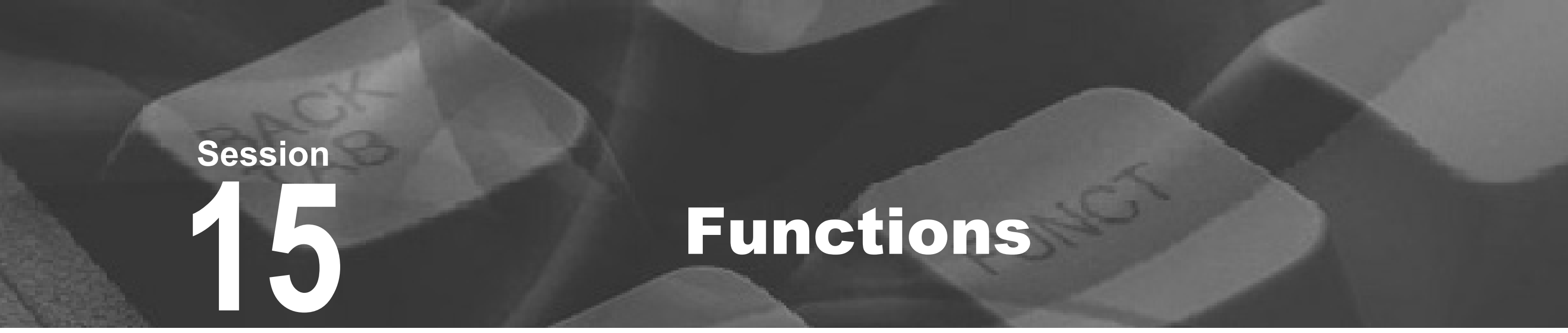
   f. After copying the first string, loop through the second string and copy the contents of that string to the end of the third string. Use the pointer variables to copy the values

   g. Print the three strings

## Try It Yourself

1. Write a C Program to reverse a character array using pointers.

2. Write a C Program to add two matrices using pointers.



# Session 15 Functions

**Objectives**

**At the end of this session, you will be able to:**

- *Explain the use of functions*
- *Explain the structure of a function*
- *Explain function declaration and function prototypes*
- *Explain the different types of variables*
- *Explain how to call functions*
  - *Call by Value*
  - *Call by Reference*
- *Explain the scope rules for a function*
- *Explain functions in multifile programs*
- *Explain Storage classes*
- *Explain function pointers*

## Introduction

A function is a self-contained program segment that carries out a specific, well-defined task. They are actually the smaller segments, which help solve the larger problem.

## 15.1 The use of Functions

Functions are generally used in **C** to execute a series of instructions. However, usage of functions is not similar to that of loops. Loops can repeat a series of instructions only if every iteration immediately follows the previous one. But calling a function causes a series of instructions to be executed at any given point within a program. The functions can be called as many times as required. Suppose a section of code in a program calculates the percentage of some given numbers. If later, in the same program, the same calculation has to be done using different numbers, instead of writing the same instructions all over again, a function can be written to calculate the percentage of any given numbers. The program can then jump to that function, perform the calculations (in the function) and jump back to the place from where it was called. This will become clear as the discussion proceeds.

Another important aspect is that functions are easier to write and understand. Simple functions can be written to do specific tasks. Also, debugging the program is easier as the program structure is more readable, due to its simplified form. Each function can individually be tested for all possible inputs, for valid as well as invalid data. Programs containing functions are also easier to maintain, because modifications, if required, can be restricted to certain functions within the program. Not only can a function be called from different points in a program, but also they can be put in a library of related functions and used by many programs, thus saving on coding time.

## 15.2 The Function Structure

The general syntax of a function in C is:

```
type_specifier function_name (arguments)
{
    body of the function
    return statement
}
```

The **`type_specifier`** specifies the data type of the value, which will be returned by the function. If no type is specified, the function is assumed to return an integer result. The arguments are separated by commas. A pair of empty parentheses must follow the function name even if the it does not include any arguments. Arguments appearing in the parentheses are also termed as **formal parameters** or **formal arguments**. The body of the function may consist of one or many statements. A function should return a value and hence at least one return statement should be there in a function.

### 15.2.1 Arguments of a function

Before discussing arguments in details, consider the following example,

**Example 1:**

```
#include <stdio.h>
main()
{
    int i;
    for(i=1; i<=10; i++)
        printf("\nSquare of %d is %d ", i, squarer(i));
}
squarer(int x)
```

```
/* int x */
{
    int j;
    j = x * x;
    return(j);
}
```

The above program calculates the square of numbers from 1 to 10. This is done by calling the function **squarer**. The data is passed from the calling routine (`main()` in the above case) to the called function **squarer** through arguments. The arguments are known as **actual arguments** in the calling routine and as **formal arguments** in the called function definition (`squarer()`). The data type of the **actual arguments** should be the same as the **formal arguments**. Also the number and order of the **actual arguments** should be the same as the **formal arguments.**

When a function is invoked, the control is passed to the called function where the **formal arguments** are replaced by **actual arguments**. The function is then executed and, on encountering the return statement, control is transferred back to the calling program.

The function `squarer()` is called by passing the number whose square is required. The argument **`x`** can be declared in one of the following ways, while defining the function.

**Method 1:**

```
squarer(int x)
/* x defined with its type in parentheses */
```

**Method 2:**

```
squarer(x)
int x;
/* x is in parentheses, and its type is defined immediately after the */
/* function name */
```

Note that in the last case, **`x`** has to be defined immediately after the function name, before the code block. This is helpful when many parameters of the same data type are passed. In such cases the type has to be mentioned only once at the beginning.

When the arguments are declared within the parentheses, each and every argument has to be defined individually, irrespective of whether they are of the same type. For example, if **`x`** and **`y`** are two arguments of a function `abc()`, then `abc(char x, char y)` is right declaration and `abc(char x, y)` is wrong.

## 15.2.2 Returning from the Function

The return statement has two purposes:

➢ It immediately transfers the control from the function back to the calling program

Whatever is inside the parentheses following the `return` is returned as a value to the calling program.

In the function `squarer()`, a variable **`j`** of the type `int` is defined, which stores the square of the passed argument. This value of this variable is returned to the calling routine through the `return` statement. A function can do a special task and return control to the calling routine without returning any value. In such a case, the `return` function can be written as `return(0)` or `return`. Note that if a function is supposed to return a value and it does not do that then it will return some garbage value.

In the program to calculate squares of numbers, the program passes data to the function squarer through arguments. There can be functions that can be called without any arguments. Here, the function performs a sequence of statements and returns the value, if required.

Note that the function `squarer()` can also be written as:

```
squarer(int x)
{
   return(x*x);
}
```

This is because an expression is valid in `return` statement as it is in an argument. As a matter of fact, the `return` function can be used in any of the following ways:

```
return;
return(constant);
return(variable);
return(expression);
return(statement on evaluation); for example: return(a>b?a:b);
```

However, a limitation of `return` is that it can return only one value.

## 15.2.3 The type of a Function

The **`type-specifier`** is used to specify the data type of the return argument of a function. In the explained example, the **`type-specifier`** is not written besides the function `squarer()`, because `squarer()` returns a value of `int` type. The **`type-specifier`** is not compulsory if an integer value is returned or if no value is returned. However, it is better to specify `int` if an integer value is to be returned and similarly `void` if the function returns nothing.

## 15.3 Invoking a Function

A function can be invoked or called from the main program simply by using its name, followed by parentheses. The parentheses are necessary to inform the compiler that a function is being referred to. When a function name is used in the calling program it is either part of a statement or a statement itself. Hence the statement always ends with a semi-colon. However, when defining the function, a semi-colon is not used in the end. The absence of the semi-colon indicates the compiler that a function is being defined and not called.

Some points to remember:

- A semicolon is used at the end of the statement when a function is called, but not after the function definition.
- Parentheses are compulsory after the function name, irrespective of whether the function has arguments or not.
- The **function**, which calls another **function**, is known as the **calling routine** or **calling function** and the **function**, which is being called, is known as the **called routine** or **called function**.
- Functions not returning an integer value have to specify the type of value being returned.
- Only one value can be returned by a function.
- A program can have one or more functions.

## 15.4 Function Declaration

A function should be declared in the `main()` function, before it is defined or used. This has to be done in case the function is called before it is defined.

Consider the following code snippet:

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

The `main()` function calls the function `address()` and the `address()` function is called before it is defined. Also it is not declared in the `main()` function. This is possible in some C compilers, because the function `address()` is called through a statement which contains nothing else but the function call. This is referred to as **implicit declaration** of a function.

## 15.5 Function Prototypes

A function prototype is a function declaration that specifies the data types of the arguments. Normally, functions are declared by specifying the type of value to be returned by the function, and the function name. However, the ANSI C standard allows the number and types of the function's arguments to be declared. A function `abc()` having two arguments **`x`** and **`y`** of type `int`, and returning a `char` type, can be declared as:

```
char abc();
```

or

```
char abc(int x, int y);
```

The later definition is called **function prototype**. When prototypes are used, C can find and report any illegal type conversions between the arguments used to call a function and the type definition of its parameters. An error will be reported even if there is a difference between the number of arguments used to call a function and define a function.

The general syntax of a function prototype definition is:

```
type function_name(type parm_name1, type parm_name2,..type parm_nameN);
```

When the function is declared without any prototype information, the compiler assumes that no information about the parameters is given. A function having no arguments can be mistaken to be declared without prototype information. To avoid this, when a function has no parameters, its prototype uses `void` inside the parentheses. As said earlier, `void` explicitly declares a function as returning no value.

For example, if a function called `noparam()` returns `char` type and has no parameters, it can be declared as

```
char noparam(void);
```

This indicates that the function has no parameters, and any call to that function, which passes parameters to the function is erroneous.

When a non-prototyped function is called all characters are converted to integers and all floats are converted to doubles. However, if a function is prototyped, the types mentioned in the prototype are maintained and no type promotions occur.

## 15.6 Variables

As discussed earlier, variables are named locations in memory that are used to hold a value that may or may not be modified by a program or a function. Variables are basically of three kinds: **local variables**, **formal parameters**, and **global variables**.

1. **Local variables** are those that are declared inside a function.

2. **Formal parameters** are declared in the definition of function as parameters.

3. **Global variables** are declared outside all functions.

### 15.6.1 Local Variables

Local variables are also known as **automatic variables**, as the keyword `auto` can be used to declare them. They can be referred to only by statements that are inside the code block, which declares them. To be precise, a local variable is created upon entry into a block and destroyed upon exit from the block. The most common code block in which a local variable is declared is a function.

Consider the following code snippet:

```
void blkl(void) /* void denotes no value returned */
{
    char ch;
    ch = 'a';
    .
    .
}
```

```
void blk2(void)
{
    char ch;
    ch = 'b';
    .
    .
}
```

The variable `ch` is declared twice, in **`blk1()`** and **`blk2()`**. The variable **`ch`** in **`blk1()`** has no bearing on or no relation to **`ch`** in **`blk2()`** because each **`ch`** is known only to the code where it is declared.

As local variables are created and destroyed within the block in which they are declared, their contents are lost outside the block scope. This implies that they cannot retain their values between calls to functions.

Though the keyword `auto` can be used to declare local variables, it is hardly ever used because all non-global variables are, by default, considered to be local.

Local variables, which are to be used by functions, are generally declared immediately after the function's opening curly brace and before any other statement. These declarations can, however, be made within a block in the function. For example,

```
void blk1(void)
{
    int t;
    t = 1;
    if(t > 5)
    {
        char ch;
        .
        .
    }
    .
}
```

In the above example the variable **`ch`** is created and will be valid only within the '`if`' code block. It cannot be referenced even in the other parts of the function **`blk1().`**

One of the advantages of declaring a variable in this way is that memory will be allocated to it only if the condition to enter the '`if`' block is satisfied. This is because local variables are declared only after the block they are defined in is entered into.

**Note:** The important point to remember is that all local variables have to be declared at the start of the block in which they are defined, prior to any executable program statement.

The following code may not work with most of the compilers:

```
void blk1(void)
{
    int len;
    len = 1;
    char ch; /* This will cause an error */
    ch = 'a';
    .
    .
}
```

## 15.6.2 Formal Parameters

A function using arguments has to declare variables to accept values of the arguments. These variables are called **formal parameters** of the function and act like any local variable inside a function.

These variables are declared inside the parentheses that follows the function's name. Consider this example:

```
blk1(char ch, int i)
{
    if(i > 5)
        ch = 'a';
    else
        i = i + 1;
    return;
}
```

The function **`blk1()`** has two parameters : **`ch`** and **`i`**.

The formal parameters have to be declared along with their types. Like in the above example, **`ch`** is of type `char` and **`i`** is of type `int`. These variables can be used inside the function like normal local variables. They are destroyed upon exit from the function. Care needs to be taken that the formal parameters declared have the same data-types as the arguments that are used to call the function. In case of a type mismatch, C may not display any error but may give unexpected results. This is because, C generally gives some result even in unusual circumstances. The programmer has to ensure that there are no type mismatch errors.

As with local variables, assignments can be made to a function's formal parameters and they can also be used in any allowable C expression.

### 15.6.3 Global Variables

Global variables are visible to the entire program, and can be used by any piece of code. They are declared outside any function of a program and hold their value throughout the execution of the program. These variables can be declared either outside the `main()` or declared anywhere before its first use. However, it is best to declare global variables at the top of the program, that is, before the `main()` function.

```
int ctr; /* ctr is global */
void blk1(void);
void blk2(void);
void main(void)
{
    ctr = 10;
    blk1 ();
    .
    .
}
void blk1(void)
{
    int rtc;
    if (ctr >8)
    {
        rtc = rtc + 1;
        blk2();
    }
}
void blk2(void)
{
    int ctr;
    ctr = 0;
}
```

In the above code, **`ctr`** is a global variable and even though it is declared outside `main()` and **`blk1()`**, it can be referenced within them. The variable **`ctr`** in **`blk2()`**, though, is a local variable and has no connection with the global variable **`ctr`**. If a global and local variable have the same name, all references to that name within the block defining the local variable will be associated with the local variable and not the global variable.

Storage for global variables is in fixed region of memory. Global variables are useful when many functions in a program use the same data. However, unnecessary use of global variables should be avoided mainly because they take up memory for the entire time that the program is executing. Also, using a global variable where a local variable would be enough makes the function using it, less general. This will be clear from the following program code:

```
void addgen(int i, int j)
{
    return(i + j);
}
int i, j;
void addspe(void)
{
    return(i + j);
}
```

Both **`addgen()`** and **`addspe()`** return the sum of the variables **`i`** and **`j`**. The **`addgen()`** function, however, is used to return the sum of any two numbers; while the **`addspe()`** function returns only the sum of the global variables **`i`** and **`j`**.

## 15.7 Storage Classes

Every C variable has a feature called as **storage class**. The storage class defines two aspects of the variable: its **lifetime** and its **visibility** (**or scope**). The **lifetime** of a variable is the length of time it retains a particular value. The **visibility** of a variable defines which parts of a program will be able to recognize it (the variable). A variable may be visible in a block, a function, a file, a group of files, or an entire program.

From the C compiler's point of view, a variable name identifies some physical location within the computer, where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept: the memory or the CPU register. The variable's storage class determines whether the variable should be stored in the memory or in a register. There are four storage classes in C. These are:

- `automatic`
- `external`
- `static`
- `register`

Keywords in bold indicate their usage as a storage specifier. The storage specifier leads the rest of the variable declaration. Its general syntax is:

```
storage_specifier type var_name;
```

### 15.7.1 Automatic Variables

These are nothing but the local variables, which have been discussed previously. The scope of an automatic variable can be smaller than the entire function, if it is declared from within a compound statement. The scope is then restricted to that compound statement. They can be declared using the specifier `auto`, though the declaration is not necessary. Any variable declared within a function or a code block is by default of the class auto and the system sets aside the required memory area for that variable.

### 15.7.2 Extern

In C, a large program can be split into smaller modules, which can be compiled separately and then linked together. This is done to speed up the compilation process for large projects. However, when the modules are linked, all the files have to be told of the global variables required by the program. A global variable can be declared only once. If two global variables having the same name are declared inside the same file, an error message like '**duplicate variable name**' may be displayed or the C compiler might simply choose one variable. The same problem occurs if all global variables required by the program are included in each and every file. Although the compiler does not issue any error message at compile time, the fact remains that copies of the same variable are being made. At the time of linking the files, the linker displays an error message such as '**duplicate label**' because it does not know which variable to use. The `extern` class is used, in a case like this. All global variables are declared in one file and the same variables are declared as `extern` in all other files. Consider the following code:

```
File1                File2
int i, j;            extern int i, j;
char a;              extern char a;
main()               xyz()
{                    {
  .                    i = j * 5
  .                    .
  .                    .
}                    }
abc()                pqr()
{                    {
  i = 123;             j = 50;
   .                   .
   .                   .
}                    }
```

**File2** has the same global variables as **File1**, except for the fact that these variables have the `extern` specifier added to their declaration. This specifier tells the compiler the types and names of the global variables being used without actually creating storage for them again.

When the two modules are linked, all references to the external variables are solved. If a variable has not been declared within a function, the compiler checks whether it matches any of the global variables. If a match is found, the compiler assumes that a global variable is being referred to.

## 15.7.3 Static Variables

The static variables are permanent variables within their own functions or files. Unlike global variables they are not known outside their function or file, but they maintain their values between calls. This means that, if a function terminates and is then re-entered later, the static variables defined within that function would still retain their values. The variable declaration begins with the static storage class designation.

It is possible to define static variables having the same names as leading external variables. The local variables (static as well as auto) take precedence over the external variables and the value of the external variables will be unaffected by any changes of the local variables. External variables having the same names as the local variables in a function cannot be accessed in that function directly.

Initial values can be assigned to variables within static variable declarations but these values must be expressed as constants or expressions. The compiler automatically assigns a default value zero to any static variable, which is not initialized. Initialization takes place at the beginning of the program.

Consider the following two programs. The difference between `auto` and `static` local variables will be quite apparent.

**Automatic variables**

**Example 2:**

```
#include <stdio.h>
main()
{
    incre();
    incre();
    incre();
}
incre()
{
    char var = 65; /* var is automatic variable */
    printf("\nThe character stored in var is %c", var++);
}
```

**The output for the above program will be:**

```
The character stored in var is A
The character stored in var is A
The character stored in var is A
```

**Static variables**

**Example 3:**

```
#include<stdio.h>
main()
{
    incre();
    incre();
    incre();
}
incre()
{
    static char var = 65; /* var is static variable */
    printf("\nThe character stored in var is %c", var++);
}
```

**The output for this program will be:**

```
The character stored in var is A
The character stored in var is B
The character stored in var is C
```

Both the programs call **`incre()`** thrice. In the first program, each time **`incre()`** is called, the variable **`var`** with storage class `auto` (default storage class) is re-initialized to **`65`** (which is the ASCII equivalent of the character A). Thus when the function terminates, the new value of **`var`** (`66`) is lost **(ASCII character B)**.

In the second program, **`var`** is of `static` storage class. Here, **`var`** is initialized to **`65`** only once the program is compiled. At the end of the first function call, **`var`** has a value of **`66`** **(ASCII B)** and similarly in the next function call **`var`** has a value of **`67`** **(ASCII C)**. At the end of the last function call **`var`** is incremented during the execution of the `printf()` statement. This value is lost when the program terminates.

## 15.7.4 Register Variables

Computers have registers in their **Arithmetic Logic Unit** (**ALU**), which are used to temporarily store data that has to be accessed repeatedly. Intermediate results of calculations are also stored in **registers**. Operations performed upon the data stored in **registers** are faster than the data stored in memory. In assembly language, a programmer has access to these **registers** and can move frequently used data into them thus making the program run faster. A high level programmer usually does not have access to the computer's registers. In C, the option of choosing a storage location for a value has been left to the programmer. If a particular value is to be used often (for example, the value that controls a loop), its storage class can be named as **register**. Then if the compiler finds a free register, and if the machine's **registers** are big enough to hold the variable, it (the variable) is placed in that register. Otherwise, the compiler treats register variables as any other automatic variables, i.e. it stores them in the memory. The keyword `register` must be used to define the `register` variables.

The **scope** and **initialization** of the **register** variables is the same as that of **automatic** variables, except for the location of storage. Register variables are local to a function. That is, they come into existence, when the function is invoked and the value is lost once the function is exited from. Initialization of these variables is the responsibility of the programmer.

As the number of registers available is limited, a programmer has to determine which variables used in the program are used repeatedly and then declare them as register variables.

The usefulness of register variables varies from one machine to another and from one C compiler to another. Sometimes, **register** variables are not supported at all – the keyword `register` is accepted but treated just like the keyword `auto`. In other cases, if **register** variables are supported and if the programmer uses it with care, a program can be made to run twice as fast.

The `register` variables are declared as shown below:

```
register int x;
register char c;
```

The register declaration can only be applied to automatic variables and formal arguments. In the latter case, the declaration looks like the following:

```
f(c, n)
register int c, n;
{
    register int i;
    .
    .
    .
}
```

Consider an example, where the sum of the cubes of the digits of a number is equal to the number itself need to be displayed. For example, 370 is such a number because

$3^3 + 7^3 + 0 = 27 + 343 + 0 = 370$

The following program prints all such numbers in the range from **1** to **999**.

**Example 4:**

```
#include <stdio.h>
main()
{
    register int i;
    int no, digit, sum;
   printf(“ \nThe numbers whose Sum of Cubes of Digits is Equal to the number
itself are :\n\n”);
    for(i=1;i<999;i++)
    {
        sum = 0;
        no = i;
        while(no)
        {
            digit = no%10;
            no = no/10;
            sum = sum + digit * digit * digit;
        }
        if(sum==i)
            printf(“t%d\n”, i);
    }
}
```

The output for the above program is:

```
The numbers whose Sum of Cubes of Digits is Equal to the number itself are:
1
153
370
371
407
```

In the above program, the value of `i` ,varies from 1 to `999`. For each of these value, the cubes of individual digits are added and the resultant sum is compared to `i`. If these two values are equal, `i` is displayed. Since `i` is used to control the looping, (the most essential part of the program), it is declared to be of the storage class register. This declaration increases the efficiency of the program.

## 15.8 Scope Rules for a Function

Scope rules are rules that decide whether one piece of code knows about or has access to another piece of code or data. In C, each function of a program is a separate block of code. The code within a function is private or local to that function and cannot be accessed by any statement in any other function except through a call to that function. The code within a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither affect or be affected by other parts of the program. To be precise, the code and data defined within one function cannot interact with the code or data defined in another function because the two functions have different scopes.

In C, all functions are at the same scope level. This means, that a function cannot be defined within another function. Because of this aspect, C is technically not a block-structured language.

## 15.9 Calling the Function

In general, functions communicate with each other by passing arguments. Arguments can be passed in one of the following two ways:

- Call by value
- Call by reference.

### 15.9.1 Call by Value

In C, by default, all function arguments are passed by value. This means that, when arguments are passed to the called function, the values are passed through temporary variables. All manipulations are done on these temporary variables only. The called function cannot change its value. Consider the following example.

**Example 5:**

```
#include <stdio.h>
main()
{
    int a, b, c;
    a = b = c = 0;
    printf("\nEnter 1st integer : ");
    scanf("%d", &a);
    printf("\nEnter 2nd integer : ");
    scanf("%d", & b);
    c = adder(a, b);
    printf("\n\na & b in main() are : %d, % d", a, b);
    printf("\n\nc in main() is : %d", c);
    /* c gives the addition of a and b */
}
adder(int a, int b)
{
    int c;
    c = a + b;
    a *= a;
    b += 5;
    printf("\n\na & b within adder function are: %d, %d ", a, b);
    printf("\nc within adder function is : %d", c);
    return(c);
}
```

The sample output for an input of **2** and **4** will be:

```
a & b in main() are : 2, 4
c in main() is : 6
a & b within adder function are : 4, 9
c within adder function is : 6
```

The above program accepts two integers, which are passed to the function **`adder()`**. The function `adder()` does the following : it takes the two integers as its arguments, adds them, squares the first integer, adds **5** to the second integer, prints the result and returns the sum of the actual arguments. The variables which are used in the `main()` and the **`adder()`** functions have the same name. However nothing else is common between them. They are stored in different memory locations. This is clear from the output of the above program. The variables **a** and **b** in the function **`adder()`** are altered from **2** and **4** to **4** and **9** respectively. However this change does not affect the values of the **a** and **b** in the `main()` function. The variables must be stored in different memory locations. The variable **c** in `main()` is different from the variable **c** in **`adder()`**.

So, arguments are said to be passed using call by value when the value of the variables are passed to the called function and any alterations on this value has no effect on the original value of the passed variable.

## 15.9.2 Call by Reference

When arguments are passed using call by value, the values of the arguments of the calling routine are not changed. However, there may be cases, where the values of the arguments have to be changed. In such cases, **call by reference** could be used. In **call by reference**, the function is allowed access to the actual memory locations of the arguments and therefore can change the value of the arguments of the calling routine.

For example, consider a function, which takes two arguments, interchanges their values and returns them. If a program like the one given below is written for this purpose, it will never work.

**Example 6:**

```
#include <stdio.h>
main()
{
    int x, y;
    x = 15; y = 20;
    printf("x = %d, y = %d\n", x, y);
    swap(x, y);
    printf("\nAfter interchanging x = %d, y = %d\n", x, y);
}
swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
    return;
}
```

The output for the above will be as follows:

```
x = 15, y = 20
After interchanging x = 15, y = 20
```

The function **`swap()`** interchanges the values of **`u`** and **`v`**, but these values are not passed back to the `main()`. This is because the variables **`u`** and **`v`** in **`swap()`** are different from the variables **`u`** and **`v`** used in `main()`. A call by reference can be used to achieve the desired result, because it will change the values of the actual arguments. Pointers are used when a call by reference has to be made.

Pointers are passed to a function as arguments to enable the called routine of the program to access variables whose scope does not extend beyond the calling function. When a pointer is passed to a function, the address of a data item is passed to the function making it possible to freely access the contents of that address from within the function. The function as well as the calling routine recognizes any change made to the contents of the address. In this way, function arguments permit data-items to be altered in the calling routine, enabling a two-way transfer of data between the calling routine and the function. When the function arguments are pointers or arrays, a **call by reference** is made to the function as opposed to a **call by value** for the variable arguments.

Formal arguments of a function, which are pointers, are preceded by an asterisk `(*)`, just like pointer variable declarations, indicating them to be pointers. Actual pointer arguments in a function call must either be declared as pointers or as referenced variables (**`&var`**).

For example, the function definition

```
getstr(char *ptr_str, int *ptr_int)
```

indicates that the arguments **`ptr_str`** points to type `char` and **`ptr_int`** points to type `int`. The function can be called by the statement,

```
getstr(pstr, &var)
```

where, **`pstr`** is declared as a pointer and the address of the variable **`var`** is passed. Assigning a value through,

```
*ptr_int = var;
```

the function, can now assign values to the variable **`var`** in the calling routine, enabling a two way transfer to and from the function.

```
char *pstr;
```

Consider the same example of **`swap()`** as shown in Example 7. This problem will work when pointers are passed instead of variables.

**Example 7:**

```
#include <stdio.h>
void main()
{
    int x, y, *px, *py;
    /* Storing address of x in px */
    px = &x;
    /* Storing address of y in py */
    py = &y;
    x = 15; y = 20;
    printf("x = %d, y = %d \n", x, y);
    swap (px, py);
    /* Passing addresses of x and y */
    printf("\n After interchanging x = %d, y = %d\n", x, y);
}
swap(int *u, int *v)
/* Accept the values of px and py into u and v */
{
    int temp;
    temp = *u;
    *u = *v;
    *v = temp;
    return;
}
```

The output of the above example will be:

```
x = 15, y = 20
After interchanging x = 20, y = 15
```

Two pointer type variables **`px`** and **`py`** are declared, and the addresses of the variables **`x`** and **`y`** are assigned to them. These pointer variables are then passed to the function **`swap()`**, which interchanges the values stored in **`x`** and **`y`** through the pointers.

## 15.10 Nesting of Function Calls

Calling one function from another is said to be **nesting** of **function calls**. A program which checks whether a string is a palindrome or not, could be considered as an example for nested function calls. A palindrome is a string of characters, which is the same when read forward or backwards. Consider the following code:

```
main()
{
    .
    .
    palindrome();
    .
    .
}
palindrome()
{
    .
    .
    getstr();
    reverse();
    cmp();
    .
    .
}
```

In the above program, the function `main()` calls the function `palindrome()`. The function **`palindrome()`** calls three other functions **`getstr()`**, **`reverse()`** and **`cmp()`**. The **`getstr()`** function obtains a string of characters from the user, the function **`reverse()`** reverses the input string and the function **`cmp()`** compares the input string and the reversed string.

Since `main()` calls **`palindrome()`**, which in turn calls the **`getstr()`**, **`reverse()`** and **`cmp()`** functions, the function calls are said to be nested within **`palindrome()`**.

Nesting of function calls as shown above is allowed, whereas defining one function within another function is not allowed by C.

## 15.11 Functions in Multifile Programs

Programs can be composed of multiple files. Such programs could make use of lengthy functions, where each function may occupy a separate file. As variables in multifile programs, functions can also be defined as `static` or `external`. The scope of the `external` function is through all the files of the program and it is the default storage class for functions. `Static` functions are recognized only within the program file and their scope does not extend outside the program file. The function header will look like,

```
static fn _type fn_name (argument list)
```

or

```
extern fn_type fn_name (argument list)
```

The keyword `extern` is optional as it is the default storage class.

## 15.12 Pointers to Functions

A confusing yet powerful feature of C is the **function pointer**. Even though a function is not a variable, it has a physical location in memory that can be assigned to a pointer. A function's address is the entry point of the function and function pointer can be used to call a function.

To know how function pointers work, it is necessary to be very clear as to how a function is compiled and called in C. As each function is compiled, source code is transformed into object code and the entry point is established. When a call is made to a function, a machine language call is made to this entry point. Therefore, if a pointer contains the address of a function's entry point, it can be used to call that function. The address of a function can be obtained by using the function's name without any parentheses or arguments. The following program will demonstrate the concept of function pointer.

**Example 8:**

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b, int (*cmp)());
main()
{
    char sl[80];
    int (*p)();
    p = strcmp;
    gets(s1);
    gets(s2);
    check(s1, s2, p);
}
void check(char *a, char *b, int (*cmp)())
{
    printf("testing for equality \n");
    if(!(*cmp)(a,b))
        printf("Equal");
    else
        printf("Not Equal");
}
```

The function **`check()`** is called by passing two character pointers and one function pointer. Inside **`check()`**, the arguments are declared as character pointers and a function pointer. Notice how the function pointers are declared. A similar syntax could be used when declaring other function pointers irrespective of the return type of the function. The parentheses around the **`*cmp`** are necessary for the compiler to interpret this statement correctly.

## Summary

- Functions are generally used in C to execute a series of instructions more than once.
- The `type _ specifier` specifies the data type of the value that will be returned by the function.
- The arguments to a function can be constants, variables, expressions or functions.
- The arguments are known as actual arguments in the calling routine and as formal arguments in the called function.
- A function has to declared in main(), before it is defined or used.
- In C, by default, all function arguments are passed by value.
- Variables are basically of three kinds: local variables, formal parameters, and global variables.
    - Local variables are declared inside a function.
    - Formal parameters are declared in the definition of function parameters.
    - Global variables are declared outside all functions.
- The storage class defines two features of the variable; its **lifetime** and its **visibility** or **scope**).
- Automatic variables are the same as local variables.
- All global variables are declared in one file and the same variables are declared as **extern** in all other files using them.
- The static variables are permanent variables within their own functions or file.
- Unlike global variables, static variables are not known outside their function or file, but they maintain their values between calls.
- If a particular value is to be used often, its storage class can be named as register.

## Summary

- As with variables in multifile programs, functions can also be defined as static or **external**.
- The code and data defined within one function cannot interact with the code or data defined in another function because the two function have different scopes.
- A function cannot be defined within another function.
- A function prototype is a function declaration that specifies the data types of the arguments.
- Calling one function from within another is said to be nesting of **function calls**.
- A function pointer can be used to call a function.

## Check Your Progress

1. A __________ is a self-contained program segment that carries out a specific, well defined task.

2. Arguments appearing in the parentheses are termed as ________.

3. If the return is ignored, control passes to the calling program when the closing braces of the code block are encountered. This is termed as _____________.

4. The function, which calls another function, is known as the _________ and the function, which is being called, is known as the _________.

5. A ________ is a function declaration that specifies the data types of the arguments.

6. __________ can be referred to only by statements that are inside the code block, which declares them.

7. _________ are visible to the entire program, and can be used by any piece of code.

8. __________ govern whether one piece of code knows about or has access to another piece of code or data

9. Arguments are said to be passed _________ when the value of the variables are passed to the called function

10. In__________, the function is allowed access to the actual memory location of the argument.

Session
# 16 Functions (Lab)

**Objectives**

**At the end of this session, you will be able to:**

- *Define and call function*
- *Use of parameters in function*

**Part I – For the first 1 Hour and 30 Minutes:**

## 16.1 Functions

As we already know, a function is a self-contained block of statements that perform a task of some kind. In this session, let us focus on how to create and use functions.

### 16.1.1 Define a function

A function is defined with a function name, is followed by a pair of curly braces in which one or more statements may be present. For example,

```
argentina()
{
    statement 1;
    statement 2;
    statement 3;
}
```

### 16.1.2 Calling a function

A function can be called from a main program by stating its name followed by a pair of braces and a semi-colon. For example,

```
argentina();
```

Now, let us look at the complete program.

**1. Invoke the editor in which you can type the C program.**

**2. Create a new file.**

**3. Type the following code :**

```
#include<stdio.h>
void main()
{
    printf("\n I am in main");
    italy();
    brazil();
    argentina();
}
italy()
{
    printf("\n I am in italy");
}
brazil()
{
    printf("\n I am in brazil");
}
argentina()
{
    printf("\n I am in argentina");
}
```

To see the output, follow these steps:

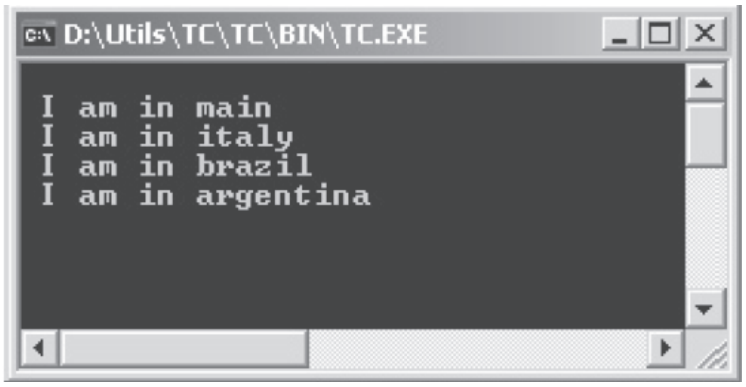
**4. Save the file with the name `functionI.C`.**

**5. Compile the file, `functionI.C`.**

**6. Execute the program, `functionI.C`.**

**7. Return to the editor.**

The sample output of the above program will be as shown in Figure 16.1.



```
I am in main
I am in italy
I am in brazil
I am in argentina
```

**Figure 16.1: Output of functionI.C**

## 16.2 Use of parameters in functions

Parameters are used to convey information to the function. The format strings and the list of variables used inside the parenthesis in the functions are parameters.

### 16.2.1 Define a parameterized function

A function is defined with a function name is followed by an opening brace followed by a parameter(s) and finally closing brace. Inside the function, one or more statements may be present. For example,

```
calculatesum (int x, int y, int z)
{
    statement 1;
    statement 2;
    statement 3;
}
```

Now, Let us look at the complete program.

**1. Create a new file.**

**2. Type the following code :**

```
#include<stdio.h>
void main()
{
    int a, b, c, sum;
    printf("\n Enter any three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    sum = calculatesum(a, b, c);
    printf("\n Sum = %d", sum);
}
calculatesum(int x, int y, int z)
{
    int d;
    d = x + y + z;
    return (d);
}
```

To see the output, follow these steps:

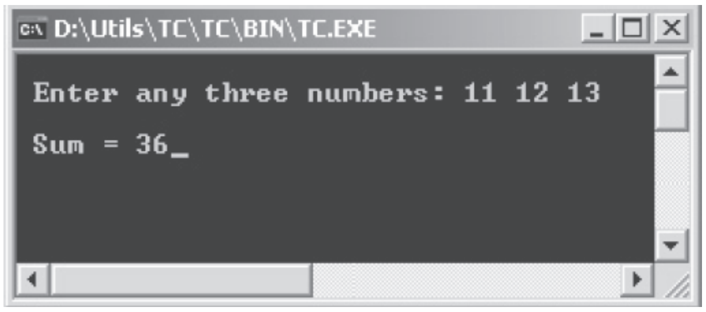
3. **Save the file with the name `functionII.C`.**

4. **Compile the file, `functionII.C`.**

5. **Execute the program, `functionII.C`.**

6. **Return to the editor.**

The sample output of the above program will be as shown in Figure 16.2.



```
Enter any three numbers: 11 12 13
Sum = 36_
```

**Figure 16.2: Output of functionII.C**

**Part II – For the next 30 Minutes:**

1. Write a C program that accepts a number and square the number with the help of a function.

   To do this,

   a. Declare a function.

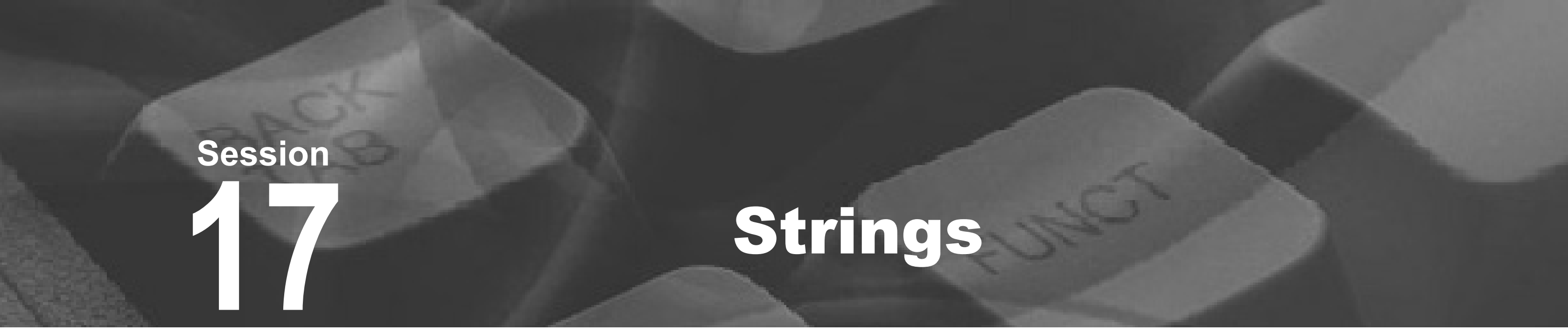
   b. Accept the number.

   c. Pass the number to the function and return the square of that number.

## Try It Yourself

1. Write a C program to find the area and perimeter of a circle.

2. Write a C program to calculate the factorial of an integer.



Session

# 17 Strings

## Objectives

**At the end of this session, you will be able to:**

- *Explain string variables and constants*
- *Explain pointers to strings*
- *Perform string input/output operations*
- *Explain the various string functions*
- *Explain how arrays can be passed as arguments to functions*
- *Describe how strings can be used as function arguments*

## Introduction

Strings in C are implemented as arrays of characters terminated by the NULL ('\0') character. This session discusses the usage and manipulation of strings.

## 17.1 String variables and constants

String variables are used to store a series of characters. Like any other variable, these variables must be declared before they are used. A typical string variable declaration is:

```
char str[10];
```

**`str`** is a character array variable that can hold a maximum of 10 characters. Consider that **`str`** is assigned a string constant,

```
"WELL DONE"
```

A string constant is a sequence of characters surrounded by double quotes. Every character in the string is stored as an array element. In memory, the string is stored as follows:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'W' | 'E' | 'L' | 'L' | ' ' | 'D' | 'O' | 'N' | 'E' | '\0' |

The `'\0'` (null) character is automatically added in the internal representation so that the program can locate the end of the string. So, while declaring a string variable, allow one extra element space for the null terminator.

## 17.1.1 Pointers to strings

Strings can also be stored and accessed using character pointers. A character pointer to a string is declared as follows:

```
char *pstr = “WELCOME”;
```

**`pstr`** is a pointer that is initialized to point to a string constant. The pointer may be modified to point elsewhere. However, the modification would cause the string to be inaccessible.

## 17.1.2 String I/O operations

String input/output (I/O) operations in C are carried out using function calls. These functions are part of the standard I/O library called `stdio.h`. A program that uses the string I/O functions must have the following statement at the beginning:

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

When the program containing this statement is compiled, the contents of the file `stdio.h` become a part of the program.

➢ **Simple string I/O operations**

The `gets()` function is the simplest method of accepting a string through standard input. Input characters are accepted till the **Enter** key is pressed. The `gets()` function replaces the terminating `‘\n’` new line character with the `‘\0’` character. The syntax is as follows:

```
gets(str);
```

where, **`str`** is a character array that has been declared.

Similarly, the `puts()` function is used to display a string on the standard output device. The newline character terminates the string output. The function syntax is:

```
puts(str);
```

where, **`str`** is a character array that has been declared and initialized. The following program accepts a name and displays a message.

**Example 1:**

```
#include <stdio.h>
void main()
{
    char name[20];
    /* name is declared as a single dimensional character array */
    clrscr(); /* Clears the screen */
    puts("Enter your name:"); /* Displays a message */
    gets(name); /* Accepts the input */
    puts("Hi there:");
    puts(name); /* Displays the input */
    getch();
}
```

If the name **Lisa** is entered, a sample output for the above program will be:

```
Enter your name:
Lisa
Hi there:
Lisa
```

➢ **Formatted string I/O operations**

The `scanf()` and `printf()` functions can also be used to accept and display string values. These functions are used to accept and display mixed data types with a single statement. The syntax to accept a string is as follows:

```
scanf("%s", str);
```

where `%s` is the format specifier that states that a string value is to be accepted. **`str`** is a character array that has been declared. Similarly, to display a string, the syntax is:

```
printf("%s", str);
```

where the `%s` format specifier states that a string value is to be displayed and **`str`** is a character array that has been declared and initialized. The `printf()` function can also be used without the format specifier to display messages.

The earlier program can be modified to accept and display a name using `scanf()` and `printf()`.

**Example 2:**

```
#include <stdio.h>
void main()
{
    char name[20];
    /* name is declared as a single dimensional character array */
    clrscr(); /* Clears the screen */
    printf("Enter your name:"); /* Displays a message */
    scanf("%s",name); /* Accepts the input */
    printf("Hi there: %s",name); /* Displays the input */
    getch();
}
```

If the name **`Brendan`** is entered, a sample output for the above program will be:

```
Enter your name: Brendan
Hi there: Brendan
```

## 17.2 String functions

C supports a wide range of string functions, which are found in the standard header file `string.h`. Few of the operations performed by these functions are:

- Concatenating strings
- Comparing strings
- Locating a character in a string
- Copying one string to another
- Calculating the length of a string

### 17.2.1 The 'strcat()' function

The `strcat()` function is used to join two string values into one. The syntax of the function is:

```
strcat(str1, str2);
```

where, **`str1`** and **`str2`** are two character arrays that have been declared and initialized. The value in

**str2** is attached at the end of **str1**.

The following program accepts a first name and a last name. It concatenates the last name to the first name and displays the concatenated name.

**Example 3:**

```
#include<stdio.h>
#include<string.h>
void main()
{
    char firstname[15];
    char lastname[15];
    clrscr();
    printf("Enter your first name:");
    scanf("%s",firstname);
    printf("Enter your last name:");
    scanf("%s",lastname);
    strcat(firstname, lastname);
    /* Attaches the contents of lastname at the end of firstname */
    printf("%s", firstname);
    getch();
}
```

A sample output for the above program will be:

```
Enter your first name: Carla
Enter your last name: Johnson
CarlaJohnson
```

### 17.2.2 The 'strcmp()' function

The equality (or inequality) of two numbers can be verified using relational operators. However, to compare strings, a function call has to be made. The function `strcmp()` compares two strings and returns an integer value based on the results of the comparison. The syntax of the function is:

```
strcmp(str1, str2);
```

where, **str1** and **str2** are two character arrays that have been declared and initialized. The function returns a value:

- Less than zero if **str1** < **str2**
- Zero if **str1** is same as **str2**
- Greater than zero if **str1** > **str2**

The following program compares one name with three other names and displays the results of the comparisons.

**Example 4:**

```
#include <stdio.h>
#include<string.h>
void main()
{
    char name1[15] = “Geena”;
    char name2[15] = “Dorothy”;
    char name3[15] = “Shania”;
    char name4[15] = “Geena”;
    int i;
    clrscr();
    i = strcmp(name1,name2);
    printf(“%s compared with %s returned %d\n”, name1, name2, i);
    i=strcmp(name1,name3);
    printf(“%s compared with %s returned %d\n”, name1, name3, i);
    i=strcmp(name1,name4);
    printf(“%s compared with %s returned %d\n”, name1, name4, i);
    getch();
}
```

A sample output for the above program will be:

```
Geena compared with Dorothy returned 3
Geena compared with Shania returned -12
Geena compared with Geena returned 0
```

Note the value returned for each comparison. It is the difference between the ASCII values of the first different characters encountered in the two strings.

## 17.2.3 The ‘strchr()’ function

The `strchr()` function determines the occurrence of a character in a string. The syntax of the function is:

```
strchr(str, chr);
```

where, **`str`** is a character array or string. **`chr`** is a character variable containing the value to be searched. The function returns a pointer to the value located in the string, or NULL if it is not present.

The following program determines whether the character ‘a’ occurs in two specified city names.

**Example 5:**

```
#include<stdio.h>
#include<string.h>
void main()
{
    char str1[15] = “New York”;
    char str2[15] = “Washington”;
    char chr = ‘a’, loc;
    clrscr();
    loc = strchr(str1,chr);
    /* Checks for the occurrence of the character value held by chr */
    /* in the first city name */
    if(loc != NULL)
        printf(“%c occurs in %s\n”, chr, str1);
    else
        printf(“%c does not occur in %s\n”, chr, str1);

    loc = strchr(str2,chr);
    /* Checks for the occurrence of the character in the second city name */
    if(loc != NULL)
        printf(“%c occurs in %s\n”, chr, str2);
    else
        printf(“%c does not occur in %s\n”, chr, str2);

    getch();
}
```

The output for the above program is:

```
a does not occur in New York
a occurs in Washington
```

## 17.2.4 The 'strcpy()' function

There are no operators in C for handling a string as a single unit. So, the assignment of one string value to another requires the use of the function `strcpy()`. The syntax of the function is:

```
strcpy(str1, str2);
```

where, **`str1`** and **`str2`** are character arrays that have been declared and initialized. The function copies the value in **`str2`** onto **`str1`** and returns **`str1`**.

The following program demonstrates the use of the `strcpy()` function. It changes the name of a hotel and displays the new name.

**Example 6:**

```
#include <stdio.h>
#include<string.h>
void main()
{
    char hotelname1[15] = "Sea View";
    char hotelname2[15] = "Sea Breeze";
    clrscr();
    printf("The old name is %s\n", hotelname1);
    strcpy(hotelname1, hotelname2);
    /* Changes the hotel name */
    printf("The new name is %s\n", hotelname1);
    /* Displays the new name */
    getch();
}
```

The output of the above program is:

```
The old name is Sea View
The new name is Sea Breeze
```

## 17.2.5 The 'strlen()' function

The `strlen()` function returns the length of a string. The length of a string can be useful in programs accessing each character of a string in a loop. The syntax of the function is:

```
strlen(str);
```

where, **`str`** is a character array that has been declared and initialized. The function returns the length of **`str`**.

The following program shows a simple implementation of the `strlen()` function. It determines the length of a company name and displays the name along with additional characters.

**Example 7:**

```
#include <stdio.h>
#include<string.h>
void main()
{
    char compname[20] = "Microsoft";
    int len, ctr;
    clrscr();
    len = strlen(compname);
    /* Determines the length of the string */
    for(ctr = 0; ctr < len; ctr++)
        /* Accesses and displays each character of the string */
        printf("%c * ", compname[ctr]);
    getch();
}
```

The output of the above program is:

```
M * i * c * r * o * s * o * f * t *
```

## 17.3 Passing Arrays to Functions

In C, when an array is passed as an argument to a function, only the address of the array is passed. The array name without the subscripts refers to the address of the array. The following code snippet passes the address of the array **`ary`** to the function **`fn_ary()`**:

```
void main()
{
    int ary[10];
    ...
    fn_ary(ary);
    ...
}
```

If a function receives a single-dimensional array, the formal parameters can be declared in one of the following ways.

```
fn_ary (int ary [10]) /* sized array */
{
    :
}
```

**or**

```
fn_arry (int ary []) /* unsized array */
{
    :
}
```

Both the above declarations produce the same results. The first method employs the standard array declaration. In the second version, the array declaration simply specifies that an array of type `int` of some length is required.

The following program accepts numbers into an integer array. The array is then passed to a function **`sum_arr()`**. The function computes and returns the sum of the numbers in the array.

**Example 8:**

```
#include <stdio.h>
void main()
{
    int num[5], ctr, sum=0;
    int sum_arr(int num_arr[]); /* Function declaration */
    clrscr();
    for(ctr = 0; ctr < 5; ctr++) /* Accepts numbers into the array */
    {
        printf("\nEnter number %d: ", ctr+1);
        scanf("%d", &num[ctr]);
    }
    sum = sum_arr(num); /* Invokes the function */
    printf("\nThe sum of the array is %d", sum);
    getch();
}
    int sum_arr(int num_arr[]) /* Function definition */
    {
        int i, total;
        for(i=0,total=0;i<5;i++) /* Calculates the sum */
           total+=num_arr[i];
        return total; /* Returns the sum to main() */
    }
```

A sample output of the above program is:

```
Enter number 1: 5
Enter number 2: 10
Enter number 3: 13
Enter number 4: 26
Enter number 5: 21
The sum of the array is 75
```

## 17.4 Passing Strings to Functions

Strings, or character arrays, can also be passed to functions. For example, the following program accepts strings into a two-dimensional character array. Then, the array is passed to a function that determines the longest string in the array.

**Example 9:**

```
#include <stdio.h>
void main()
{
    char lines[5][20];
    int ctr, longctr=0;
    int longest(char lines_arr[][20]);
    /* Function declaration */
    clrscr();
    for(ctr = 0; ctr < 5; ctr++) /* Accepts string values into the array */
    {
        printf("\nEnter string %d: ", ctr+1);
        scanf("%s", lines[ctr]);
    }
    longctr = longest(lines);
    /* Passes the array to the function */
    printf("\nThe longest string is %s", lines[longctr]);
    getch();
}
int longest(char lines_arr[][20]) /* Function definition */
{
    int i=0, l_ctr=0, prev_len, new_len;
    prev_len = strlen(lines_arr[i]);
    /* Determines the length of the first element */
    for(i++; i<5; i++)
    {
        new_len=strlen(lines_arr[i]);
        /* Determines the length of the next element */
        if(new_len > prev_len)
            l_ctr=i;
        /* Stores the subscript of the longer string */
            prev_len = new_len;
    }
    return l_ctr;
    /* Returns the subscript of the longest string */
}
```

A sample output of the program is given below.

```
Enter string 1: The
Enter string 2: Sigma
Enter string 3: Protocol
Enter string 4: Robert
Enter string 5: Ludlum
The longest string is Protocol
```

## Summary

- Strings in C are implemented as arrays of characters terminated by the NULL ('\0') character.
- String variables are used to store a series of characters.
- A string constant is a sequence of characters surrounded by double quotes.
- Strings can be stored and accessed using character pointers.
- String I/O operations in C are carried out using functions that are part of the standard I/O library called stdio.h.
- The gets() and puts() functions are the simplest method of accepting and displaying strings respectively.
- The scanf() and printf() functions can be used to accept and display strings along with other data types.
- C supports a wide range of string functions, which are found in the standard header file string.h.
- The strcat() function is used to join two string values into one.
- The function strcmp() compares two strings and returns an integer value based on the results of the comparison.
- The strchr() function determines the occurrence of a character in a string.
- The strcpy() function copies the contents of one string onto another.
- The strlen() function returns the length of a string.
- In C, when an array is passed as an argument to a function, only the address of the array is passed.
- The array name without the subscripts refers to the address of the array.

## Check Your Progress

1. Strings are terminated by the __________ character.

2. The number of characters that can be input into `char arr[15]` is __________.

3. Modification of the string pointer can lead to data loss. **(True / False)**

4. The character is used to print a new line in `printf()`.

5. To use the `strcat()` function, the _________ header file must be included in the program.

6. Two pointers can be compared only if both these variables are pointing to variables of different types. **(True / False)**

7. `strcmp()` returns ____________ if two strings are identical.

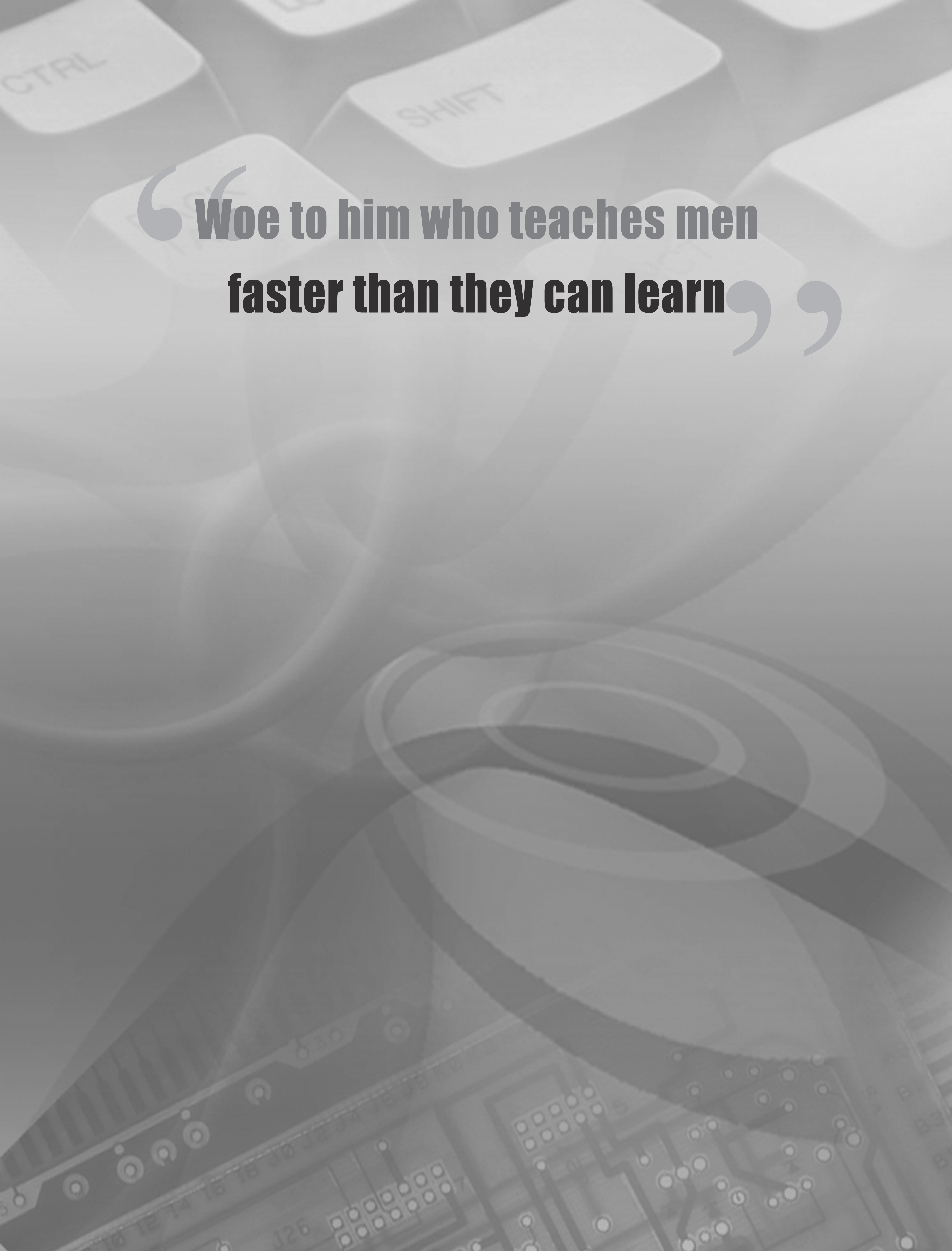
8. When an array is passed to a function, only its ____________ is passed.

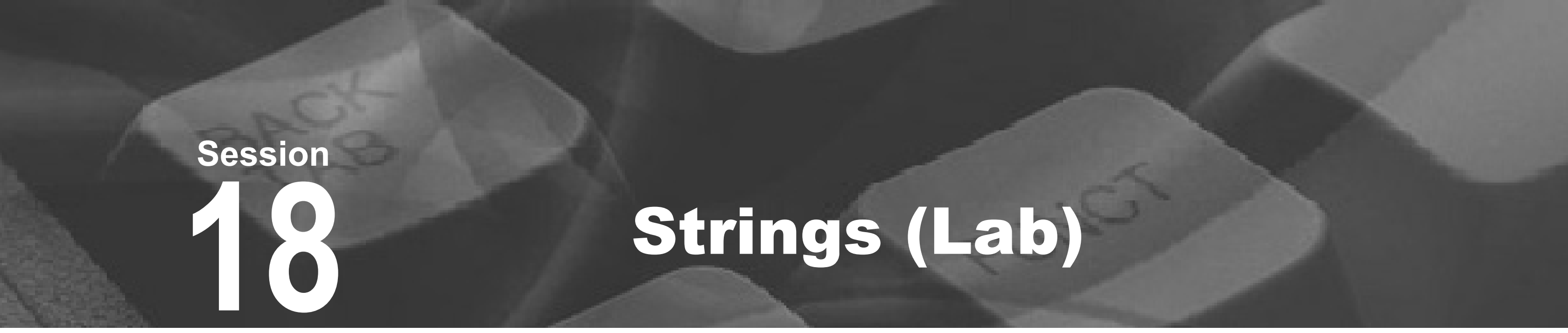
## Try It Yourself

1. Write a program that accepts two strings. The program should determine whether the first string occurs at the end of the second string.

2. Write a program that accepts an array of integers and displays the average. Use a function to calculate the average.

“Woe to him who teaches men faster than they can learn”





# Session 18 Strings (Lab)

**Objectives**

**At the end of this session, you will be able to:**

- *Use strings functions*
- *Pass arrays to functions*
- *Pass strings to functions*

The steps given in this session are detailed comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

**Part I – For the first 1 Hour and 30 Minutes:**

## 18.1 Strings Functions

The string functions in C are found in the standard header file `string.h`. This file must be included in each program that uses the string functions.

### 18.1.1 Sorting strings using library functions

String functions are useful for manipulating character arrays. For example, the length of a string can be determined using the `strlen()` function. Let us write a C program to sort 5 strings in descending order of their length. The steps are listed below:

**1. As we learnt in the theory session, in C, to use the string functions from the library we need to include the two header files: `stdio.h, string.h`.**

The code for the same will be,

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

**2. Declare a character array to hold the 5 strings.**

The code will be,

```
char str_arr[5][20];
```

**3. Accept the five strings in a `for` loop.**

The code will be,

```
for(i=0;i<5;i++)
{
  printf("\nEnter string %d: ",i+1);
  scanf("%s", str_arr[i]);
}
```

**4. Compare the length of each string with the others. Swap if the string length is less than the other.**

The code will be,

```
for(i=0;i<4;i++)
  for(j=i+1;j<5;j++)
  {
    if(strlen(str_arr[i]) < strlen(str_arr[j]))
    {
       strcpy(str, str_arr[i]);
       strcpy(str_arr[i], str_arr[j]);
       strcpy(str_arr[j], str);
    }
  }
```

An array `str` is being used to aid the swap operation.

**5. Display the strings in the sorted array.**

The code for the same will be,

```
printf("\nThe strings in descending order of length are:");
for(i=0;i<5;i++)
   printf("\n%s", str_arr[i]);
```

Let us look at the complete program.

1. **Invoke the editor in which you can type the C program.**

2. **Create a new file.**

3. **Type the following code:**

```
#include <stdio.h>
#include <string.h>
void main()
{
    int i, j;
    char str_arr[5][20], str[20];
    clrscr();
    for(i=0;i<5;i++)
    {
        printf("\nEnter string %d: ",i+1);
        scanf("%s", str_arr[i]);
    }
    for(i=0;i<4;i++)
        for(j=i+1;j<5;j++)
        {
            if(strlen(str_arr[i]) < strlen(str_arr[j]))
            {
                strcpy(str, str_arr[i]);
                strcpy(str_arr[i], str_arr[j]);
                strcpy(str_arr[j], str);
            }
        }
    printf("\nThe strings in descending order of length are:");
    for(i=0;i<5;i++)
        printf("\n%s", str_arr[i]);
    getch();
}
```

To see the output, follow these steps:

4. **Save the file with the name `stringI.C`.**

**5. Compile the file, `stringI.C`.**

**6. Execute the program, `stringI.C`.**

**7. Return to the editor.**

The sample output of the program is given below:

```
Enter string 1: This
Enter string 2: sentence
Enter string 3: is
Enter string 4: not
Enter string 5: sorted
The strings in descending order of length are:
sentence
sorted
This
not
is
```

## 18.1.2 Convert a character array to upper case using functions

Strings can be passed to functions for manipulation. When strings, or character arrays, are passed to functions, actually the address is passed. To demonstrate this, let us write a C program to convert a set of strings to upper case. The conversion to upper case will be achieved using a function.

The steps are listed below:

**1. Include the required header files.**

The code will be,

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

**2. Declare an array to hold 5 strings.**

The code for the same is,

```
char names[5][20];
```

**3. Declare a function that accepts a string as an argument.**

The code for the same is,

```
void uppername(char name_arr[]);
```

**4. Accept 5 strings into the array.**

The code is,

```
for(i=0;i<5;i++)
{
    printf("\nEnter string %d: ", i+1);
    scanf("%s", names[i]);
}
```

**5. Pass each string to the function for upper case conversion. After conversion, display the modified string.**

The code for the same will be,

```
for(i=0;i<5;i++)
{
    uppername(names[i]);
    printf("\nNew string %d: %s", i+1, names[i]);
}
```

**6. Define the function.**

The code is,

```
void uppername(char name_arr[])
{
    int x;
    for(x=0;name_arr[x] != '\0'; x++)
    {
        if(name_arr[x]>=97 && name_arr[x]<=122)
            name_arr[x]=name_arr[x]-32;
    }
}
```

The condition checks the ASCII values of each character in the string. If the character is in lower case, it is converted to upper case. Note that the ASCII value of ‘A’ is 65 and that of ‘a’ is 97.

Let us look at the complete program.

**1. Create a new file.**

**2. Type the following code:**

```
#include <stdio.h>
#include <string.h>
void main()
{
    int i;
    char names[5][20];
    void uppername(char name_arr[]);
    clrscr();
    for(i=0;i<5;i++)
    {
        printf(”\nEnter string %d: ”, i+1);
        scanf(”%s”, names[i]);
    }
    for(i=0;i<5;i++)
    {
        uppername(names[i]);
        printf(”\nNew string %d: %s”, i+1, names[i]);
    }
    getch();
}

void uppername(char name_arr[])
{
    int x;
    for(x=0;name_arr[x] != ’\0’; x++)
    {
        if(name_arr[x]>=97 && name_arr[x]<=122)
            name_arr[x]=name_arr[x]-32;
    }
}
```

To see the output, follow these steps:

**3. Save the file with the name `stringII.C`.**

**4. Compile the file, `stringII.C`.**

**5. Execute the program, `stringII.C`.**

**6. Return to the editor.**

A sample output of the program is shown below.

```
Enter string 1: Sharon

Enter string 2: Christina

Enter string 3: Joanne

Enter string 4: Joel

Enter string 5: Joshua

New string 1: SHARON
New string 2: CHRISTINA
New string 3: JOANNE
New string 4: JOEL
New string 5: JOSHUA
```

**Part II – For the next 30 Minutes:**

1. Write a C Program to display the number of times a specified character occurs in a string. Set a loop to perform the operation 5 times.

   To do this,

   a. Declare a character variable and a character array.

   b. Declare a function that accepts a character array and a character variable and returns an integer value.

   c. Set up a loop to accept a string and a character 5 times.

   d. Accept the string and the character.

   e. Pass the string and the character to the function and accept the value returned in an integer variable.

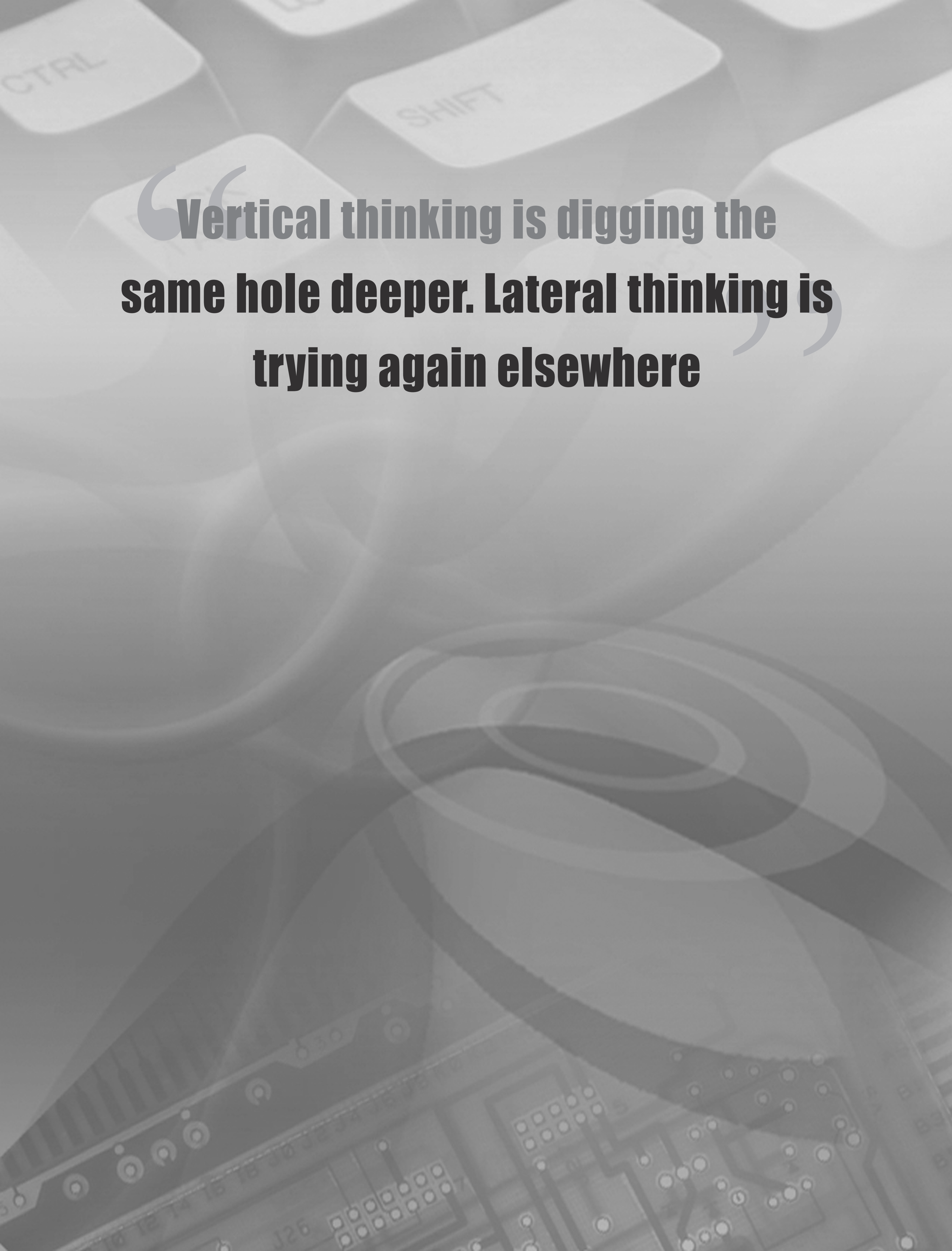
   f. Print the returned value.

   g. Code the function definition. Compare each character of the string with the specified character. Increment an integer variable whenever the specified character occurs in the string. Finally, return the integer variable to the `main()`.

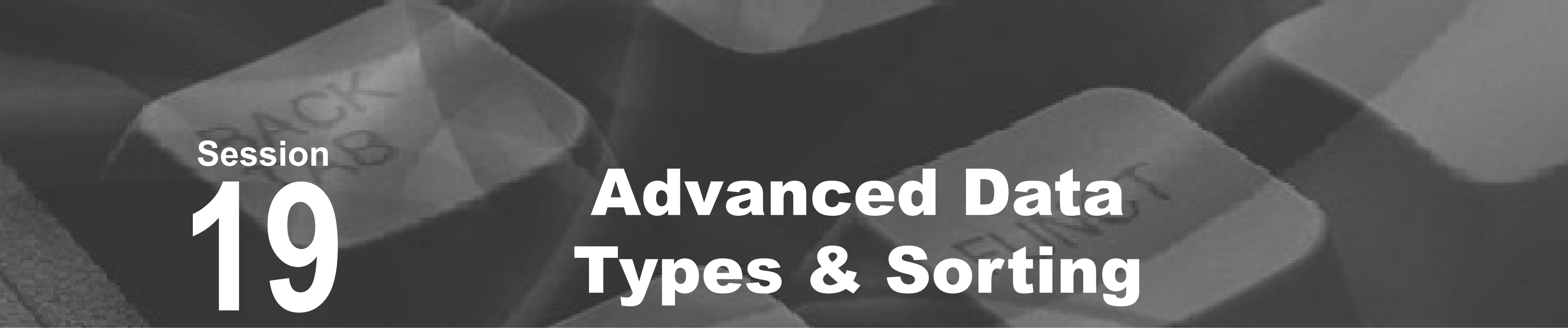
## Try It Yourself

1. Write a C Program to accept 5 names and a prefix. Insert the prefix at the beginning of each name in the array. Display the modified names.

2. Write a C Program to accept the average yearly temperature of the past five years, for five cities. Display the maximum and minimum temperature for each city. Use functions to determine the maximum and minimum temperatures.



“Vertical thinking is digging the same hole deeper. Lateral thinking is trying again elsewhere”



Session 19

# Advanced Data Types & Sorting

**Objectives**

**At the end of this session, you will be able to:**

- *Explain structures and their use*
- *Define structures*
- *Declare structure variables*
- *Explain how structure elements are accessed*
- *Explain how structures are initialized*
- *Explain how assignment statements are used with structures*
- *Explain how structures can be passed as arguments to functions*
- *Use arrays of structures*
- *Explain the initialization of structure arrays*
- *Explain pointers to structures*
- *Explain how structure pointers can be passed as arguments to functions*
- *Explain the typedef keyword*
- *Explain array sorting with the Insertion sort and Bubble sort methods*

## Introduction

Applications in the real world situations require different types of data to be stored. The predefined data types supported by C may prove inadequate in such situations. So, C allows custom data types to be created. One such data type is **structure**. A structure is a grouping of variables under one name. A data type can also be assigned a new name using the `typedef` keyword.

Applications store enormous amounts of data. In such cases, locating a particular data item can be very time consuming. Arranging the values in some sequence eases the situation. In this session, we will also look at some algorithms for sorting arrays.

# Session 19

## Advanced Data Types & Sorting

## 19.1 Structures

Variables can be used to hold one piece of information at a time and arrays can be used to hold several pieces of information of the same data type. However, a program may require operating upon data items of different types together as a unit. In this case, neither a variable nor an array is adequate.

For example, a program is written to store data on a catalog of books. The program requires that the name of each book (a character array), its author's name (another character array), the edition number (an integer), the price of the book (a float) be entered and stored. A multidimensional array cannot be used to do this, as an array must be of the same data type. This is where a structure makes the things simpler.

A structure consists of a number of data items, which need not be of the same type, grouped together. In the above example, a structure would consist of the book's name, the author name, the edition number, and the price of the book. The structure could hold as many of these items as desired.

### 19.1.1 Defining a Structure

A **structure definition** forms a template for creating structure variables. The variables in the structure are called **structure elements** or **structure members**.

Generally, the elements in a structure are logically related because they refer to a single entity. The book catalog example can be represented as follows:

```
struct cat
{
    char bk_name [25];
    char author [20];
    int edn;
    float price;
};
```

The above statement defines a new data type called `struct` **`cat`**. Each variable of this type consists of four elements - **`bk_ name`**, **`author`**, **`edn`**, and **`price`**. The statement does not declare any variable, and so it does not set aside any storage in memory. It just defines the structure of cat. The keyword `struct` tells the compiler that a structure is being defined. The tag **`cat`** is not a variable name, since a variable is not being declared. It is a **type name**. The elements of the structure are defined within braces, and a semicolon terminates the entire statement.

### 19.1.2 Declaring Structure Variables

Once the structure has been defined, one or more variables of that type can be declared. This can be done as follows:

```
struct cat books1;
```

This statement will set aside enough memory to hold all items in the structure. The above declaration performs a function similar to variable declarations like `int` **`xyz`** and `float` **`ans`**. It tells the compiler to set aside storage for a variable of specific type and assigns a name to the variable.

As with `int`, `float` and other data types there can be any number of variables of a given structure type. In a program, two variables **`books1`** and **`books2`** of the structure type **`cat`** can be declared. This can be done in several ways.

```
struct cat
{
    char bk_name[25];
    char author[20];
    int edn;
    float price;
} books1, books2;
```

**or**

```
struct cat books1, books2;
```

**or**

```
struct cat books1;
struct cat books2;
```

These declarations set aside memory for both **`books1`** and **`books2`**.

Individual structure elements are referenced through the use of the **dot operator (.)**, which is also known as the **membership operator**. The general syntax for accessing a structure element is:

```
structure_name.element_name
```

For example, the following code refers to the field **`bk_name`** of the structure variable **`books1`** declared earlier.

```
books1.bk_name
```

To read in the name of the book, the code would be:

```
scanf("%s", books1.bk_name);
```

To print the book name, the code would be:

```
printf("The name of the book is %s", books1.bk_name);
```

## 19.1.3 Initializing Structures

Like variables and arrays, structure variables can be initialized at the point of declaration. The format is similar to the one used to initialize arrays. Consider the following structure that stores the employee number and name:

```
struct employee
{
    int no;
    char name [20];
};
```

Variables **emp1** and **emp2** of the type **employee** can be declared and initialized as:

```
struct employee emp1 = {346, "Abraham"};
struct employee emp2 = {347, "John"};
```

Here, after the usual declaration of the structure type, the two structure variables **emp1** and **emp2** are declared and initialized. Their declaration and initialization happens at the same time through a single line of code. The initialization of a structure is similar to that of an array – the variable type, the variable name, and the assignment operator followed by braces enclosing a list of values, with the values separated by commas.

## 19.1.4 Assignment Statements used with Structures

It is possible to assign the values of one structure variable to another variable of the same type using a simple assignment statement. That is, if **books1** and **books2** are structure variables of the same type, the following statement is valid.

```
books2 = books1;
```

Also in cases, where direct assignment is not possible, the in-built function `memcpy()` can be used. The prototype for this function is

```
memcpy (char * destn, char &source, int nbytes);
```

This function copies **nbytes** starting from the address source to a block **nbytes** bytes starting from the address **destn**. The function requires the user to specify the size of the structure (**nbytes**), which can be obtained by the `sizeof()` operator. Using `memcpy()`, the contents of **`books1`** can be copied to **`books2`** as follows:

```
memcpy (&books2, &books1, sizeof(struct cat));
```

## 19.1.5 Structures within Structures

It is possible to have one structure within another structure. However, a structure cannot be nested within itself. Having one structure within another is many times a practical requirement. Consider an example, where, a record of the person borrowing the book and details of the books borrowed has to be maintained. The following structure can be used for the same.

```
struct issue
{
    char borrower [20];
    char dt_of_issue[8];
    struct cat books;
}issl;
```

This declares **`books`** to be a component of the structure **`issue`**. This component itself is a structure of type `struct` **`cat`**. The structure can be initialized as:

```
struct issue issl = { "Jane", "04/22/03", {"Illusions",
"Richard Bach", 2, 150.00}};
```

Nested braces are used to initialize a structure within a structure.

To access the elements of the structure the format will be similar to the one used with normal structures, that is to access name of borrower, the code will be

```
issl.borrower
```

However to access elements of the structure **`cat`**, which is a part of another structure **`issue`**, the following expression will be used:

```
issl.books.author
```

This refers to the element author in the structure books in the structure **`issl`**.

The level for nesting structures is restricted only by the availability of memory. It is possible to have a structure within a structure within a structure and so on. The variable names used are usually self descriptive of the form.

For example,

```
company.division.employee.salary
```

Also remember that if a structure is nested within another, it has to be declared prior to the structure that uses it.

## 19.1.6 Passing Structures as Arguments

A structure variable can be passed as an argument to a function. This is a useful facility and it is used to pass groups of logically related data items together instead of passing them one by one. However, when a structure is used as an argument, care should be taken that the type of the argument matches the type of the parameter.

For example, a structure is declared for storing the customer name, number and the principal amount deposited by the customer. The data is accepted in the `main()` and the structure is passed to a function **`intcal()`** that calculates the payable interest. The code is as follows:

**Example 1:**

```
#include <stdio.h>
void main()
{
    struct strucintcal/* Defines the structure */
    {
        char name[20];
        int numb;
        float amt;
    }xyz; /* Declares a variable */
    void intcal(struct strucintcal);
    clrscr();
    /* Accepts data into the structure */
    printf("\nEnter Customer name: ");
    gets(xyz.name);
    printf("\nEnter Customer number: ");
    scanf("%d",&xyz.numb);
    printf("\nEnter Principal amount: ");
    scanf("%f", &xyz.amt);
    intcal(xyz); /* Passes the structure to a function */
    getch();
}
```

```
void intcal(struct { char name[20];
                     int numb;
                     float amt;
                   } abc)
{
    float si, rate = 5.5, yrs = 2.5;
    /* Computes the interest */
    si = (abc.amt * rate * yrs) / 100;
    printf ("\nThe customer name is %s",abc.name);
    printf("\nThe customer number is %d",abc.numb);
    printf("\nThe amount is %f",abc.amt);
    printf("\nThe interest is %f",si);
    return;
}
```

A sample output of the program is given below.

```
Enter Customer name: Jane
Enter Customer number: 6001
Enter Principal Amount: 30000
The customer name is Jane
The customer number is 6001
The amount is 30000.000000
The interest is 4125.000000
```

It is possible to define a structure without a tag. This is useful when the variable is declared along with the structure definition itself. The tag is not needed in such situations.

## 19.1.7 Array of Structures

One of the most common uses of structures is in arrays of structures. To declare an array of structures, a structure is first defined, and then an array variable of that type is declared. For example, to declare an array of structures of the type **cat**, the statement would be:

```
struct cat books[50];
```

Like all array variables, arrays of structures begin indexing at 0. The array name followed by its subscript enclosed in square brackets stands for an element of that array. After the above declaration, this element is a structure by definition. So all the rules of referencing fields apply thereafter. After the structure array **books** is declared,

```
books[4].author
```

will refer to the variable author of the fourth element of the array **books**.

## 19.1.8 Initialization of Structure Arrays

An array of any type is initialized by enclosing the list of values of its elements within a pair of braces. This rule applies even when the elements are structures. The effective initialization contains nested braces. Consider the following example,

```
struct unit
{
    char ch;
    int i;
};
struct unit series [3] =
{
    {'a', 100}
    {'b', 200}
    {'c', 300}
};
```

This declares **series** to be an array of structures, each of the type **unit**. While initializing, each element is initialized as a structure. The whole list is then enclosed within braces to indicate that the array is being initialized.

## 19.1.9 Pointers to Structures

C supports pointers to structures but there are some special aspects to structure pointers. Like other pointers, structure pointers are declared by placing an asterisk(*) in front of the structure variable's name. For example, the following statement declares pointer **ptr_bk** of the structure type **cat**.

```
struct cat *ptr_bk;
```

Now to assign the address of structure variable books of type **cat**, the statement would be:

```
ptr_bk = &books;
```

The -> operator is used to access the elements of a structure using a pointer. This operator is a combination of the minus (-) and the greater than (>) signs and is also known as the **combination operator**. For example, the field **author** can be accessed in any of the following ways:

```
ptr_bk->author
```

**or**

```
books.author
```

**or**

```
(*ptr_bk).author
```

In the last expression, the parentheses are required because the period operator (.) has a higher precedence than the indirection (*) operator. Without the parentheses the compiler would generate an error, because **`ptr_bk`** (a pointer) is not directly compatible with the period operator.

As with all pointer declarations, the declaration of a pointer allocates space for the pointer and not what it points to. So, when a structure pointer is declared, space is allocated for the address of the structure and not for the structure itself.

### 19.1.10 Structure Pointers as Arguments

Structure pointers are useful as arguments to functions. At the time of calling the function, a pointer to the structure or the explicit address of a structure variable is passed to the function. This enables the function to modify the structure elements directly.

## 19.2 The typedef keyword

A new data type name can be defined by using the keyword `typedef`. This does not create a new data type, but defines a new name for an existing type. The general syntax of the `typedef` statement is:

```
typedef type name;
```

where **`type`** is any allowable data type and **`name`** is the new name for this type.

The new name defined, is in addition to, and not a replacement for, the existing data type. For example, a new name for `float` can be created in the following way:

```
typedef float deci;
```

This statement will tell the compiler to recognize **`deci`** as another name for `float`. A `float` variable can be declared using **`deci`** as shown below:

```
deci amt;
```

Here, **amt** is a floating point variable of type **deci**, which is another name for float. After it has been defined, **deci** can be used as a data type in a **typedef** statement to assign another name to float. For example,

```
typedef deci point;
```

The above statement tells the compiler to recognize point as another name for **deci**, which is another name for float. The typedef feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. As a result, the structure can be referenced more concisely. In addition, the name given to a user defined structure type often suggests the purpose of the structure within the program. In general terms, a user-defined structure can be written as:

```
typedef struct new_type
{
    type var1;
    type var2;
}
```

Here, **new_type** is the user-defined structure type and not a structure variable. Structure variables can now be defined in terms of the new data type. An example is:

```
typedef struct
{
    int day;
    int month;
    int year;
} date;
date due_date;
```

Here, **date** is the new data type and **due_date** is a variable of type **date**.

Remember that **typedef** cannot be used with storage classes.

## 19.3 Sorting Arrays

Sorting means arranging the array data in a specified order such as ascending or descending. Data in an array is easier to search when the array is sorted.

There are several methods for sorting arrays. We will examine the following two methods:

- Bubble Sort
- Insertion Sort

### 19.3.1 Bubble Sort

The name of this sorting process describes the way it works. Here, the comparisons begin from the bottom-most element and the smaller element bubble up towards the top. The process of sorting a 5-element array in ascending order is given below:

- The value in the 5th element is compared against the value in the 4th element.
- If the value in the 5th element is smaller than the value in the 4th, the values in the two elements are swapped.
- Next, the value in the 4th element is compared against the value in the 3rd, and in a similar way, the values are swapped if the value in the lower element is found to be greater than that in the upper element.
- The value in the 3rd element is compared against the value in the 2nd, and this process of comparing and swapping continues.
- At the end of one pass, the smallest value reaches the first element. In symbolic terms, it can be stated that the smallest value has ‘bubbled’ up.
- In the next pass, the comparing starts off again with the lowest element, and works its way up to the 2nd element. Since the first element already contains the smallest value, it does not need to be compared.

In this way, at the end of the sorting process, the smaller elements bubble up towards the top, while the bigger values sink down. Figure 19.1 illustrates the bubble sort method.

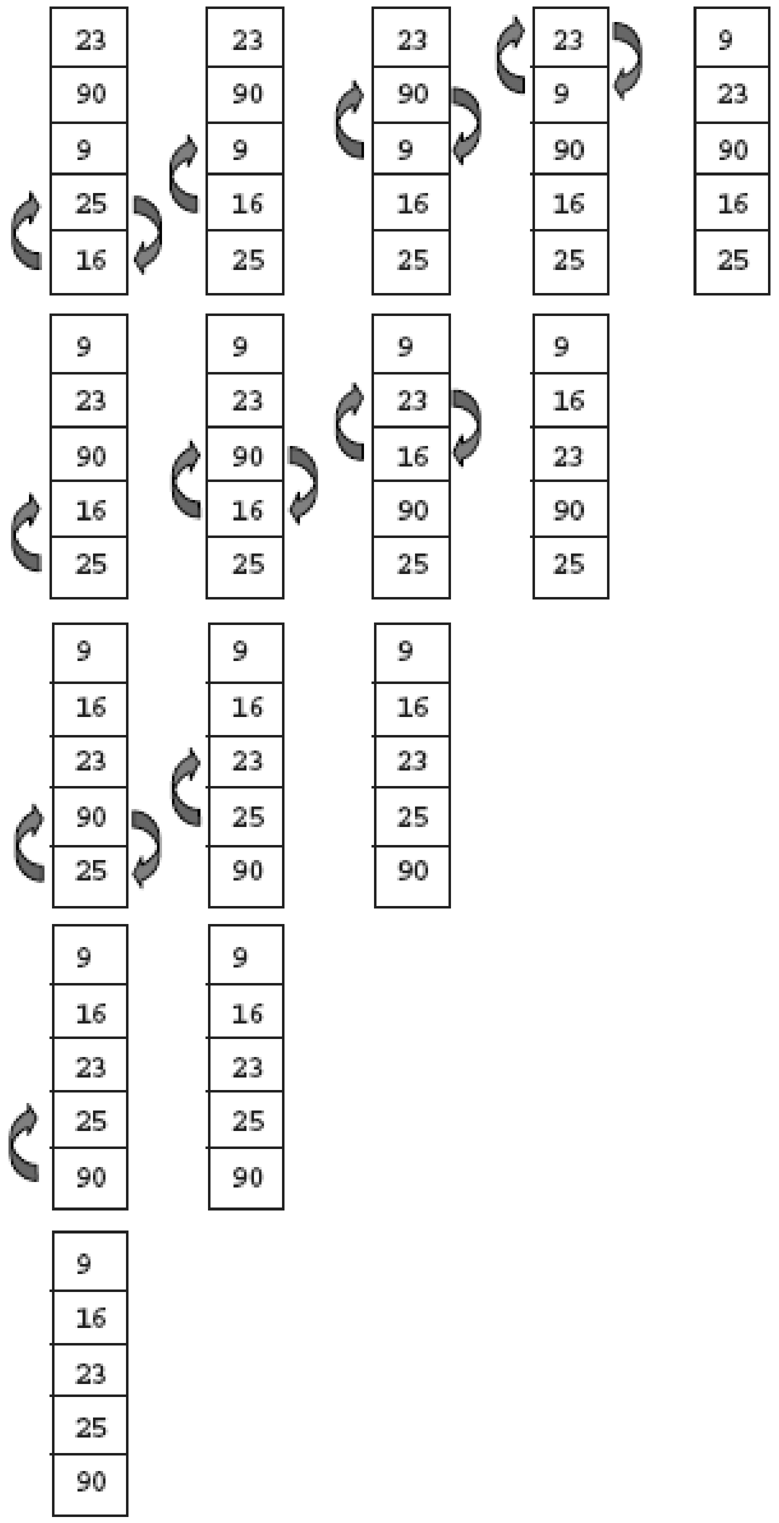


**Figure 19.1: Bubble Sort**

The program to perform the bubble sort is given here.

**Example 2:**

```
#include <stdio.h>
void main()
{
    int i, j, temp, arr_num[5] = { 23, 90, 9, 25, 16};
    clrscr();
    for(i=3;i>=0;i—) /* Tracks every pass */
        for(j=4;j>=4-i;j--) /* Compares elements */
        {
            if(arr_num[j]<arr_num[j-1])
            {
                temp=arr_num[j];
                arr_num[j]=arr_num[j-1];
                arr_num[j-1]=temp;
            }
        }
    printf(”\nThe sorted array”);
    for(i=0;i<5;i++)
    printf(”\n%d”, arr_num[i]);
    getch();
}
```

## 19.3.2 Insertion Sort

In the Insertion sort method, each element in the array is examined, and put into its proper place among the elements that have already been sorted. When the last element is put into its proper position, the array is sorted. For example, considering that an array has 5 elements,

- The value in the 1st element is assumed to be in sorted order.
- The value in the 2nd element is compared with the sorted part of the array that currently contains only the 1st element.
- If the value in the 2nd element is smaller, it is inserted before the 1st element. Now, the first two elements form the sorted list and the remaining form the unsorted list.
- The next element from the unsorted list, element 3, is then compared with the sorted list.
- If the value in the 3rd element is smaller than the 1st element, the value in the 3rd element is inserted before the 1st element.

- Else, if the value in the 3rd element is smaller than the 2nd element, the value in the 3rd element is inserted before the 2nd element. Now, the sorted part of the array contains 3 elements while the unsorted part contains 2 elements.

- The process of comparing the elements in the unsorted part with those in the sorted part continues till the last element of the array has been compared.

At the end of the sort process, each element has been inserted in its proper location. Figure 19.2 illustrates the working of insertion sort.

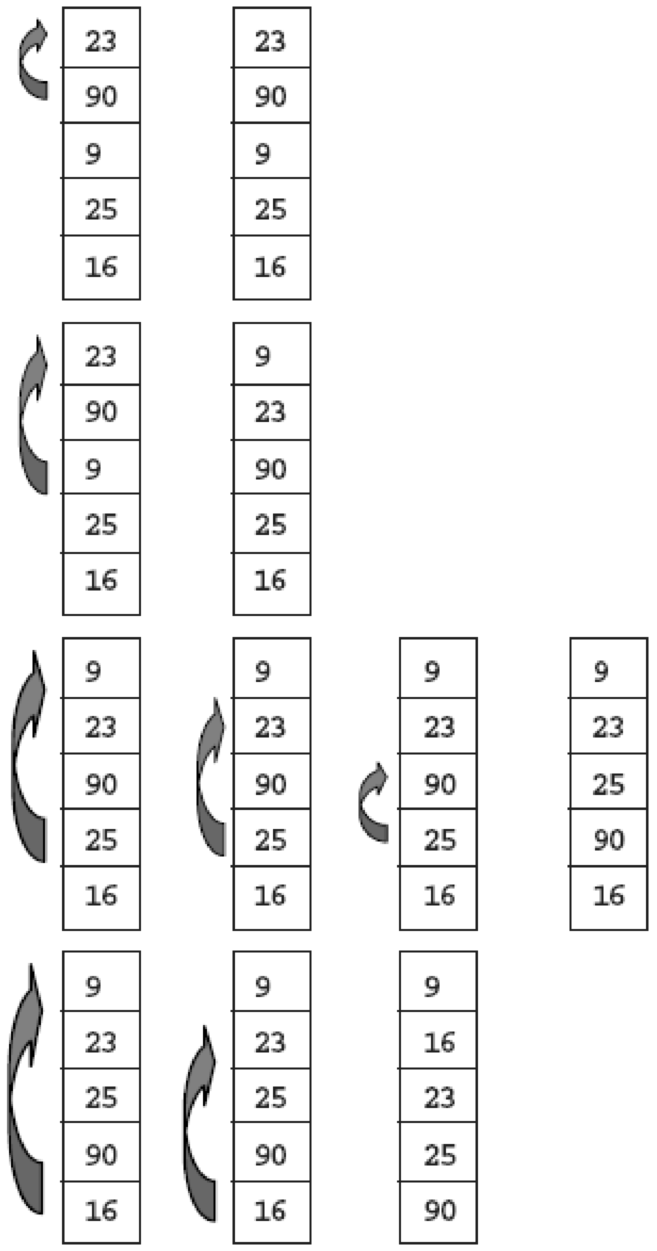


**Figure 19.2: Insertion Sort**

The program to perform the insertion sort is given below.

**Example 3:**

```
#include<stdio.h>
void main()
{
    int i, j, arr[5] = { 23, 90, 9, 25, 16 };
    char flag;
    clrscr();
    /*Loop to compare each element of the unsorted part of the array*/
    for(i=1; i<5; i++)
    /*Loop for each element in the sorted part of the array*/
        for(j=0, flag='n'; j<i && flag=='n'; j++)
        {
            if(arr[j]>arr[i])
            {
                /*Invoke the function to insert the number*/
                insertnum(arr, i, j);
                flag='y';
            }
        }
    printf("\n\nThe sorted array\n");
    for(i=0; i<5; i++)
        printf("%d\t", arr[i]);
    getch();
}
insertnum(int arrnum[], int x, int y)
{
    int temp;
    /*Store the number to be inserted*/
    temp=arrnum[x];
    /*Loop to push the sorted part of the array down from the position*/
    /*where the number has to inserted*/
    for(; x>y; x--)
        arrnum[x]=arrnum[x-1];
    /*Insert the number*/
    arrnum[x]=temp;
}
```

## Summary

- A structure is a grouping of variables of different data types under one name.
- A structure definition forms a template that may be used to create structure variables.
- Individual structure elements are referenced using the dot operator (.), which is also known as the membership operator.
- Values of one structure variable can be assigned to another variable of the same type using a simple assignment statement.
- It is possible to have one structure within another structure. However a structure cannot be nested within itself.
- A structure variable can be passed as an argument to another function.
- The most common implementation of structures is in the form of arrays of structures.
- The -> operator is used to access the elements of a structure using a pointer to that structure.
- A new data type name can be defined by using the keyword typedef.
- Two techniques for sorting an array are bubble sort and insertion sort.
- In bubble sort, the values of the elements are compared with the value in the adjacent element. In this method, the smaller elements bubble up, and at the end the array is sorted.
- In insertion sort, each element in the array is examined, and inserted into its proper place among the elements that have already been sorted.

## Check Your Progress

1. A __________ groups together a number of data items, which need not be of the same data type.

2. Individual structure elements are referenced through the use of the __________.

3. Values of one structure variable can be assigned to another variable of the same type using a simple assignment statement. **(True / False)**

4. It is impossible to have one structure within another structure. **(True / False)**

5. A new data type name can be defined by using the _________keyword.

6. In bubble sort, the _________elements are compared.

7. In insertion sort, if an unsorted element has to be put in a particular sorted location, values are swapped. **(True / False)**

## Try It Yourself

1. Write a C program to implement an inventory system. Store the item number, name, rate and quantity on hand in a structure. Accept the details for five items into a structure array and display the item name and its total price. At the end, display the grand total value of the inventory.

2. Write a C program to store the names and scores of 5 students in a structure array. Sort the structure array in descending order of scores. Display the top 3 scores.