UNIT-III

POINTER VARIABLE:

Definition:-A pointer is a variable which stores the address of another variable. A pointer provides access to a variable by using the address of that variable. Thus, a pointer will have direct access to the address of the memory location. Like any other variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is

Syntax:- **data_type *pointer_name;**

Here, **data_type** is the data type of the value that the pointer will point to. **Pointer_name** is any valid user defined variable name. For example

int *pnum;

char *pch;

float *pfnum;

In each of the above statements, a pointer variable is declared to point to the specified data type. Although all these pointers, **pnum, pch,** and **pfnum** point to different data types but they will occupy the same amount of space in memory. But how much space they will occupy will depend on the platform where the code is going to run. To verify this, execute the following code and observe the result.

```
#include<stdio.h>
main()
{
int   *pnum;
char   *pch;
float  *pfnum;
double *pdnum;
printf("\n size of integer pointer = %d\n", sizeof(pnum));
printf("\n size of character pointer = %d\n", sizeof(pch));
printf("\n size of float pointer = %d\n", sizeof(pfnum));
printf("\n size of double pointer = %d\n", sizeof(pdnum));
}
```

Output:

```
size of integer pointer = 4
size of character pointer = 4
size of float pointer = 4
size of double pointer = 4
```

Note: The size occupied by any pointer variable will depends on the platform where the code is going to run.

POINTER OPERATORS:

There are two pointer operators one is * and another is **&**. These are unary operators (Remember a unary operator requires only one operand). The **&** operator returns the memory address of its operand and the second operator * returns the value located at the address.

Pointer operators:

Operator	**Operator Name**	**Purpose**
*	Value at Operator	Gives Value stored at Particular address
&	Address Operator	Gives Address of Variable

In order to create pointer to a variable we use "*" operator and to find the address of variable we use "&"operator. Don't Consider "&" and "*" operators as Logical AND and Multiplication Operator in Case of Pointers.

Important points to remember:

1. '&' operator is called as **address Operator**
2. **'*'** is called as **'Value at address'** Operator
3. '**Value at address'** Operator gives 'Value stored at Particular address.
4. **'Value at address'** is also called as **'Indirection Operator'**

Write a C program to demonstrate the usage of pointer operators.

```
#include<stdio.h>

int main()
{
int n = 20;
printf("\nThe address of n is %u",&n);
printf("\nThe Value of n is %d",n);
printf("\nThe Value of n is %d",*(&n));
}
```

Output:

```
The address of n is 1002

The Value of n is 20

The Value of n is 20
```

How *(&n) is same as printing the value of n ?

&n	Gives address of the memory location whose name is ‘n’
*	means value at Operator gives value at address specified by &n.
m = &n	Address of n is stored in m , but remember that m is not ordinary variable like ‘n’

So, the C compiler must provide space for it in memory. Below is Step by Step Calculation to compute the value

```
m  =  * ( &n   )
   =  * ( Address of Variable 'n')
   =  * ( 1000 )
   =  Value at Address 1000
   =  20
```

POINTER EXPRESSIONS:

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions, such as ***assignments, conversions,*** and ***arithmetic.***

(i) Pointer Assignment:

You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward.

For example:

```
#include <stdio.h>
int main(void)
{
int x = 99;
int *p1, *p2;
p1 = &x;
p2 = p1;
/* print the value of x twice */
printf("Values at p1 and p2: %d %
d\n", *p1, *p2);
/* print the address of x twice */
printf("Addresses pointed to by p1 and p2: %p %p", p1, p2);
return 0;
}
```

After the assignment sequence

```
p1 = &x;
p2 = p1;
```

p1 and **p2** both point to **x**. Thus, both **p1** and **p2** refer to the same object. Sample output from the program, which confirms this, is shown here.

```
Values at p1 and p2: 99 99
Addresses pointed to by p1 and p2: 0063FDF0 0063FDF0
```

(ii) Pointer Conversion: It is also possible to assign a pointer of one data type to a pointer of another type. However, doing so involves a pointer conversion.

Thus a pointer conversion is nothing but converting a pointer of one data type into a pointer of another data type. Here pointer conversion can done this by using void pointer (void *).

Void pointer is also called ***generic pointer***. A generic pointer is a pointer that has **void** as its data type. The void pointer or the generic pointer is a special type of pointer that can be used to point to variables of any data type. It is declared like a normal pointer variable but using the void keyword as the pointer's data type.
For example

```
#include <stdio.h>
int main(void)
{
int x=10;
char ch='A';
void *gp;
gp=&x;
printf("\n Generic pointer points to the integer value =%d", *(int*)gp);
gp=&ch;
printf("\n Generic pointer points to the character value =%c", *(char*)gp);
return 0;
}
```

Output:

```
Generic pointer points to the integer value =10
Generic pointer points to the integer value =A
```

(iii) Pointer Arithmetic:

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer

ptr++

After the above operation, the **ptr** will point to the location 1004 because each time **ptr** is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array

```
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;

   for ( i = 0; i < MAX; i++) {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the next location */
      ptr++;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below

```
#include <stdio.h>
const int MAX = 3;
int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = &var[MAX-1];

  for ( i = MAX; i > 0; i--) {

    printf("Address of var[%d] = %x\n", i-1, ptr );
    printf("Value of var[%d] = %d\n", i-1, *ptr );

    /* move to the previous location */
    ptr--;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

```
#include <stdio.h>

const int MAX = 3;
```

```
int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have address of the first element in pointer */
  ptr = var;
  i = 0;

  while ( ptr <= &var[MAX - 1] ) {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* point to the previous location */
    ptr++;
    i++;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Rules for pointer operation:

The following rules apply when performing operations on pointer variables

- A pointer variable can be assigned to address of another variable.
- A pointer variable can be assigned the values of other pointer variables.
- A pointer variable can be initialized with NULL or 0 values.
- A pointer variable can be prefixed or post fixed with increment and decrement operator.
- An integer value may be added or subtracted from a pointer variable.
- When two pointers points to the same array, one pointer variable can be subtracted from another.
- When two pointers points to the objects of same data types, they can be compared using relational
- A pointer variable cannot be multiple by a constant.
- Two pointer variables cannot be added.
- A value cannot be assigned to an arbitrary address.

POINTER AND ARRAYS:

There is a close relationship between pointers and arrays in C, let us start: An array name is a constant pointer pointing to the first element of the array. Therefore, in the declaration

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance**

```
double *p;
double balance[10];

p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, ***(balance + 4)** is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in 'p', you can access the array elements using ***p**, ***(p+1)**, ***(p+2)** and so on. Given below is the example to show all the concepts discussed above −

```
#include <stdio.h>

int main () {

  /* an array with 5 elements */
  double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
  double *p;
  int i;

  p = balance;

  /* output each array element's value */
  printf( "Array values using pointer\n");

  for ( i = 0; i < 5; i++ ) {
    printf("*(p + %d) : %f\n",  i, *(p + i) );
  }

  printf( "Array values using balance as address\n");

  for ( i = 0; i < 5; i++ ) {
    printf("*(balance + %d) : %f\n",  i, *(balance + i) );
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

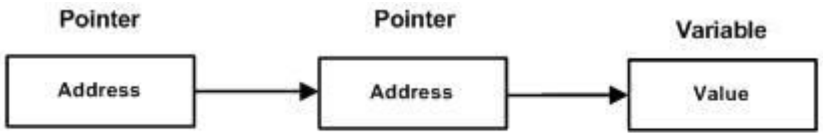
```
Array values using pointer
*(p + 0) : 1000.000000
*(p + 1) : 2.000000
*(p + 2) : 3.400000
*(p + 3) : 17.000000
*(p + 4) : 50.000000

Array values using balance as address
*(balance + 0) : 1000.000000
*(balance + 1) : 2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000
*(balance + 4) : 50.000000
```

In the above example, **p** is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in **p**, ***p** will give us the value available at the address stored in **p**, as we have shown in the above example.

POINTER TO POINTER (MULTIPLE INDIRECTION):

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

```
#include <stdio.h>

int main ()
{
  int  var;
  int  *ptr;
  int  **pptr;

  var = 3000;
  /* take the address of var */
  ptr = &var;

  /* take the address of ptr using address of operator & */
  pptr = &ptr;

  /* take the value using pptr */
  printf("Value of var = %d\n", var );
  printf("Value available at *ptr = %d\n", *ptr );
  printf("Value available at **pptr = %d\n", **pptr);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

INITIALIZING POINTERS:

If a pointer is declared and before it has been assigned a value, it contains an unknown value. If you want to use the pointer before giving it a valid value, you will probably crash your program— and possibly your computer's operating system as well— a very nasty type of error!

There is an important convention that most C programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is given the value null (which is zero). Null is used because C guarantees that no object will exist at the null address. Thus, any pointer that is null implies that it points to nothing and should not be used.

One way to give a pointer a null value is to assign zero to it. For example, the following initializes **p** to null.

char *p = 0;

Additionally, many of C's headers, such as **<stdio.h>** , define the macro **NULL**, which is a null pointer constant. Therefore, you will often see a pointer assigned null using a statement Such as this:

p = NULL;

Note: NULL pointer is a pointer which does not point to any valid memory address.

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program

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

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

POINTERS TO FUNCTIONS:

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. A function pointer is a variable that stores the address of a function that can later be called through that function pointer. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>
// A normal function with an int parameter and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}


int main()
{
   // fun_ptr is a pointer to function fun()
   void (*fun_ptr)(int) = &fun;

   /* The above line is equivalent of following two lines
      void (*fun_ptr)(int);
      fun_ptr = &fun;
   */

   // Invoking fun() using fun_ptr
   (*fun_ptr)(10);

   return 0;
}
```

Output:

```
Value of a is 10
```

<u>/*swapping of 2 numbers using third variable using pointers*/</u>

```
# include < stdio.h>
void swap (int *, int *);
main ( )
{
int a=10, b=20;
swap(&a, &b); /* a,b are actual parameters */
printf (“a= %d \t b=%d ”, a ,b);
}
void swap (int *x , int *y) /* x, y are formal parameters */
{
int t;
t = *x;
*x=*y;
*y=t;
}
```

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
3) A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing *, the program still works.

```
#include <stdio.h>
// A normal function with an int parameter and void return type
void fun(int a)
```

```
{
   printf("Value of a is %d\n", a);
}

int main()
{
   void (*fun_ptr)(int) = fun;  // here & symbol is removed

   fun_ptr(10);  // here * symbol is  removed

   return 0;
}
```

Run on IDE

Output:

```
Value of a is 10
```

4) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

C'S DYNAMIC ALLOCATION FUNCTIONS:

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under **"stdlib.h"** for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer to first byte of allocated memory
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to first byte of allocated memory
free()	De-allocates the previously allocated space
realloc()	Changes the size of previously allocated space

malloc()

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

Syntax of malloc()

```
ptr=(cast-type*)malloc(byte-size);
```

Here, **ptr** is pointer of cast-type. The **malloc()** function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns **NULL** pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of **int** either 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

Calloc()

The name calloc stands for “contiguous allocation”. The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of **n** elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

Free()

When a variable is allocated space during the compile time, then the memory used by that variable is automatically released by the system in accordance with its storage class. But when we dynamically allocate memory then it is our responsibility to release the space when it is not required. Therefore, Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

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

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

realloc()

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

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, **ptr** is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    for(i=0;i<n2;++i)
        printf("%u\t",ptr+i);
    return 0;
}
```

S.NO.	MALLOC	CALLOC
1	It allocates only single block of requested memory.	It allocates multiple blocks of requested memory.

2	int *ptr;
	ptr = malloc(20 * sizeof(int));
	For the above, 20*4 bytes of memory only allocated in one block.
	Total = 80 bytes
		int *ptr;
		Ptr = calloc(20, 20 * sizeof(int));
		For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes

3	malloc () doesn’t initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
4	type cast must be done since this	Same as malloc () function

	function returns void pointer
	int *ptr;
	ptr=(int*)malloc(sizeof(int)*20);
		int *ptr;
		ptr=(int*)calloc(20,20*sizeof(int));

PROBLEMS WITH POINTERS:

Nothing will get you into more trouble than a wild pointer! Pointers are a mixed blessing. They give you tremendous power, but when a pointer is used incorrectly, or contains the wrong value, it can be a very difficult bug to find.

1. Uninitialized pointers might cause segmentation fault
2. If pointers are updated with incorrect values, it might lead to memory corruption.
3. **Dangling Pointer in C**

    a. Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory.
    b. In short pointer pointing to non-existing memory location is called dangling pointer.

Problem : If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as **dangling pointer** and this problem is known as **dangling pointer problem**.

Dangling pointer occurs in 3 cases.

- De-allocation of memory
- Function Call
- Variable goes out of scope

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
   int *ptr = (int *)malloc(sizeof(int));

   // After below free call, ptr becomes a dangling pointer
   free(ptr);
    // No more a dangling pointer
   ptr = NULL;
}
```

4. Memory Leak

Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate. To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```
/* Function with memory leak */
#include <stdlib.h>
 void f()
{
  int *ptr = (int *) malloc(sizeof(int));
   /* Do some work */
   return; /* Return without freeing ptr*/
}
```

```
/* Function without memory leak */
#include <stdlib.h>;
 void f()
{
  int *ptr = (int *) malloc(sizeof(int));
   /* Do some work */
   free(ptr);
  return;
}
```

FUNCTIONS IN C

Definition: A function is a self-contained block of code that carries out some specific and well-defined task.

TYPES OF FUNCTIONS PRESENT IN C

C functions are classified into two types. They are as follows

1. Library Functions (or) pre-defined functions
2. User Defined Functions

1. Library Functions

These are the built-in functions, available in standard library of C. The standard C library is collection various types of functions which perform some standard and predefined tasks. These library functions are also called as pre-defined functions.

Example: **abs (a)** function gives the absolute value of a, available in <math.h> header file

pow (x, y) function computes x power y. available in <math.h> header file

printf ()/scanf () performs I/O functions. Etc..,

2. User Defined Functions

These functions are written by the programmer (user) to perform some specific tasks.

Example: **main (), sum(), fact()** etc.

The Major distinction between these two types is that library functions are not required to be written by us (user). Whereas a user defined function has to be written (or developed) by the user at the time of writing a program.

USER-DEFINED FUNCTIONS IN C

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

ADVANTAGES OF USER-DEFINED FUNCTIONS

1. **Modular Programming** It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. **Reduction of source code** The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.
3. **Easier Debugging** It is easy to locate and isolate a faulty function for further investigation.
4. **Code Reusability** a program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.
5. **Function sharing** Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

THE SYNTAX (OR) GENERAL FORM OF A C FUNCTION

```
return_type        function_name (argument declaration)
{
   //local declarations
     ……
     ……
   //statements
      ……
   return (expression);
 }
```

Figure: 3.2 General Form of A C Function

return-type

Specifies the type of value that a function returns using the return statement. It can be any valid data type. **If no data type is specified the function is assumed to return an integer result**.

function-name

Must follow same rules of variable names in C. No two functions have the same name in a C program.

argument declaration

Arguments are, a comma-separated list of variables that receive the values of the argument when function is called. **If there are no argument declaration the bracket consists of keyword void.**

THE TERMINOLOGY USED IN C FUNCTIONS

A C function name is used three times in a program

1. for function declaration
2. in a function call
3. for function definition.

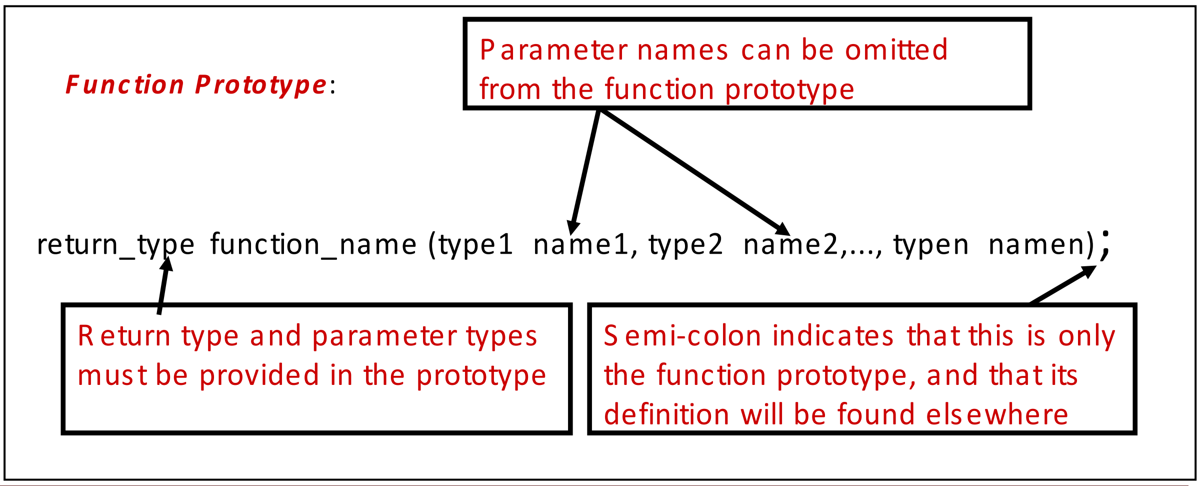
➔FUNCTION DECLARATION (OR) PROTOTYPE

The ANSI C standard expands the concept of forward function declaration. This expanded declaration is called a function prototype.

A function prototype performs two special tasks.

1. First it identifies the return type of the function so that the compiler can generate the correct code for the return data.
2. Second, it specifies the type and number of arguments used by the function.

The general form of the prototype is


Function Prototype:
Parameter names can be omitted from the function prototype
return_type function_name (type1 name1, type2 name2,..., typen namen);
Return type and parameter types must be provided in the prototype
Semi-colon indicates that this is only the function prototype, and that its definition will be found elsewhere

Note: The prototype normally goes near the top of the program and must appear before any call is made to the function.

➔THE FUNCTION CALL

A function call is a postfix expression. The operand in a function is the function name. The operator is the parameter lists (…), which contain the actual parameters.

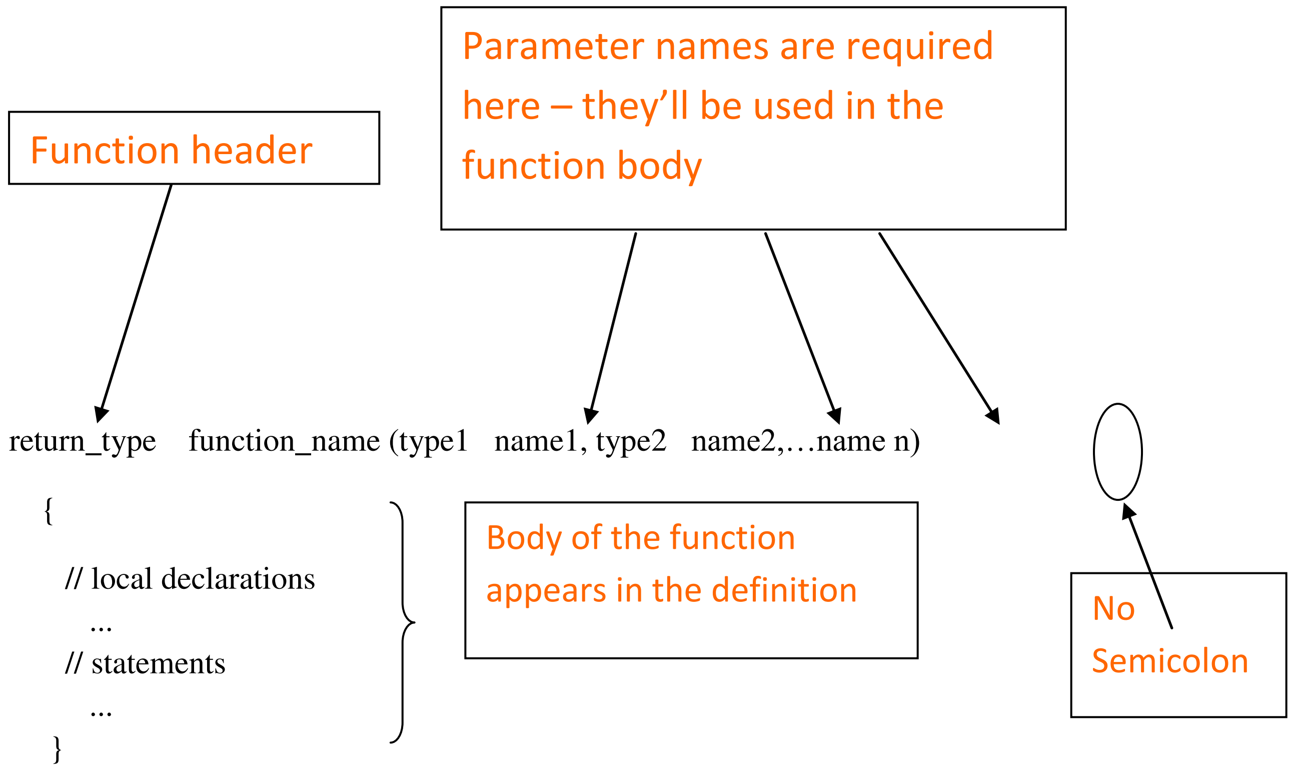
Example:

```
        void main ()
    {
      sum (a, b);
    }
```

When the compiler encounters the function call ,the control is transferred to the function sum().The function sum() is executed line by line and outputs the sum of the two numbers and returns to main when the last statement in the following has executed and conceptually, the function's ending } is encountered.

➔FUNCTION DEFINITION

The function definition contains the code for a function. It is made up of two parts: **The function header and the function body**, the compound statement is must.



```
return_type    function_name (type1   name1, type2   name2,…name n)
   {
     // local declarations
        …
     // statements
        …
   }
```

Function header
Parameter names are required here – they'll be used in the function body
Body of the function appears in the definition
No Semicolon

Figure: Function Definition

- **Function header consists of three parts: the return type, the function name, and the formal parameter list.**

- **function body contains local declarations and function statements and return statement.**

- Variables can be declared inside function body.

- A Function cannot be defined inside another function.

CATEGORIES OF USER -DEFINEDFUNCTIONS

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

Category 1: Functions with no arguments and no return values

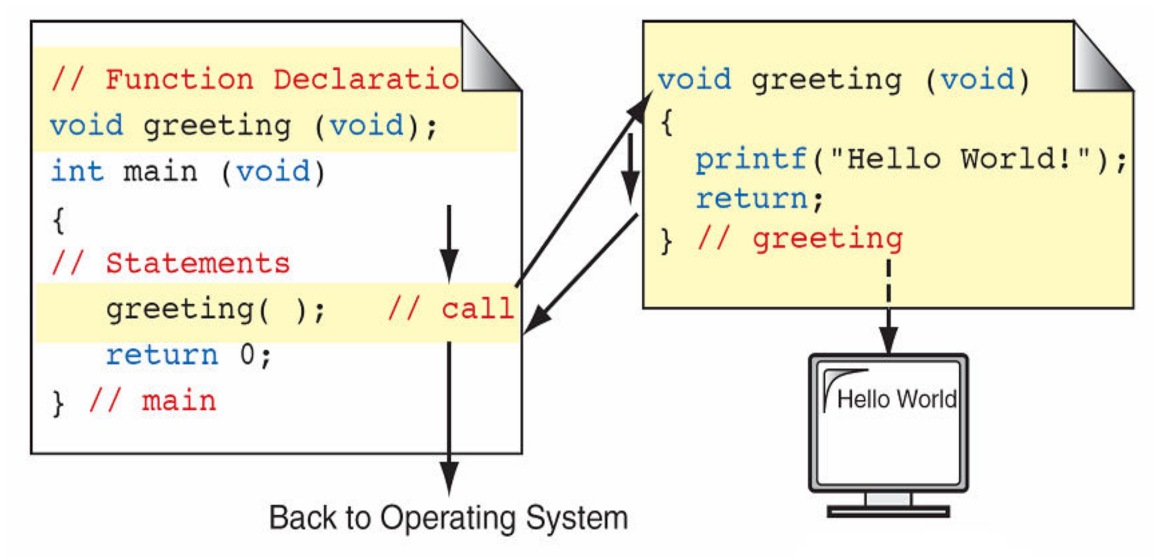
Category 2: Functions with no arguments and return values

Category 3: Functions with arguments and no return values

Category 4: Functions with arguments and return values

FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUES

This type of function has no arguments, meaning that it does not receive any data from the calling function. Similarly this type of function will not return any value. Here the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function.



```
// Function Declaratio
void greeting (void);
int main (void)
{
// Statements
   greeting( );    // call
   return 0;
} // main
```

```
void greeting (void)
{
   printf("Hello World!");
   return;
} // greeting
```

Figure: Functions with no arguments and no return values

Observe from the above figure that the function **greeting()** do not receive any values from the function **main()** and it does not return any value to the function **main()**. Observe the transfer of control between the functions indicated with arrows.

//C program to find sum of two numbers using functions with no arguments and no return values

```
#include<stdio.h>
void sum ();
void main ()
{
  clrscr ();
  sum ();    /*calling function */
  getch ();
}
void sum ()
{
  int x, y, z;
  printf ("\n Enter the values of x and y:  ");
  scanf ("%d%d", &x, &y);
  z=x+y;
  printf ("\n The sum = %d",z);
}
```

FUNCTIONS WITH NO ARGUMENTS AND RETURN VALUES

There are two ways that a function terminates its execution and returns to the caller.

1. When the last statement in the function has executed and conceptually the function's ending '}' is encountered.

2. Whenever it faces return statement.

THE RETURN STATEMENT

- ❖ The return statement is the mechanism for returning a value from the called function to its caller.

The general form of the return statement is

return expression;

❖ The calling function is free to ignore the returned value. Further more, there need not be expression after the return.

❖ In any case if a function fails to return a value, its value is certain to be garbage.

❖ **The return statement has two important uses**

1. It causes an immediate **exit of the control** from the function. That is ,it causes program execution to return to the calling function.

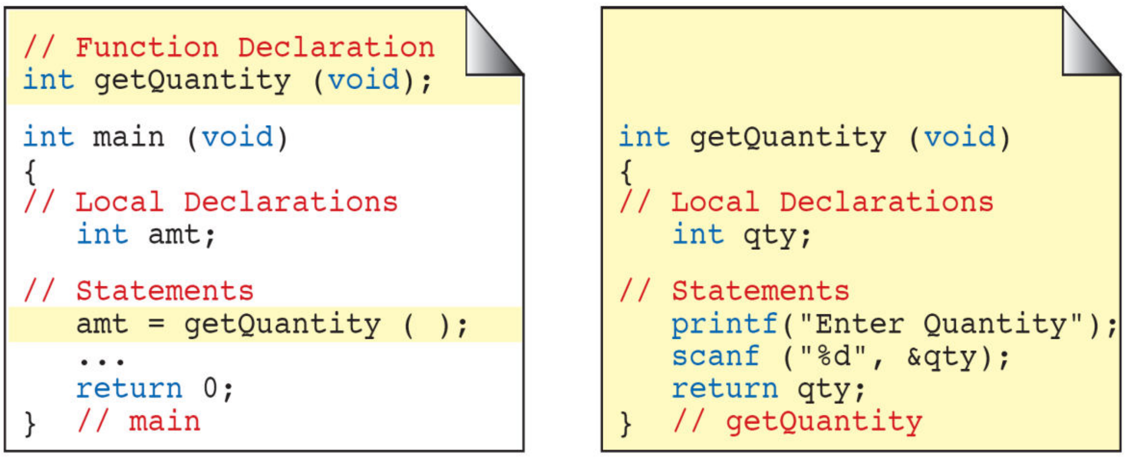
2. It returns the value present in the expression.

example: return(x+y);

return (6*8);

return (3);

return;



```
// Function Declaration
int getQuantity (void);

int main (void)
{
// Local Declarations
   int amt;

// Statements
   amt = getQuantity ( );
   ...
   return 0;
}  // main
```

```
int getQuantity (void)
{
// Local Declarations
   int qty;

// Statements
   printf("Enter Quantity");
   scanf ("%d", &qty);
   return qty;
}  // getQuantity
```

Figure: Functions with no arguments and return values

In this category, there is no data transfer from the calling function to the called function. But, there is data transfer from called function to the calling function.

In the above example, observe from the above figure that the function **getQuantity()** do not receive any value from the function **main()**. But, it accepts data from the keyboard, and returns the value to the calling function.

// C program to find sum of two numbers using functions with no arguments and return values

```
#include<stdio.h>
int sum ();
void main ()
{
  int c;
  clrscr ();
  c=sum ();    /*calling function */
  printf ("\n The sum = %d", c);
  getch ();
}
int sum ()
{
  int x, y;
  printf ("\n Enter the values of x and y:  ");
  scanf ("%d%d", &x, &y);
  return x+y;
}
```

RETURNING VALUES FROM MAIN()

When we use a return statement in main (), some program returns a termination code to the calling process (usually to the Operating System).**The return values must be an integer**.

For many Operating Systems, a return value of **'0'** indicates that the program terminated normally. All other values indicate that some error occurred. For this reason, it is good practice to use an explicit return statement.

FUNCTIONS WITH ARGUMENTS AND NO RETURN VALUES

In this category there is data transfer from the calling function to the called function using parameters. But, there is no data transfer from called function to the calling function.

Local Variables

Variables that are defined within a function are called local variables.A local variable comes into existence when the function is entered and is destroyed upon exit.

Function arguments

The arguments that are supplied in to two categories

1. actual arguments/parameters

2. formal arguments/parameters

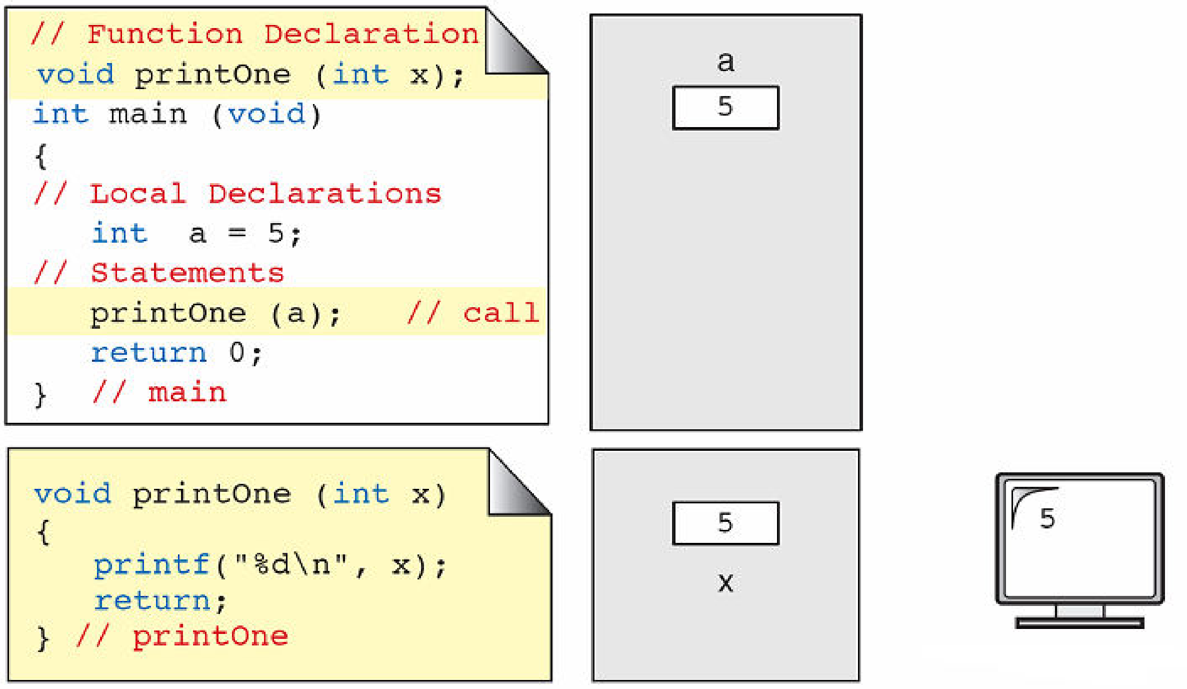
Actual arguments/parameters

Actual parameters are the expressions in the calling functions. These are the parameters present in the calling statement (function call).

Formal arguments/parameters

Formal parameters are the variables that are declared in the header of the function definition. This list defines and declares that will contain the data received by the function. These are the value parameters; copies of the values being passed are stored in the called functions memory area.

Note: **Actual and Formal parameters must match exactly in type, order, and number. Their names however, do not need to match.**



```
// Function Declaration
void printOne (int x);
int main (void)
{
// Local Declarations
   int  a = 5;
// Statements
   printOne (a);    // call
   return 0;
}  // main
```

```
void printOne (int x)
{
   printf("%d\n", x);
   return;
} // printOne
```

Figur: Functions with arguments and no return values

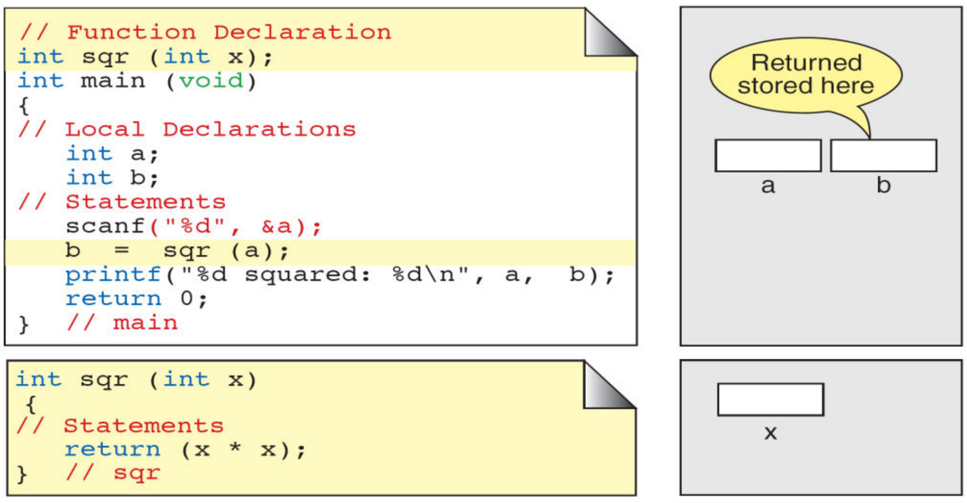
Observe from the above figure, the function **printOne()** receives one value from the function **main** (), display the value copy of a.

//C program to find sum of two numbers using functions with arguments and no return values

```
#include<stdio.h>
void sum (int ,int );
void main ()
{
  int a, b;
  printf (“\n Enter the values of a and b:  “);
  scanf (“%d%d”, &a, &b);
  sum (a, b);    /*calling function */
}
void sum (int x, int y)
{
  int z;
  z=x+y;
  printf (“\n The Sum =%d”, z);
}
```

FUNCTIONS WITH ARGUMENTS AND RETURN VALUES

In this category there is data transfer between the calling function and called function.



```
// Function Declaration
int sqr (int x);
int main (void)
{
// Local Declarations
    int a;
    int b;
// Statements
    scanf("%d", &a);
    b  =  sqr (a);
    printf("%d squared: %d\n", a,  b);
    return 0;
}  // main
```

```
int sqr (int x)
 {
// Statements
    return (x * x);
}  // sqr
```

Figure: Functions with arguments and return values

Observe from the above figure, the function **sqrt** receive one value from the function **main**(), finds the square of the number and sends the result back to the calling function.

//C program to find sum of two numbers using functions with arguments and return values

```
#include<stdio.h>
int sum (int ,int );
void main ()
{
  int a, b;
   printf (“\n Enter the values of a and b:  “);
  scanf (“%d%d”, &a, &b);
  c=sum (a, b);    /*calling function */
  printf (“\n The Sum =%d”, c);
 }
int sum (int x, int y)
{
  int z;
  return x+y;
 }
```

Note: generally we are using functions with arguments and return values (category 4) in our applications. Why because the job of a function is to perform a well-defined task, it carries inputs and sends result after executed. In real world the programming teams codes the modules (functions) according to the input (arguments) given by the team coordinators.

Understanding the Scope of a Function

The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Here in this section we will discuss one specific scope: the one defined by function.

Each function is a discrete block of code. Thus, a function defines a block scope. This means that a function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use **goto** to jump into the middle of another function.) The code that constitutes the body of a function is hidden from the rest of the program, and unless it uses global variables, it can neither affect nor be affected by other parts of the program. Stated another way, 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.

Parameter passing Methods/Techniques (or) Communication among Functions (or) Function Arguments

If a function is to accept arguments, it must declare the parameters that will receive the values of the arguments. The function arguments or parameters occur only after the function name. There are two ways of passing parameters to the functions.

1. Call by value
2. Call by reference

Call by value

When a function is called with actual parameters, the values of actual parameters are copied into the formal parameters. If the values of the formal parameters changes in the function, the values of the actual parameters are not changed. This way of passing parameters is called call by value (pass by value).

In the below example, the values of the arguments to **swap() 10** and **20** are copied into the parameters **x** and **y**. Note that the values of **x** and **y** are swaped in the function. But, the values of actual parameters remain same before swap and after swap.

```
// C program illustrates call by value
#include<stdio.h>
void swap (int , int );
void main ()
{
  int a, b;
  printf (“\n Enter the values of a and b:  “);
  scanf (“%d%d”, &a, &b);
  swap (a, b);   /*calling function */
  printf (“\nFrom main The Values of a and b a=%d, b=%d “, a,
b);
}
void swap (int x, int y)
{
  int temp;
  temp=x;
  x=y;
  printf (“\n The Values of a and b af   y=temp;
ter swapping a=%d, b =%d”, x, y);
}
```

OUTPUT

```
Enter the values of a and b:  10 20
The Values of a and b after swapping a=20, b=10
```

Figure 3.5 Example Call by Value

Note: In call by value any changes done on the formal parameter will not affect the actual parameters.

Call by Reference

- ❖ Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- ❖ We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap. This is known as **"call by value".**
- ❖ Here we are going to discuss how to pass the address.

Call by Reference

Instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference", since we are referencing the variables.

Here the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Here The formal parameters should be declared as pointer variables to store the address.

The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now swapped when the control is returned to main function.

```
#include <stdio.h>
void swap ( int *pa, int *pb ) ;
int main ( )
{
  int a = 5, b = 6;
  printf("a=%d b=%d\n",a,b) ;
  swap (&a, &b) ;
  printf("a=%d b=%d\n",a,b) ;
  return 0 ;
}
```

```
void swap( int *pa, int *pb )
{
  int temp;
  temp= *pa; *pa= *pb;  *pb = temp ;
  printf ("a=%d  b=%d\n", *pa, *pb);
}
```

Results:

```
   a=5  b=6
   a=6  b=5
   a=6  b=5
```

Observe the following points when the program is executed,

- ❖ The address of actual parameters **a** and **b** are copied into formal parameters **pa** and **pb**.

❖ In the function header of **swap()**, the variables **a** and **b** are declared as pointer variables.

❖ The values of **a** and **b** accessed and changed using pointer variables **pa** and **pb**.

Call by Value	***Call by Reference***
When Function is called the values of variables are passed.	**When a function is called address of variables is passed.**
Formal parameters contain the value of actual parameters.	**Formal parameters contain the address of actual parameters.**
Change of formal parameters in the function will not affect the actual parameters in the calling function	**The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters**
Execution is slower since all the values have to be copied into formal parameters.	**Execution is faster since only addresses are copied.**

Table: Difference between Call by Value and Call by Reference

The Four C Scopes

In the preceding discussion (and throughout the remainder of this book) the terms *local* and *global* are used to describe in a general way the difference between identifiers that are declared within a block and those declared outside all blocks. However, these two broad categories are more finely subdivided by C. Standard C defines four scopes that determine the visibility of an identifier. They are summarized here:

SCOPE	**MEANING**
File scope	Starts at the beginning of the file and ends with the end of the file. It refers only to those identifiers that are declared outside of all functions. File scope identifiers are visible throughout the entire file. Variables that have file scope are global.
Block scope	Begins with the opening { of a block and ends with its Associated closing }. However, block scope also extends to function parameters in a function definition. That is, function parameters are included in a function's block scope. Variables with block scope are local to their block.
Function prototype scope	Identifiers declared in a function prototype; visible within the prototype.
Function scope	Begins with the opening { of a function and ends with its

	closing }. Function scope applies only to labels. A label is used as the target of a **goto** statement, and that label must be within the same function as the **goto**.

C – Type Qualifiers

C – Type qualifiers: The keywords which are used to modify the properties of a variable are called type qualifiers. C provides three type qualifiers **const, volatile, restrict.**

Const and volatile qualifiers can be applied to variables and pointers, but restrict qualifiers may only be applied to pointers.

TYPES OF C TYPE QUALIFIERS:

There are three types of qualifiers available in C language. They are,

1. const
2. volatile
3. restrict

1. CONST KEYWORD:

- A variable can be made unchanged during program execution by declaring the variable as a constant. The keyword **const** is placed just before the variable declaration to become as constant variable.

  Syntax:

  const data_type variable_name; (or) const data_type *variable_name;

  For example:

  const int a;

  Here **a** is a constant and its value cannot be changed.

If we declare like

int *const x;

In the above example, the pointer to x is a constant. The value that **x** points Can be changed, but the value of **x** cannot be changed.

- Constants are also like normal variables. But, only difference is, their values can't be modified by the program once they are defined.
- They refer to fixed values. They are also called as literals.
- They may be belonging to any of the data type.

2. VOLATILE KEYWORD:

- Variables that can be changed at any time by external programs or the same program are as volatile variables. The keyword volatile is placed before declaration. To make a variable value changeable by the current program and unchangeable by other programs, declare the variable as volatile and constant.
- **Syntax:**

  volatile data_type variable_name;

  - **(or)**

  Volatile data_type *variable_name;

Examples:

```
volatile int x;
volatile const int y;
int * volatile z; (or) volatile int * z;
```

- The variable **x** value can be changed by any program at any time and the variable **y** can be changed in the current program but not by the external programs. The variable **z** is a pointer to a volatile int.

3. RESTRICT KEYWORD:

- The restrict type qualifier may only be applied to a pointer. A pointer declaration that uses this type qualifier establishes a special association between the pointer and the object it accesses, making that pointer and expressions based on that pointer, the only ways to directly or indirectly access the value of that object,
- A pointer is the address of a location in memory. More than one pointer can access the same chunk of memory and modify it during the course of a program. The restrict type qualifier is an indication to the compiler that, if the memory addressed by the restrict qualified pointer is modified, no other pointer will access that same memory. The compiler may choose to optimize code involving restrict-qualified pointers in a way that might otherwise result in incorrect behavior. It is the responsibility of the programmer to ensure that restrict-qualified pointers are used as they were intended to be used. Otherwise, undefined behavior may result.

STORAGE CLASS SPECIFIERS (or) STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. In addition to this, the storage class gives the following information about the variable or the function.

- The storage class of a function or a variable determines the part of memory where storage space will be allocated for that variable or function (whether the variable/function will be stored in a register or in RAM)
- It specifies how long the storage allocation will continue to exist for that function or variable.
- It specifies the scope of the variable or function, i.e., the storage class indicates the part of the C program in which the variable name is visible or the part in which it is accessible. In other words, whether the variable/function can be referenced throughout the program or only within the function, block or source file where it has been defined.
- It specifies whether the variable or function has internal, external, or no linkage.
- It specifies whether the variable will be automatically initialized to zero or to any indeterminate value.

C supports four storage classes: ***automatic, register, external*** and ***static***. The general syntax for specifying the storage class of a variable can be given as

Syntax:

storage_class_specifier data_type variable_Name;

(or)

storage_class_specifier data_type variable _Name = value;

The Automatic Storage Class:

1. The auto storage class specifier is used to explicitly declare a variable with ***automatic storage***. It is the default storage class for variables declared inside a block. For example, if we write
   auto int x;
   then **x** is an integer that has automatic storage. It is deleted when the block in which **x** was declared exits. The keyword used to define automatic storage class is **auto**
2. The default value of automatic storage class variable is ***garbage value***. i.e., if auto variables are not initialized at the time of declaration, then they contain some garbage value.
3. The scope of automatic variable is local to the block in which it is declared. These variables are stored in the primary memory of the computer.
4. The following code uses an auto integer that is local to the function in which it is defined

```
void main()
{
 auto mum = 20 ;
 {
      auto num = 60 ;
              printf("nNum : %d",num);
 }
 printf("nNum : %d",num);
}
```

Output :

```
Num : 60
Num : 20
```

Note : Two variables are declared in different blocks , so they are treated as different variables

The Register Storage Class:

1. The register storage class specifier is used to explicitly declare a variable with ***register storage***. The keyword used to define register storage variable is ***register.***
   A register variable is declared in the following manner
   register int x;
2. The default value of register storage class variable is ***garbage value***. i.e., if register variables are not initialized at the time of declaration, then they contain some garbage value.
3. When a variable is declared using **register** as its storage class, it is stored in a CPU register instead or RAM. Like auto variables, register variables also have block scope i.e., each time a block is entered, the storage for register variable defined in that block are accessible and the moment that block is exited, the variables becomes no longer accessible for use.
4. The following code uses an register variable that is local to the function in which it is defined

```
#include<stdio.h>
main()
{
int num1,num2;
register int sum;
printf("\nEnter the Number 1 : ");
scanf("%d",&num1);
printf("\nEnter the Number 2 : ");
scanf("%d",&num2);
sum = num1 + num2;
printf("\nSum of Numbers : %d",sum);
}
```

Explanation of the above program:

1. In the above program we have declared two variables num1,num2. These two variables are stored in RAM.
2. Another variable is declared which is stored in register variable. Register variables are stored in the register of the microprocessor. Thus memory access will be faster than other variables.
3. If we try to declare more register variables then it can treat variables as Auto storage variables as memory of microprocessor is fixed and limited.

Note: One drawback of using a register variable is that they cannot be operated using the unary '&' operator because it does not have a memory location associated with it.

The Static Storage Class:

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);
 static int count = 5; /* global variable */
 main() {

   while(count--) {
      func();
   }
   return 0;
}

/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();


main()
 {
   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("count is %d\n", count);
}
```

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

```
$gcc main.c support.c
```

It will produce the executable program **a.out**. When this program is executed, it produces the following result −

```
count is 5
```

Storage class	**Location of storage**	**Default value**	**scope**	**lifetime**
Automatic	Memory	Garbage value	local	within the function in which it has been declared.
Register	CPU registers	Garbage value	Local	within the function in which it has been declared

Static	Memory	zero	Local	the value persist till the end of the program.
external	Memory	Garbage value	Global	Entire Program.

Table: Storage Class Specifiers

THE RETURN STATEMENT

The return statement is used to terminate the execution of a function and returns control to the calling function. When the *return* statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.

- ❖ The return statement is the mechanism for returning a value from the called function to its caller.
- ❖ The return statement may or may not return a value to the calling function.
- ❖ The general form of the return statement is

return expression;

- ❖ The calling function is free to ignore the returned value. Furthermore, there need not be expression after the return.

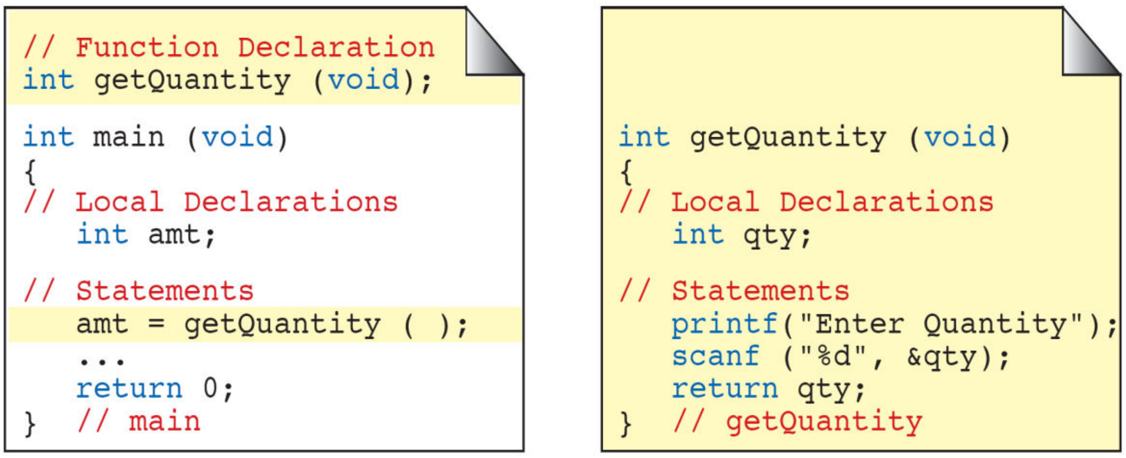
- ❖ In any case if a function fails to return a value, its value is certain to be garbage.

- ❖ The return statement has two important uses

1. It causes an immediate **exit of the control** from the function. That is, it causes program execution to return to the calling function.

2. It returns the value present in the expression.

```
example:  return(x+y);
          return (6*8);
          return (3);
          return;
```



```
// Function Declaration
int getQuantity (void);

int main (void)
{
// Local Declarations
   int amt;

// Statements
   amt = getQuantity ( );
   ...
   return 0;
}  // main
```

```
int getQuantity (void)
{
// Local Declarations
   int qty;

// Statements
   printf("Enter Quantity");
   scanf ("%d", &qty);
   return qty;
}  // getQuantity
```

Figure: Functions with no arguments and return values

In this category, there is no data transfer from the calling function to the called function. But, there is data transfer from called function to the calling function.

In the above example, observe from the above figure that the function **getQuantity()** do not receive any value from the function **main()**. But, it accepts data from the keyboard, and returns the value to the calling function.