PROGRAMMING IN C

UNIT III FUNCTIONS AND POINTERS

Introduction to functions: Function prototype, function definition, function call, Built-in functions (string functions, math functions) – Recursion – Example Program: Computation of Sine series, Scientific calculator using built-in functions, Binary Search using recursive functions – Pointers – Pointer operators – Pointer arithmetic – Arrays and pointers – Array of pointers – Example Program: Sorting of names – Parameter passing: Pass by value, Pass by reference – Example Program: Swapping of two numbers and changing the value of a variable using pass by reference

APEC

Introduction:

Structured programming is a programming technique in which a larger program is divided into smaller subprograms. These subprograms are easy to understand, easy to implement. The structured programming enables code reusability. Code reusability is a method of writing code once and using it many times. In C, the structured programming can be designed using functions concept. Using functions concept, we can divide larger program into smaller subprograms and these subprograms are implemented individually.

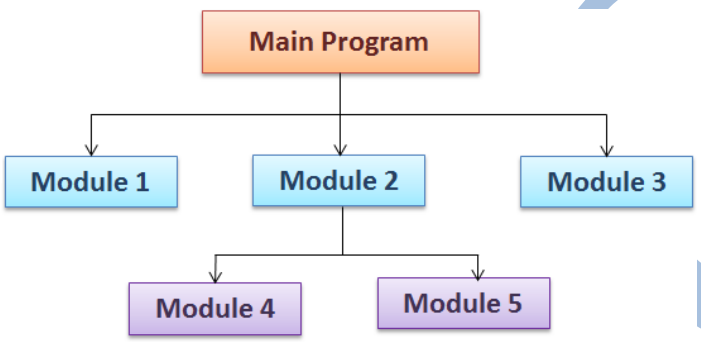


Figure 3.1 Program is divided into multiple functions.

In the previous chapters, we have discussed the three functions, namely, **main()**, **printf()** and **scanf()**. Every C program starts with the user-defined function main(). The C language supports two types of functions.

- ***Library functions***
    - Library functions are pre-defined set of functions
    - These are called as system defined functions or Pre-Defined Functions.
    - printf(), scanf(), sqrt() are examples of library functions
- ***User-defined functions***
    - User-defined functions have to be developed by the user at the time of writing the program.
    - main() is an example of user-defined functions.

Define Function

- A function is a block of statements that together perform a specific task.
- Every C program has at least one function, which is main().
- A function can also be referred as a method or a sub-routine or a procedure, etc.

Uses of C Functions

- Using functions, larger program can be divided into smaller modules.
- Modular programming is implemented using functions
- Main advantages of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability
- It is easy to track when the program is divided into functions.
- Once a function is created it can be used many times (code re-usability).

Syntax for Function:

Every function in C has the following.

- Function Declaration (Function Prototype)
- Function Definition
- Function Call

Function Declaration

- The function declaration informs the compiler about function name, datatype of the return value and number of arguments.
- The function declaration is also called as function prototype.
- The function declaration is performed before main function or inside main function or inside any other function.

Syntax for function declaration:

```
returnTypefunctionName(parameters);
```

- returnType specifies the datatype of the value which is sent as a return value from the function definiton.
- functionName is a user defined name used to identify the function uniquely.
- parametersList is the data values that are sent to the function definition.

Example:

```
//function name = display, receives a character as argument and returns nothing
void display(char);

//function name = sum, receives two integers as argument and returns an integer
intsum(int,int);
```

APEC

Function Definition

- The function definition provides the actual code of that function. The function definition is also known as body of the function
- The actual instructions of a function are written inside the braces "{ }".
- The function definition is performed before main function or after main function.

Syntax for Function definition

```
returnTypefunctionName(parameters)
{
//Function body
}
```

- Function body is the collection of statements.

Example:

```
int sum (int x, int y)
{
// Function body has 2 statements
int s = x+y;
return s;
}
```

Function Call

- Function call statement calls the function by matching its name and arguments. A function call can be made by using function name and providing the required parameters.

Syntax for Function call

```
functionName(parameters);
or
variable=functionName(parameters);
```

Example:

```
display(a);
or
 s = sum(x,y);
```

Working of C - function

AEC

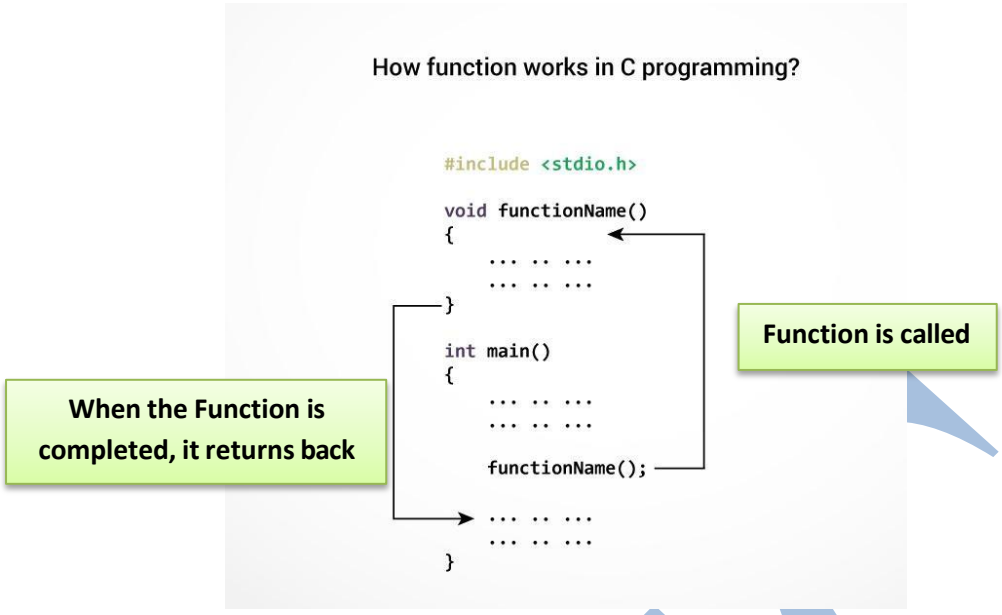


Figure 3.2 Working of a C - function

```
/* Simple C program using Functions*/
void message();              //Function prototype
void main()
{
     message();              //Function call
   }

void message()               // Function Definition
{
printf("Hello World");
 }
```

Execution:

Output:

```
Hello World
```

```
/* C program to explain Multiple Functions in a program*/
void biology();              //Function prototypes
void maths();
void commerce();
```

APEC

```
void main()
{
biology();            //Function call 1
maths();  //Function call 2
     commerce();      //Function call 3
   }

void biology()              // Function Definition 1
{
printf("I am Biology Student\n");
 }

void maths()              // Function Definition 2
{
printf("I am maths Student\n");
 }

void commerce()             // Function Definition 2
{
printf("I am commerce Student\n");
 }
```

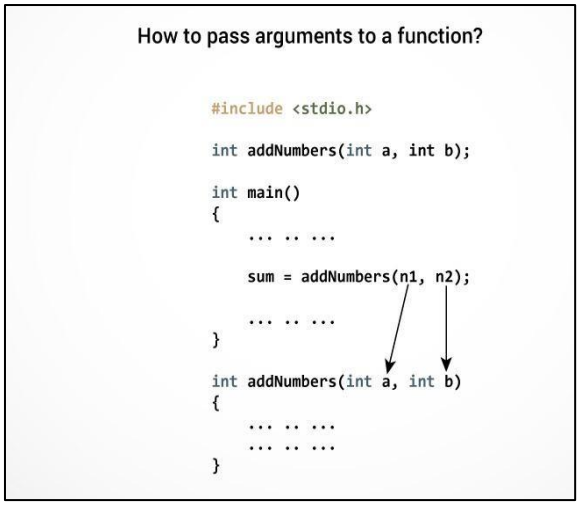
Execution:

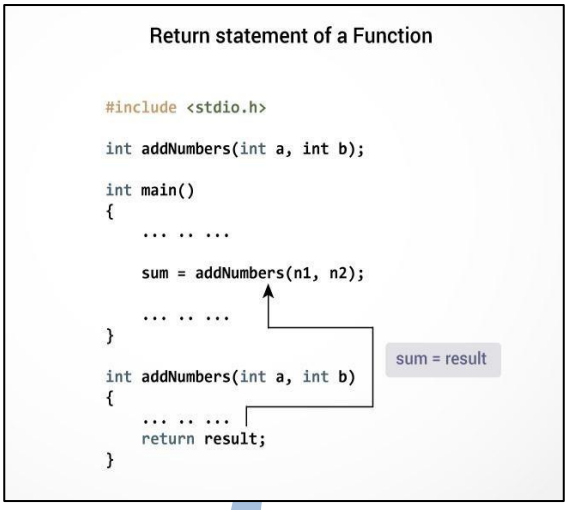
Output:

```
I am Biology Student
I am maths Student
I am commerce Student
```

In the concept of functions, the function call is known as "Calling Function" and the function definition is known as "Called Function". When we make a function call, the execution control jumps from calling function to called function. After executing the called function, the execution control comes back to calling function from called function. The data values transferred from calling function to called function are called as Parameters and the data value transferred from called function to calling function is called Return value.

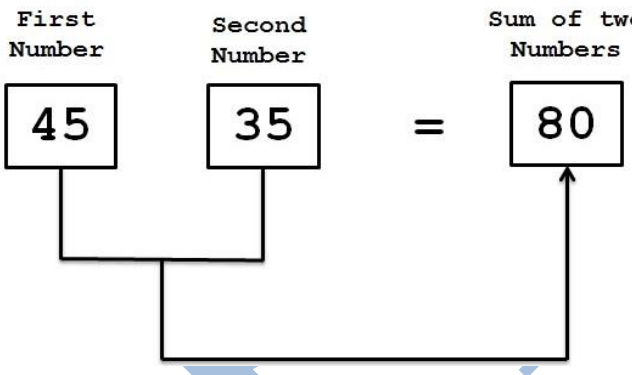
APEC





/*C Program to add integers and display sum in main() function */

Example:




```
#include <stdio.h>
int addNumbers(int a, int b);//function declaration
void main()
{
  intsum;
  sum = addNumbers(10,20);          //function call
  printf("sum of two numbers = %d",sum);
}
intaddNumbers(inta,int b) //function definition
{
  /* Start of function definition. */
   int add;
   add=a+b;
   return add;////return statement of function
/* End of function definition. */
```

```
}
```

Execution:

Output:

```
sum of two numbers  = 30
```

FUNCTION PROTOTYPES:

Based on the data flow between the calling function and called function, the functions are classified as follows...

- Functions with no arguments and no return value.
- Functions with no arguments and with return value
- Functions with arguments and no return value.
- Functions with arguments and with return value

Functions with no arguments and no return value.

- In this type of functions there is no data transfer between calling function and called function.
- The execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function.

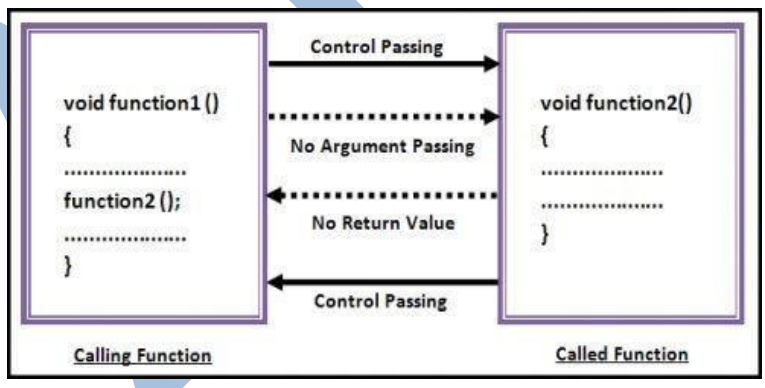


Figure 3.4No arguments and no return values

```
/* C program to explain functions with no arguments and no return
value */
#include <stdio.h>
void main()
```

```
{
   void addition() ; // function declaration

   addition() ; // function call

}
void addition()  // function definition
{
  int num1, num2 ;
  printf("Enter any two integer numbers : ") ;
  scanf("%d%d", &num1, &num2);
  printf("Sum = %d", num1+num2 ) ;
}
```

Execution:

Input:

```
Enter any two integer numbers : 20 30
```

Output:

```
Sum = 50
```

Functions with no arguments and with return value

- In this type of functions there is no data transfer from calling function to called function (parameters) but there is data transfer from called function to calling function (return value).
- The execution control jumps from calling function to called function and executes called function, and finally comes back to the calling function along with a return value.

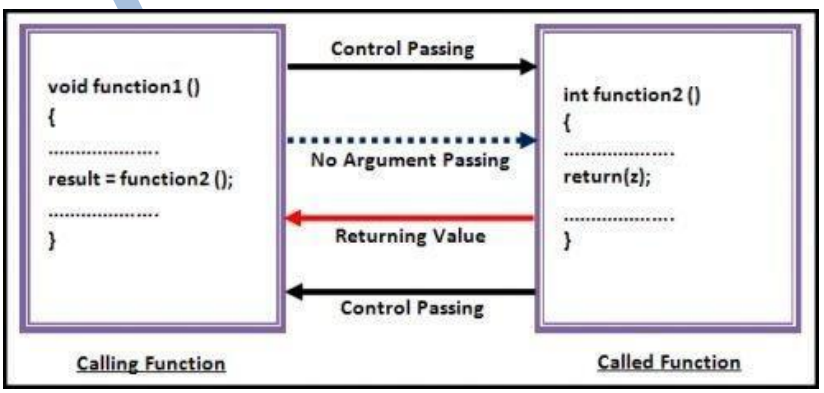


Figure 3.5No arguments and with return values

APEC

```c
/* C program to explain functions with no arguments and with
return value */

#include <stdio.h>
void main()
{
   int result ;
   int addition() ; // function declaration

   result = addition() ; // function call
   printf("Sum = %d", result) ;
}
int addition()  // function definition
{
   int num1, num2 ;
   printf("Enter any two integer numbers : ") ;
   scanf("%d%d", &num1, &num2);
   return (num1+num2) ;
}
```

Execution:

Input:

```
Enter any two integer numbers : 20 30
```

Output:

```
Sum = 50
```

Functions with arguments and no return value.

- In this type of functions there is data transfer from calling function to called function (parameters) but there is no data transfer from called function to calling function (return value).
- The execution control jumps from calling function to called function along with the parameters and executes called function, and finally comes back to the calling function.

APEC

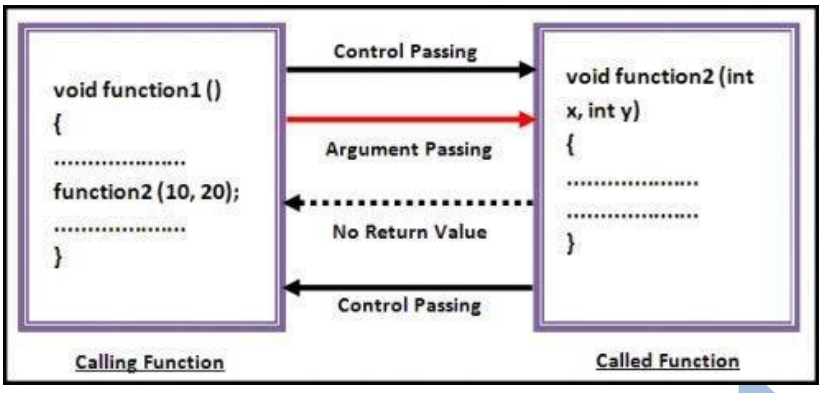


Figure 3.6With arguments and no return values

```
/* C program to explain functions with arguments and no return
value */
#include <stdio.h>
void main()
{
   int num1, num2 ;
   void addition(int, int) ; // function declaration
   printf("Enter any two integer numbers : ") ;
   scanf("%d%d", &num1, &num2);

  addition(num1, num2) ; // function call
}

void addition(int a, int b)  // function definition
{
  printf("Sum = %d", a+b ) ;
}
```

Execution:

Input:

```
Enter any two integer numbers : 20 30
```

Output:

```
Sum = 50
```

Functions with arguments and return value.

APEC

- In this type of functions there is data transfer from calling function to called function (parameters) and also from called function to calling function (return value).
- The execution control jumps from calling function to called function along with parameters and executes called function, and finally comes back to the calling function along with a return value.

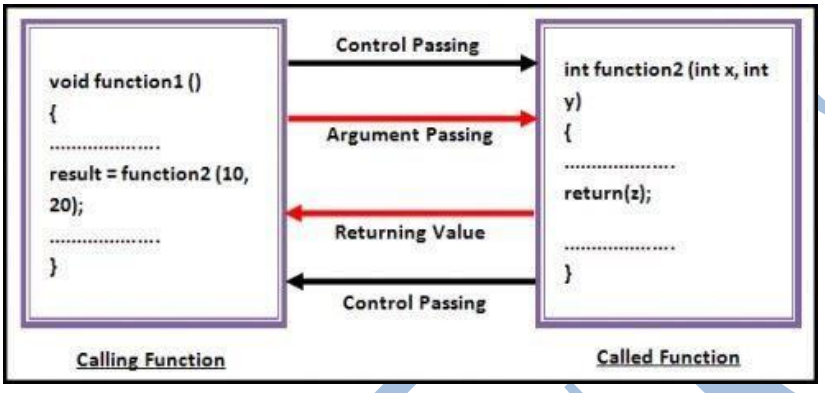


Figure 3.7With arguments and with return values

```
/* C program to explain functions with arguments and with return
value */
#include <stdio.h>
void main()
{
   int num1, num2, result ;
   int addition(int, int) ; // function declaration
   printf("Enter any two integer numbers : ") ;
   scanf("%d%d", &num1, &num2);

   result = addition(num1, num2) ; // function call
   printf("Sum = %d", result) ;
}

int addition(int a, int b)  // function definition
{
   return (a+b) ;
}
```

Execution:

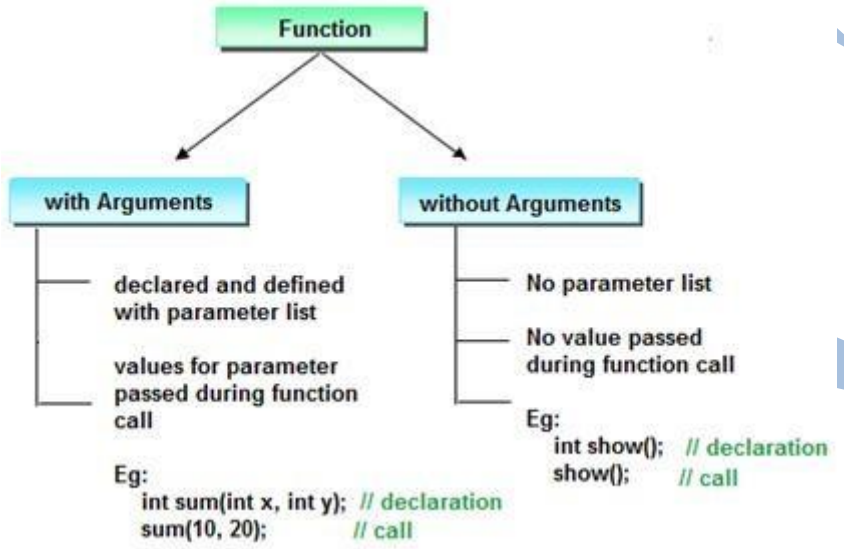
Input:

```
Enter any two integer numbers : 20 30
```

Output:

```
Sum = 50
```

Functions with arguments Vs Functions without arguments.



Parameter Passing in C

When the execution control is transferred from calling function to called function it may carry one or more number of data values. These data values are called as parameters.

- ***Parameters*** are the data values that are passed from calling function to called function.

In C, there are two types of parameters and they are as follows:

- Actual Parameters
  - The actual parameters are the parameters that are specified in calling function.
- Formal Parameters
  - The formal parameters are the parameters that are declared at called function.

When a function gets executed, the copy of actual parameters is copied to the formal parameters.

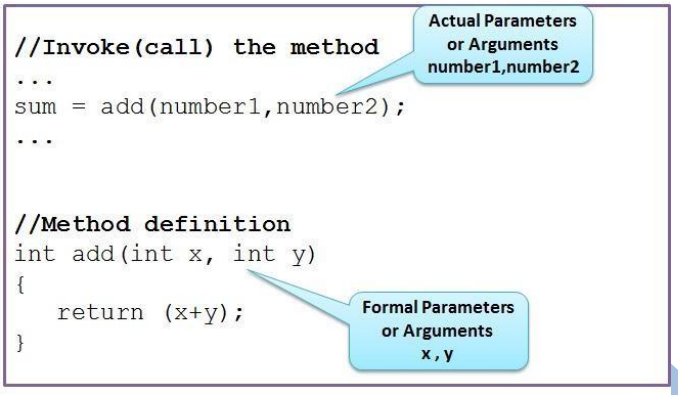


Figure 3.8Formal and Actual Parameters

There are two methods to pass parameters from calling function to called function and they are as follows.

- Call by Value (or) Pass by value
- Call by Reference (or) Pass by Reference

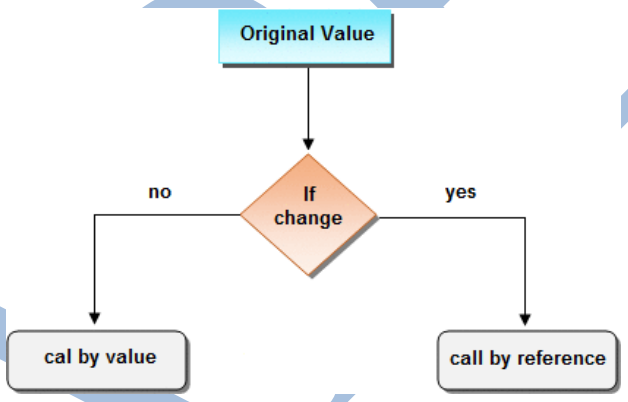


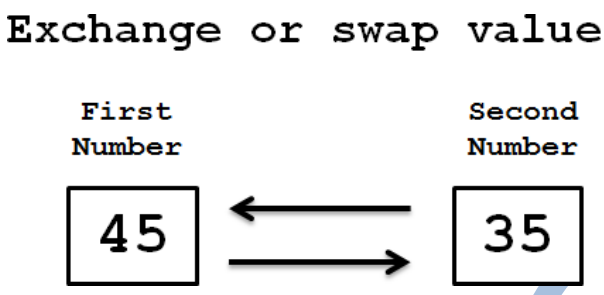
Figure 3.10Call by ValueVsCall by Reference

Call by Value

- In call by value parameter passing method, the copy of actual parameter values are copied to formal parameters.
- The changes made on the formal parameters have no effect on the actual parameters.
    - That means, after the execution control comes back to the calling function, the actual parameter values remains same.

/* C program to swap two numbers using call by value */

Example:




```
#include <stdio.h>
void main()
{
  int num1, num2 ;
  void swap(int,int) ; // function declaration
   num1 = 10 ;
   num2 = 20 ;

   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
   swap(num1, num2) ; // calling function
   printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
}

void swap(int a, int b)  // called function
{
   int temp ;
   temp = a ;
   a = b ;
   b = temp ;
}
```

Execution:

Output:

```
Before swap: num1 = 10, num2 = 20
After swap: num1 = 10, num2 = 20
```

Here, num1 and num2 are called actual parameters,a andb are called formal parameters. num1 is copied to a and num2 is copied to b. Changes made on a and b

APEC

have no effect on num1 and num2.

Call by Reference

- In Call by Reference parameter passing method, the address of the memory location address is copied to formal parameters. The address of the actual parameters is passed to the called function and is received by the formal parameters (pointers).
    - Hence changes made on the formal parameters have effect on the actual parameters also.

```
/* C program to swap two numbers using call by reference */

#include <stdio.h>
void main()
{
   int num1, num2 ;
   void swap(int*,int*) ; // function declaration
   num1 = 10 ;
   num2 = 20 ;

   printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
   swap(&num1, &num2) ; // calling function
   printf("\nAfter swap: num1 = %d\nnum2 = %d", num1, num2);
}

void swap(int*a, int*b)  // called function
{
   int temp ;
   temp = *a ;
   *a = *b ;
   *b = temp ;
}
```

Execution:

Output:

```
Before swap: num1 = 10, num2 = 20
After swap: num1 = 20, num2 = 10
```

APEC

Here, the addresses num1 and num2 are copied to a and b. Hence, changes made on the pointer variables a and b have effect on the actual parameters too.

Nesting of Functions:

- A function calling another function is known as a nested function.

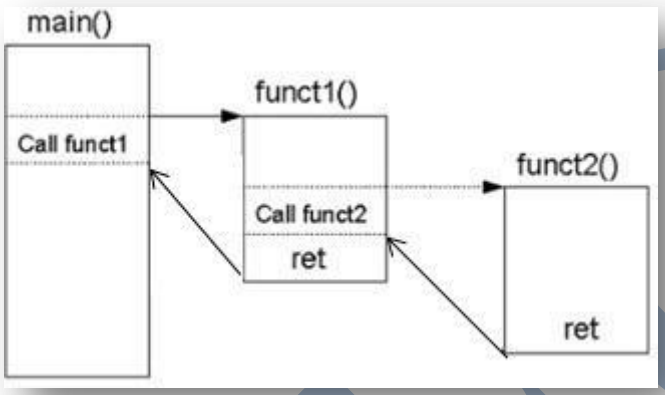


Figure 3.2 Nested Functions

```
/* one Function calling another Function in a C program*/
void message();            //Function prototype
void message();            //Function prototype
void main()
{
      message();           //Function call
   }

void message()             // Function Definition
{
    say();
    printf("Hello World");
 }

void say()
{
   printf("This will be executed first\n");
  }
```

APEC

Execution:

Output:

```
This will be executed first
Hello World
```

APEC

Built-In Functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.The various built-in functions along with the header files are displayed in the following table.

Table 3.1 Header files and standard functions		
Header File	**Purpose**	**Example**
stdio.h	Provides functions to perform standard I/O operations	printf(), scanf()
conio.h	Provides functions to perform console I/O operations	clrscr(), getch()
string.h	Provides functions to handle string data values	strlen(), strcpy()
stdlib.h	Provides functions to perform general functions	calloc(), malloc()
math.h	Provides functions to perform mathematical operations	sqrt(), pow()
time.h	Provides functions to perform operations on time and date	time(), localtime()
graphics.h	Provides functions to draw graphics.	circle(), rectangle()

- "math.h" header file supports all the mathematical related functions in C language. All the arithmetic functions used in C language are given below.

Table 3.2 Math Functions			
Function	**Description**	**Example**	**Output**
Include the Header file: **#include <math.h>**			
abs ( )	Returns the positive absolute value of an integer.	abs(200) abs(-400)	200 400
floor ( )	Returns the nearest integer.	floor(5.1) floor(5.9)	5 5

		floor(-5.4)	-6
round ( )	Returns the nearest integer value	round(5.4) round(5.9) round(-5.4)	5 6 -5
ceil ( )	Returns nearest integer value which is greater than or equal to the value passed	ceil(5.4) ceil(5.9)	6 6
sin ( )	Returns sine value.	sin(0.314)	0.3088
cos ( )	Returns cosine value.	cos(0.314)	0.9511
tan ( )	Returns tangent value.	tan(0.314)	0.3247
sinh ( )	Returnshyperbolic sine value.	sinh(0.25)	0.2526
cosh ( )	Returns hyperbolic cosine value.	cosh(0.25)	1.0314
tanh ( )	Returns hyperbolic tangent value.	tanh(0.25)	0.2449
exp ( )	Returns the exponential "e" value	exp(6.25)	518.01
log ( )	Returns natural logarithm value.	log(6.25)	1.8326
log10 ( )	Returns base 10 logarithm values.	log10(6.25)	0.7959
sqrt ( )	Returns the square root of a number.	sqrt(16) sqrt(2)	4 1.141
pow ( )	Returns the power of a number.	pow(2,4) pow(5,3)	16 125
trunc( )	Truncates the decimal value from floating point value and returns integer value	trunc(16.99) trunc(20.1)	16 20

- "string.h" header file supports all the string functions in C language. All the string functions are given below.

Table 3.3 String Functions

Function	Description	Example	Output
Include the Header file: **#include <string.h>**			
strcat ( )	Concatenates s2 at the end of s1	s1 = "World " s2 = "of C"	World of C

Table 3.3 String Functions			
Function	**Description**	**Example**	**Output**
		strcat (s1,s2) printf(“%s”,s1)	
strncat ( )	Appends a portion of string to another	strncat (s1,s2,2) printf(“%s”,s1)	World of
strcpy ( )	Copies s2 into s1	strcpy (s1,s2) printf(“%s”,s1)	of C
strncpy ( )	Copies given number of characters of one string to another	strncpy (s1,s2,2) printf(“%s”,s1)	of
strlen ( )	Gives the length of s1	c = strlen(s1) printf(“%d”,c)	5
strcmp ( )	Returns 0 if s1 is same as s2. Returns <0 if s1<s2. Returns >0 if s1>s2	i = strcmp(s1,”World”) printf(“%d”,i) i = strcmp(s1,s2) printf(“%d”,i) i = strcmp(s2,s1) printf(“%d”,i)	0 1 -1
strcmpi ( )	Same as strcmp() function. But, this function negotiates case.	i = strcmp(s1,”WORLD”) printf(“%d”,i)	0
strchr ( )	Returns pointer to first occurrence of char in str1	i = strchr(s1,’o’) printf(“%d”,i)	1
strrchr ( )	Returns the last occurrence of given character in a string	s1 = “World of C” i = strchr(s1,’o’) printf(“%d”,i)	6
strstr ( )	Returns pointer to first occurrence of s2 in s1	s1 = “World of C” i = strstr(s1,”of”) printf(“%d”,i)	6
strrstr ( )	Returns pointer to last occurrence of s2 in s1	s1 = “Test to Test you ” i = strstr(s1,”Test”) printf(“%d”,i)	8
strdup ( )	Duplicates the string	s1 = “Hello”	Hello

APEC

Table 3.3 String Functions			
Function	**Description**	**Example**	**Output**
		s2 = strdup(s1) printf("%s",s2)	
strlwr ( )	Converts string to lowercase	s1 = "Hello" printf("%s",strlwr(s1))	hello
strupr ( )	Converts string to uppercase	s1 = "Hello" printf("%s",strupr(s1))	HELLO
strrev ( )	Reverses the given string	s1 = "Hello" printf("%s",strrev(s1))	olleH
strset ( )	Sets all character in a string to given character	s1 = "Hello" printf("%s",strrev(s1,'#'))	#####
strnset ( )	It sets the portion of characters in a string to given character	s1 = "Hello" printf("%s",strrev(s1,'#',2))	##llo
strtok ( )	Tokenizing given string using delimiter	s1 = "Hello to all" printf("%s",strtok(s1,":"))	Hello:to:all

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, a function calling itself is called recursive function.

```
void recursion()
{
   recursion(); /* function calls itself */
}

void main()
{
  recursion();
 }
```

The recursive functions should be used very carefully because; when a function called by itself it enters into infinite loop. Define the condition to exit from the function call so that the recursive function gets terminated.Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

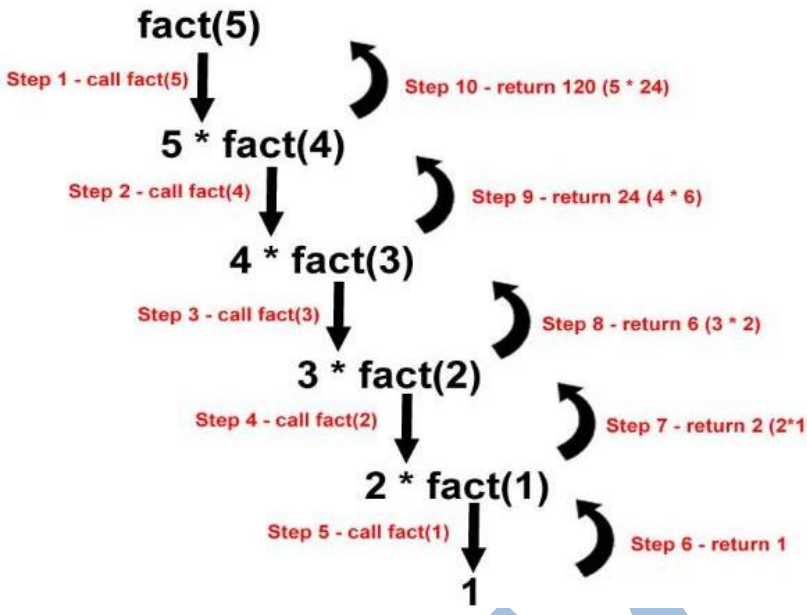


Figure 3.8Factorial using Recursion

```
/* C Program to find Factorial of a given number using Recursion */

#include <stdio.h>
int fact(int i)
{
   if(i <= 1)
   {
      return 1;
   }
   return i * fact(i - 1);
}
void  main()
{
  int i = 5;
  printf("Factorial = %d", fact(i));
}
```

Execution:

Output:

```
Factorial = 120
```

APEC

/* C Program to generate Fibonacci Series using Recursion */

Example:

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	13+21=34
1+2=3	21+34=55
2+3=5	34+55=89
3+5=8	55+89=144
5+8=13	89+144=233
8+13=21	144+233=377

```
#include <stdio.h>
int fibonacci(int i)
{
   if(i == 0)
      return 0;

   if(i == 1)
      return 1;
   return fibonacci(i-1) + fibonacci(i-2);
}

void  main()
{
   int i;
   for (i = 0; i < 10; i++)
      printf("%d\t", fibonacci(i));
  }
```

Execution:

Output:

```
0     1     1     2     3     5     8     13    21    34
```

Merits of using Recursion

- To write efficient programs using a minimum amount of code.
- Nominal chance of syntax and logical errors.

APEC

- Increases the program efficiency.

Demerits of using Recursion

- Can cause infinite loops and other unexpected results if not written properly
- Recursion takes a lot of stack space, usually not considerable when the program size is small.
- Recursion uses more processor time.

Key Points to remember while writing Functions

- Every C program has a function called main()
- C Functions can be invoked from anywhere within a C program.
- C functions can be called with or without arguments/parameters. These arguments inputs to the functions.
- C functions may or may not return values to calling functions. These values output of the functions.
- When a function completes its task, program control is returned to the function from where it is called.
- Before calling and defining a function, declare function prototype to inform the compiler about the function name, function parameters and return value type.
- When return data type of a function is “void”, then, it won’t return any values
- When return data type of a function is other than void such as “int, float, double”, it returns value to the calling function.

APEC

Example programs

Program 1: Computation of Sine series using Built-in functions

```
/* C Program to Compute Sine series using Built-in functions */
```

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
void main()
{
   int n, x1;
   float acc, term, den, x, sinx=0, sinval;
   clrscr();
   printf("Enter the value of x (in degrees)\n");
   scanf("%f",&x);
   x1 = x;
   /* Converting degrees to radians*/
   x = x*(3.142/180.0);
   sinval = sin(x);
   printf("Enter the accuary for the result\n");
   scanf("%f", &acc);
   term = x;
   sinx = term;
   n = 1;
   do
   {
     den = 2*n*(2*n+1);
     term = -term * x * x / den;
     sinx = sinx + term;
     n = n + 1;
  } while(acc<= fabs(sinval - sinx));
  printf("Sum of the sine series = %f\n", sinx);
```

APEC

```
  printf("Using Library function sin(%d) = %f\n", x1,sin(x));
  getch();
 }
```

Execution:

Input:

```
Enter the value of x (in degrees)
45
Enter the accuary for the result
0.00001
```

Output:

```
Sum of the sine series =0.707178
Using Library function sin(45)= 0.707179
```

Program 2: Scientific Calculator using Built-in functions

```
/* C Program Scientific Calculator using Built-in functions */
```




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

void main(void)

{
  int choice, i, a, b;
  float x, y, result;
  clrscr();
```

A.DEC

```
do

{
  printf("\nSelect your operation (0 to exit):\n");
  printf("1. Addition\n2. Subtraction\n3. Multiplication\n4. Division\n");
  printf("5. Square root\n6. X ^ Y\n7. X ^ 2\n8. X ^ 3\n");

  printf("Choice: ");
  scanf("%d", &choice);


  if(choice == 0)
     exit(0);
  switch(choice)
 {
  case 1:
     printf("Enter X: ");
     scanf("%f", &x);
     printf("\nEnter Y: ");
     scanf("%f", &y);
     result = x + y;
     printf("\nResult: %f", result);
     break;

  case 2:
     printf("Enter X: ");
     scanf("%f", &x);
     printf("\nEnter Y: ");
     scanf("%f", &y);
     result = x - y;
     printf("\nResult: %f", result);
     break;

  case 3:
     printf("Enter X: ");
     scanf("%f", &x);
     printf("\nEnter Y: ");
     scanf("%f", &y);
     result = x * y;
     printf("\nResult: %f", result);
     break;
```

APEC

```
 case 4:
    printf("Enter X: ");
    scanf("%f", &x);
    printf("\nEnter Y: ");
    scanf("%f", &y);
    result = x / y;
    printf("\nResult: %f", result);
    break;

 case 5:
    printf("Enter X: ");
    scanf("%f", &x);
    result = sqrt(x);
    printf("\nResult: %f", result);
    break;

case 6:
    printf("Enter X: ");
    scanf("%f", &x);
   printf("\nEnter Y: ");
   scanf("%f", &y);
   result = pow(x, y);
   printf("\nResult: %f", result);
   break;

 case 7:
   printf("Enter X: ");
   scanf("%f", &x);
   result = pow(x, 2);
   printf("\nResult: %f", result);
   break;

 case 8:
   printf("Enter X: ");
   scanf("%f", &x);
   result = pow(x, 3);
   printf("\nResult: %f", result);
   break;

default:
```

APEC

```
     printf("\nInvalid Choice!");
   }
 } while(choice);
 getch();
 }
```

Execution:

Input:

```
Select your operation (0 to exit):
1.Addition
2.Subtraction
3.Multiplication
4.Division
5.Square root
6.X^Y
7.X^2
8.X^3
```

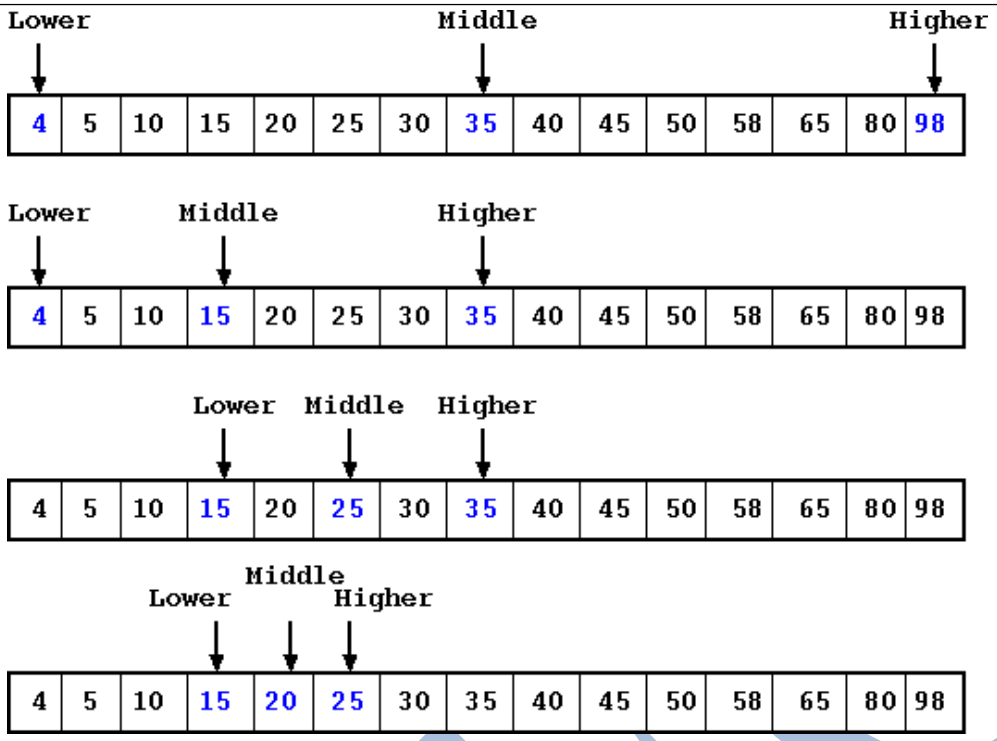
Output:

```
Choice: 1
Enter X:5
Enter Y:4
Result:9
```

Program 3: Binary Search using Recursive functions

```
/* C Program to perform Binary Search using Recursive functions */
```

APEC




```
#include<stdio.h>
void main()
{
  int a[10],i,n,m,c,l,u;

  printf("Enter the size of an array: ");
  scanf("%d",&n);

  printf("Enter the elements of the array: " );
  for(i=0;i<n;i++)
    scanf("%d",&a[i]);

  printf("Enter the number to be search: ");
  scanf("%d",&m);

  l=0,u=n-1;
  c=binary(a,n,m,l,u);

  if(c==0)
     printf("Number is not found.");
  else
     printf("Number is found.");
 }
```

APEC

```
int binary(int a[],intn,intm,intl,int u)
{
  intmid,c=0;
  if(l<=u)
  {
    mid=(l+u)/2;
   if(m==a[mid])
      c=1;
   else if(m<a[mid])
      return binary(a,n,m,l,mid-1);
   else
      return binary(a,n,m,mid+1,u);
     }
  else
   return c;
 }
```

Execution:

Input:

```
Enter the size of an array:
4
Enter the elements of the array:
4 5 6 7
Enter the number to be search:
7
```

Output:

```
Number is found.
```

APEC

Pointers in C

Pointers are the powerful feature of C that differentiates it from other popular programming languages like: Java and Python.In c programming language, we use normal variables to store data values. When we declare a variable, the compiler allocates required memory with specified name. We use a special type of variable called pointer to store the address of another variable with same data type.

- Pointer is a special type of variable used to store the memory location address of another variable.

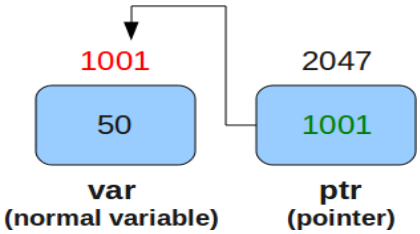


Figure 3.9Pointer variable

When a variable name x is declared as integer type, a memory location is selected with a name x.

$$int\ x;$$

address corresponding to x is shown.

The variable name is the symbolic name for the address in memory. The numeric value of the

	Variable name	Address	Contents
Memory	X	2568	

Figure 3.10Effect of declaring a variable

where, $int\ x\ =\ 32$; is declared, the integer 32 is stored as contents of x.

	Variable name	Address	Contents
Memory	X	2568	32

Figure 3.11A variable name with a value assigned to it.

Accessing the Address of Variables

In c programming language, the reference operator "&" is used to access the address of variable. For example, to access the address of a variable $\&marks$ is used, $p = \&marks$ would assign the address 5000(location of marks) to the pointer variable p. The & operator can be remembered as "$address\ of$" operator.

Declaring Pointers (Creating Pointers)

The pointer variable is declared similar to the creation of normal variable but the name is prefixed with * symbol.

Syntax:

```
datatype *pointerName ;
```

A variable declaration prefixed with * symbol becomes a pointer variable.

Example:

$$int\ *ptr;$$

The variable "ptr" is a pointer variable is of integer datatype, is used to store any integer variable address.

Assigning Address to Pointer

To assign address to a pointer variable use the assignment operator with the following syntax.

Syntax:

```
pointerVariableName = &variableName ;
```

Example:

$$int\ a,*ptr;$$

Here, "a" is a normal integer variable and variable "ptr" is an integer pointer variable

Accessing Variable Value Using Pointer

Pointer variables are used to store the address of other variables. We can use this address to access the value of the variable through its pointer. Use the symbol "*" in front of pointer variable name to access the value of variable to which the pointer is pointing.

Syntax

```
*pointerVariableName
```

APEC

```
/* C Program to explain Reference operator (&) and
Dereference operator (*) */

#include<stdio.h>
#include<conio.h>
void main()
{
  int a = 10, *ptr ;
  clrscr();
  ptr = &a ;
  printf("Address of variable a = %u\n", ptr) ;
  printf("Value of variable a = %d\n", *ptr) ;
  printf("Address of variable ptr = %u\n", &ptr) ;
}
```

Execution:

Output:

```
Address of variable a = 65524
Value of variable a = 10
Address of variable ptr = 65526
```

In the above example program, variable a is a normal variable and variable ptr is a pointer variable. Address of variable a is stored in pointer variable ptr. Here ptr is used to access the address of variable a and $*ptr$ is used to access the value of variable a.

Memory Allocation of Pointer Variables

Every pointer variable is used to store the address of another variable. Memory address of any memory location is an unsigned integer value.

```
/* C Program to explain memory location in Pointers */

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

APEC

```
    char a;
    int x;
    a = „X";
    x = 549;
    printf("%c is at address = %u\n",a, &a) ;
    printf("%d is at address = %u\n",x, &x);
}
```

Execution:

Output:

```
X is at address = 5482
549 is at address = 5484
```

```
/* C program to explain handling of pointers        */

#include <stdio.h>
void main()
{
    int *pc;
    int c;

    c=22;
    printf("Address of c:%u\n",&c);
    printf("Value of c:%d\n\n",c);

    pc=&c;
    printf("Address of pointer pc:%u\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);

    c=11;
    printf("Address of pointer pc:%u\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);

    *pc=2;
    printf("Address of c:%u\n",&c);
    printf("Value of c:%d\n\n",c);
}
```

APEC

Execution:

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

Chain of pointers

A pointer to a pointer is a form of multiple in-directions, 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.



Figure 3.12 Pointer to Pointer

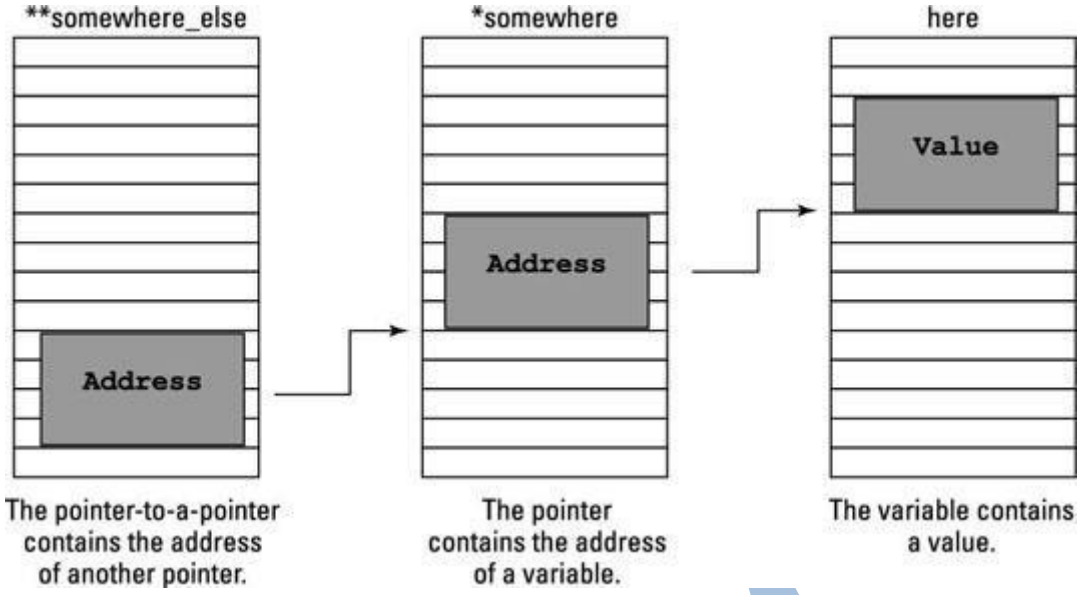


Figure 3.12 Pointer to Pointer – In depth

Syntax:

```
datatype **pointerName ;
```

A variable that is a pointer to a pointer must be declared 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;*

```
/* C program to explain Pointers to Pointers */

#include <stdio.h>
void main ()
{
  int var;
  int  *ptr;
  int  **pptr;

  var = 3000;
  printf("Value of var = %d\n", var );
```

```
   /* take the address of var */
  ptr = &var;
  printf("Address of 3000 =  %u\n", ptr );
  printf("Value available at *ptr = %d\n", *ptr );

   /* take the address of ptr using address of operator & */
  pptr = &ptr;
  printf("Address of pointer =  %u\n", pptr );
  /* take the value using pptr */
  printf("Value available at **pptr = %d\n", **pptr);
}
```

Execution:

Output:

```
Value of var = 3000
Address of 3000 = 7956
Value available at *ptr = 3000
Address of pointer =  6598
Value available at **pptr = 3000
```

Pointers Arithmetic Operations in C

Pointer variables are used to store address of variables. Address of any variable is an unsigned integer value i.e., it is a numerical value. Arithmetic operations can be performed on pointer values. But the result depends on the amount of memory required by the variable to which the pointer is pointing.

The following arithmetic operations are performed on pointers.

- Adding a number to pointer.
- Subtracting a number form a pointer.
- Incrementing a pointer.
- Decrementing a pointer.
- Subtracting two pointers.

Adding Numbers to Pointers

Adding a number N to a pointer leads the pointer to a new location after adding N times size of data type.

ptr + N = ptr + (N * sizeof(pointer_data_type))

APEC

Let ptr be a 4-byte integer pointer, initially pointing to location 5000.

ptr + 5 = 5000 + 4*5 = 5020. Now, pointer ptr will point at memory location 5020.

Subtracting Numbers from Pointers

Subtracting a number N from a pointer is subtracting N times the data type and moving to the new location.

```
ptr - N =  ptr - (N * sizeof(pointer_data_type))
```

Let ptr be a 6-byte double pointer, initially pointing to location 5000.

ptr - 3 = 5000 - 6*3 = 4982. Now, pointer ptr will point at memory location4982.

Incrementing a Pointer

- "Incrementing a pointer increases its value by the number of bytes of its data type"

Let ptr be an integer pointer which points to the memory location 5000 and size of an integer variable is 32-bit(4 bytes). Now, when we increment pointer ptr

$$ptr++;$$

will point to memory location 5004 because it will jump to the next integer location which is 4 bytes next to the current location. Incrementing a pointer is not same as incrementing an integer value. On incrementing, a pointer will point to the memory location after skipping N bytes, where N is the size of the data type (in this case it is 4).A character (1 bytes) pointer is incremented by 1 bytes.

```
ptr + (sizeof(pointer_data_type))
```

Decrementing a Pointer

- "Decrementing a pointer increases its value by the number of bytes of its data type"

$$ptr--;$$

ptrwill point to 4996.

```
ptr - (sizeof(pointer_data_type))
```

Subtracting Pointers

The difference between two pointer returns indicates. It gives the total number of elements between two pointers. If an integer pointer '$ptr1$' points at memory location 0 and integer pointer '$ptr2$' points at memory location 10008, the result of $ptr2 - ptr1$ is 2.

```
/* C program to explain Pointer Arithmetic Operations      */

#include <stdio.h>
void main()
{
   int int_var = 10, *int_ptr;
   char char_var = 'A', *char_ptr;
   float float_val = 4.65, *float_ptr;

   /* Initialize pointers */
    int_ptr = &int_var;
    char_ptr = &char_var;
    float_ptr = &float_val;
    printf("Address of int_var = %u\n", int_ptr);
    printf("Address of char_var = %u\n", char_ptr);
    printf("Address of float_var = %u\n\n", float_ptr);

    /* Incrementing pointers */
    int_ptr++;
    char_ptr++;
    float_ptr++;
    printf("After increment address in int_ptr = %u\n", int_ptr);
    printf("After increment address in char_ptr = %u\n", char_ptr);
    printf("After increment address in float_ptr = %u\n\n", float_ptr);

   /* Adding 2 to pointers */
   int_ptr = int_ptr + 2;
   char_ptr = char_ptr + 2;
   float_ptr = float_ptr + 2;
   printf("After addition address in int_ptr = %u\n", int_ptr);
   printf("After addition address in char_ptr = %u\n", char_ptr);
   printf("After addition address in float_ptr = %u\n\n", float_ptr);
}
```

Execution:

APEC

Output:

```
Address of int_var = 2293300
Address of char_var = 2293299
Address of float_var = 2293292
After increment address in int_ptr = 2293304
After increment address in char_ptr = 2293300
After increment address in float_ptr = 2293296

After addition address in int_ptr = 2293312
After addition address in char_ptr = 2293302
After addition address in float_ptr = 2293304
```

Pointers and Arrays

In c programming language, when an array is declared, the compiler allocates required amount of memory and creates constant pointer with array name and stores the base address to that pointer. The address of first element of an array is called as base address of that array.The array name itself acts as pointer to the first element of that array.

$$int\ names[6];$$

That For the above declaration, the compiler allocates 12 bytes of memory and the address of first memory location (i.e. $names[0]$) is stored in a constant pointer called $names$. address means in the above example, $names$ is a pointer to $names[0]$.

```
/* C Program to explain Arrays and Pointers */
#include<stdio.h>
void main()
{
  int names[6] = {89, 45, 58, 72, 90, 93} ;
  int *ptr ;

  ptr = names ;
  printf("Base Address of 'names' array = %u\n", ptr) ;
}
```

Execution:

Output:

```
Base Address of 'names' array = 8956
```

APEC

Rules

- An array name is a constant pointer.
- Array name can be used to access the address and value of all the elements of that array.
- Since array name is a constant pointer we can't modify its value.

Consider the following example statements.

$$ptr = names + 2;$$

Here, the pointer variable "ptr" is assigned with address of "$names$[2]" element.

$$printf("Address\ of\ 'names[4]' = \%u", names + 4);$$

The above $printf$ statement displays the address of element "$names$[4]".

$$printf("Value\ of\ 'names[0]' = \%d", *names);$$

$$printf("Value\ of\ 'names[3]' = \%d", *(names + 3));$$

In the above two statements, first printf statement prints the value 89 (i.e., value of names[0]) and the second printf statement prints the value 72 (i.e., value of names[3]).

$$names++;$$

The above statement generates compilation error because the array name acts as a constant pointer. So we can't change its value.

In the above example program, the array name names can be used as follows...

$names\ is\ same\ as\ \&names[0]$	$*names\ is\ same\ as\ names[0]$
$names + 1\ is\ same\ as\ \&names[1]$	$*(names + 1)\ is\ same\ as\ names[1]$
$names + 2\ is\ same\ as\ \&names[2]$	$*(names + 2)\ is\ same\ as\ names[2]$
$names + 3\ is\ same\ as\ \&names[3]$	$*(names + 3)\ is\ same\ as\ names[3]$
$names + 4\ is\ same\ as\ \&names[4]$	$*(names + 4)\ is\ same\ as\ names[4]$
$names + 5\ is\ same\ as\ \&names[5]$	

APEC

	* $(names + 5)$ $is\ same\ as$ $names$[5]

APEC

```
/* C  program to explain Arrays and Pointers */

#include <stdio.h>
void main()
{
  int i;
  int a[5] = {1, 2, 3, 4, 5};
  int *p = a;  // same as int*p = &a[0]
  for (i=0; i<5; i++)
   {
     printf("%d", *p);
     p++;
  }
}
```

Execution:

Output:

1 2 3 4 5

```
/* C Program to find Sum of numbers using Arrays and Pointers
*/

#include <stdio.h>
void main()
{
  int i, classes[6],sum = 0;
  printf("Enter 6 numbers:\n");
  for(i = 0; i < 6; ++i)
  {
      // (classes + i) is equivalent to &classes[i]
      scanf("%d",(classes + i));

      // *(classes + i) is equivalent to classes[i]
      sum += *(classes + i);
  }
```

APEC

```
    printf("Sum = %d", sum);
}
```

Execution:

Input

```
Enter 6 numbers:
2
3
4
5
3
4
```

Output:

```
Sum = 21
```

Pointers to Multi-Dimensional Array

In case of multi-dimensional array also the array name acts as a constant pointer to the base address of that array.

$$int\ marks[3][3];$$

Here, the array name marks acts as constant pointer to the base address (address of marks [0][0]) of that array. The element marks[1][2] is accessed as *(*(marks + 1) + 2).

Pointers and Strings

A String is a sequence of characters stored in an array. A string always ends with null ('\0') character. A pointer to array of characters or string can be looks like the following:

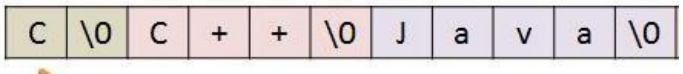


Figure 3.12 Pointers and strings

```
/* C program to explain Pointers and Strings */

#include <stdio.h>
void main()
{
    int i;
    char *lang[4] = {"C","C++","Java"};
```

```
    for(i = 0; i < 3; i++)
      printf("%s\n", lang[i]);
 }
```

Execution:

Output:

```
C
C++
Java
```

In the above program, a pointer array of character data type is declared and strings like `"C","C++","Java"` are initialized to the pointer array (*lang[]). Note that the size of the array is not declared as it is of character pointer type.

Array of Pointers

One important use of pointers is in handling of a table of strings.Consider the following array of strings:

$$char\ lang[3][20] = \{C, C++, Java\}$$

It means that the *lang* is a array containing 3 names, each with a maximum length of 25 characters (including null character). The total space needed for the *lang* array is 60 bytes.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
lang[0]	C	\0																		
lang[1]	C	+	+	\0																
lang[2]	J	a	v	a	\0															

Instead of making each row a fixed number of characters, make it a pointer to a string of varying length.

$$char\ *\ lang[3]= \{C, C++, Java\}$$

declares*lang* to an array of 3 pointers.
This declaration needs only 11 bytes.

	1	2	3	4	5
lang[0]	C	\0			
lang[1]	C	+	+	\0	
lang[2]	J	a	v	a	\0

```
/* C Program to Sort names using Pointers */

#include <stdio.h>
void main()
{
   char *fruit[] = {"apricot","banana","pineapple","apple",
                    "persimmon","pear","blueberry\n"};
   char *temp;
   int a,b,x;

   for(a=0;a<6;a++)
     for(b=a+1;b<7;b++)
      if(*(fruit+a) > *(fruit+b))
      {
        temp = *(fruit+a);
        *(fruit+a) = *(fruit+b);
        *(fruit+b) = temp;
      }

   for(x=0;x<7;x++)
      puts(fruit[x]);
  }
```

Execution:

Output:

```
Apricot
Banana
Pineapple
apple
persimmon
pear
blueberry
```

Benefits of using pointers:

APEC

- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allow us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.

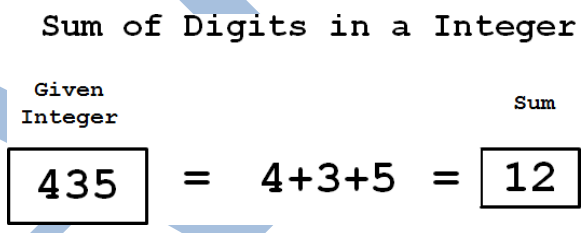
Drawbacks of using pointers:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Additional Programs from previous Question Papers

/* C Program to Compute the Sum of Digits of a given Integer*/

Example:




```
#include <stdio.h>
void main()
{
    int num, no, digit, sum = 0;
    printf("Enter the number \n");
    scanf("%d", &num);
    no = num;
    while (num> 0)
    {
        digit = num % 10;
```

```
        sum  = sum + digit;
        num /= 10;
    }
    printf("Given number = %d\n", no);
    printf("Sum of the digits of %d = %d\n", sum);
 }
```

Execution:

Input:

```
Enter the number
167
```

Output:

```
Given number = 167
Sum of the digits of 167 = 14
```

```
/* C Program to Check if a Given String is Palindrome */
/*
Define Palindrome:
A string is said to be palindrome if reverse of the string is
same as string. Some palindrome strings examples are "dad",
"radar", "madam" etc.                                        */
```

Example:




```
#include <stdio.h>
#include <string.h>
void main()
{
    char string1[20];
```

```
    int i, length;
    int flag = 0;

    printf("Enter a string:\n");
    scanf("%s", string1);
    length = strlen(string1);

    for(i=0;i < length ;i++)
     {
        if(string1[i] != string1[length-i-1])
         {
            flag = 1;
            break;
         }
      }
    if (flag)
       printf("%s is not a palindrome\n", string1);
    else
       printf("%s is a palindrome\n", string1);
  }
```

Execution:

Input:

```
Enter a string: 12
Enter a string: dad
```

Output:

```
12 is not a palindrome
dad is a palindrome
```

/* C program for Using recursion, find the sum of the series:

$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \cdots$ */

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

APEC

```
{
  int power(int,int);
  int fact(int);
  float sum(int,int);
  void main()
  {
   int x,n;
   float f;
   printf("Enter the 'x' value:\n");
   scanf("%d",&x);
   printf("Enter the 'n' value:\n");
   scanf("%d",&n);

   f=sum(x,n);
   printf("\n sum of the series:%f\n",f);
  }

int power(intx,int i)
{
  int p;
  if(i==0)
     p=1;
  else
    p=power(x,i-1)*x;
  return p;
}

int fact(int x)
{
   int f;
   if(x==0)
     f=1;
   else
     f=fact(x-1)*x;
   return f;
}

float sum(intx,int n)
{
   int i;
   float s,s1=0,s2,s3;
```

APEC

```
    for(i=0;i<n;i++)
    {
      s2=power(x,i);
      s3=fact(n);
      s=s2/s3;
      s1=s1+s;
   }
   return s1;
   }
}
```

Execution:

Input:

```
Enter the 'x' value: 2
Enter the 'n' value: 3
```

Output:

```
sum of the series: 1.166667
```

```
/* C Program to perform sum = 1+ (1+2) + (1+2+3) + … + (1+2…+n)*/

#include <stdio.h>
void main()
{
  int n, sum,i,j;
  printf("Please enter an integer, n =\n ");
  scanf("%d", &n);

  for(i=1;i<=n;i++)
     for(j=1;j<=i;j++)
          sum = sum + n;
   printf("sum = %d\n", sum);
}
```

Execution:

Input:

```
Please enter an integer, n = 2
```

APEC

Output:

```
sum = 4
```

/* C Program to Compute the Sum of the series */

Sum of the series: $1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

```
#include <stdio.h>
void main()
{
   int n,i;
   int sum=0;
   printf("Enter the n i.e. max values of series:\n ");
   scanf("%d",&n);

   sum = (n * (n + 1) * (2 * n + 1 )) / 6;
   printf("Sum of the series : %d\n", sum);
 }
```

Execution:

Input:

```
Enter the n i.e. max values of series: 5
```

Output:

```
Sum of the series: 55
```

/* C Program to Compute the Sum of the series */

Sum of the series: $1^2 + 3^2 + 5^2 \ldots + n^2$

```
#include <stdio.h>
void main()
{
   int i,n,sum=0;
```

APEC

```
    printf("1^2+3^2+5^2+…..+n^2");
    printf("Enter value of n:\n");
    scanf("%d",&n);

    for(i=1;i<=n;i+=2)
        sum = sum + (i*i);
    printf("Sum of the series :%d\n",sum);
 }
```

Execution:

Input:

```
Enter value of n: 2
```

Output:

```
Sum of the series : 5
```

/* C Program to Compute the Sum of the series */

Sum of the series: $1 + 3 + 5 + \ldots n$

```
#include <stdio.h>
void main()
{
    int i,n,sum=0;
    printf("1 + 3 + 5 + ….. + n");
    printf("Enter value of n:\n");
    scanf("%d",&n);

    for(i=1;i<=n;i+=2)
        sum = sum + i;
   printf("Sum of the series :%d\n",sum);
    }
```

Execution:

Input:

```
Enter value of n: 9
```

Output:

```
Sum of the series : 165
```

/*C Program to find Factorial of a given number using Iteration */

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

APEC

```
    int i,n = 5;

    for (i =1; i < n; i++)
        fact = fact * i
    printf("Factorial = %d\n", fact);
}
```

Execution:

output:

```
Factorial = 120
```

/* C Programto find largest and smallest element in an array using functions*/

Example:

Largest and Smallest element

Given numbers: 56,213,5,98,902

Smallest Element	Largest Element
5	902

```
#include <stdio.h>
void main()
{
  int arr[30],size,largest,i;
  printf("Enter the size of the array :\n ");
  scanf("%d",&size);
  printf("Enter the numbers:\n ");

  for(i=0;i<size;i++)
      scanf("%d",&arr[i]);

  largest = find_largest(arr,size);
  printf("\nThe largest element is: %d\n",largest);

  smallest = find_smallest(arr,size);
  printf("\nThesmallest element is: %d\n",smallest);
```

```
 }

int find_largest(int arr1[],int size1)
{
  int large,i;
  large=arr1[0];
  for(i=1;i<size1;i++)
      if(arr1[i] >large)
         large =arr1[i];
  return(large);
 }

int find_smallest(int arr1[],int size1)
{
  int small,i;
  small=arr1[0];
  for(i=1;i<size1;i++)
     if(arr1[i] < small)
        small =arr1[i];
  return(small);
 }
```

Execution:

Input:

```
Enter the size of the array: 4
Enter the numbers: 22 55 11 44
```

Output:

```
The largest element is: 55
The smallest element is: 11
```

APEC