UNIT III FUNCTIONS AND POINTERS 9

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

4.1 Functions

A function is a subprogram of one or more statements that performs a specific task when called.

Advantages of Functions:

1. Code reusability
2. Better readability
3. Reduction in code redundancy
4. Easy to debug & test.

Classification of functions:

- Based on who develops the function
- Based on the number of arguments a function accepts

1. Based on who develops the function

There are two types.

1. Library functions
2. User-defined functions

1. Library functions [Built-in functions]

Library functions are predefined functions. These functions are already developed by someone and are available to the user for use. Ex. printf( ), scanf( ).

2. User-defined functions

User-defined functions are defined by the user at the time of writing a program. Ex. sum( ), square( )

4.2 Declaration of Function (Function prototype)

All the function need to be declared before they are used. (i.e. called)

General form:

returntype functionname (parameter list or parameter type list);

a) Return type – data type of return value. It can be int, float, double, char, void etc.
b) Function name – name of the function
c) Parameter type list –It is a comma separated list of parameter types. Parameter names are optional.

Ex: **int add(int, int);**

or

int add(int x, int y);

Function declaration must be terminated with a semicolon(**;**).

4.3 FUNCTION DEFINITION

It is also known as function implementation. Every function definition consists of 2 parts:

1. Header of the function
2. Body of the function

1. Header of the function

General form:

```
returntype functionname (parameter list)
```

1. The header of the function is not terminated with a semicolon.
2. The return type and the number & types of parameters must be same in both function header & function declaration.

2. Body of the function

- It consists of a set of statements enclosed within curly braces.
- The return statement is used to return the result of the called function to the calling function.

General form of Function Definition

```
returntype functionname (parameter list)
{
        statements;
        return (value);
}
```

4.4 FUNCTION CALL:

- A function can be called by using its name & actual parameters.
- It should be terminated by a semicolon (**;**).

4.4.1. Working of a function

```
void main()
{
int x,y,z;
int abc(int, int, int)  // Function declaration
…..
…..
abc(x,y,z) // Function Call
…      Actual arguments
…
}

int abc(int i, int j, int k)   // Function definition
```

```
{            Formal arguments
…….
….
return (value);
}
```

Calling function – The function that calls a function is known as a calling function.
Called function – The function that has been called is known as a called function.
Actual arguments – The arguments of the calling function are called as actual arguments.
Formal arguments – The arguments of called function are called as formal arguments.

Steps for function Execution:

1. After the execution of the function call statement, the program control is transferred to the called function.
2. The execution of the calling function is suspended and the called function starts execution.
3. After the execution of the called function, the program control returns to the calling function and the calling function resumes its execution.

FUNCTION PROTOTYPES:

4.4.2. Classification of Function based on arguments (Function Invocation)

Depending upon their inputs and outputs, functions are classified into:

1. Function with no input and output or (With no arguments and no return values)
2. Function with inputs and no output or (With arguments and no return values)
3. Function with input and one output or (With arguments and one return value)
4. Function with input and outputs or (With arguments and more than one return values)

1. Function with no input and output

- This function doesn't accept any input and doesn't return any result.
- These are not flexible.

Example program:

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

void show( )
{
        printf("Hai \n");
}

Output:
Hai
```

No arguments are passed.
No values are sent back.

2. Function with inputs and no output

Arguments are passed through the calling function. The called function operates on the values but no result is sent back.

Example program:

```
#include<stdio.h>
void main()
{
        int a;
        void show( );
        printf("Enter the value for a \n");
        scanf("%d", &a);
        show(a);
}                               Arguments are passed.
                                No values are sent back.

void show(int x)
{
        printf("Value =%d", x);
}
```

Output:

```
Enter the value for a
10
Value = 10
```

3. Function with input and one output

- Arguments are passed through the calling function i.e.) the actual argument is copied to the formal argument.
- The called function operates on the values.
- The result is returned back to the calling function.
- Here data is transferred between calling function and called function.
- A function can return only one value. We can't return more than one value by writing multiple return statements.

Example program:

```
#include<stdio.h>
void main()
{
        int r;
        float area;
        float circlearea(int);
        printf("Enter the radius \n");
        scanf("%d",&r);
        area=circlearea(r);
        printf("Area of a circle =%d\n", area);
}                                               Arguments are passed
int circlearea(int r1)                          Result is sent back
{
        return 3.14 * r1 * r1;
        }
```

Output:
Enter the radius
2
Area of circle = 12.000

4. Function with input and outputs

More than one value can be indirectly returned to the calling function by making use of pointers.

E.g. Program:

Call by reference program

4.4.3 Pass by value & Pass by reference

Argument passing methods in 'C' are,

1. Pass by value
2. Pass by reference

1. Pass by value
   - In this method the values of actual arguments are copied to formal arguments.
   - Any change made in the formal arguments does not affect the actual arguments.
   - Once control, return backs to the calling function the formal parameters are destroyed.

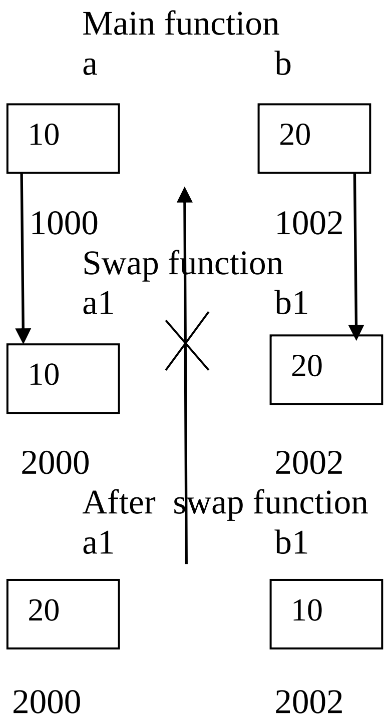
E.g. Program:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
         int a,b;
         void swap(int ,int);
         a=10;
         b=20;
         printf("\n Before swapping: a = %d  and b = %d",a,b);
         swap(a, b);
         printf("\n After swapping: a= %d and b= %d",a,b);
         getch();
        }

void swap(int a1,int b1)
        {
                int temp;
                temp=a1;
                a1=b1;
                b1=temp;
        }
```

OUTPUT:

```
Before swapping: a =10 and b =20
After swapping: a =10 and b = 20
```



2. Pass by reference

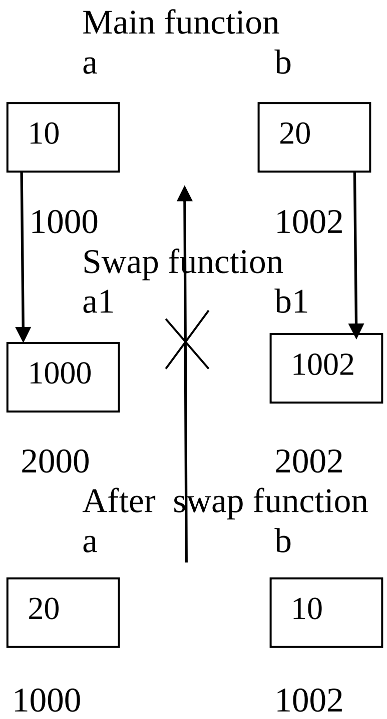
- In this method, the addresses of the actual arguments are passed to formal argument.
- Thus formal arguments points to the actual arguments.
- So changes made in the arguments are permanent.

Example Program:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
          int a,b;
          void swap(int *,int *);
          a=10;
          b=20;
          printf("\n Before swapping: a= %d and b= %d",a,b);
          swap(&a,&b);
          printf("\n After swapping: a= %d and b= %d",a,b);
          getch();
        }
void swap(int *a1,int *b1)
        {
        int t;
        t = *a1;
        *a1 = *b1;
        *b1 = t;
        }
```

OUTPUT:

```
Before swapping: a = 10 and b = 20
After swapping: a = 20 and b = 10
```



a1, b1 points to a and b.

4.5. Built in Functions(String Functions, Math Functions):

Library functions are predefined functions. These functions are already developed by someone and are available to the user for use.

Declaration:

The declarations of library functions are available in the respective header files. To use a library function, the corresponding header file must be included.

Library of Mathematical functions.

These are defined in math.h header file.

Example:

1	double cos(double x)- Returns the cosine of a radian angle x
2	double sin(double x)- Returns the sine of a radian angle x.
3	double exp(double x)- Returns the value of e raised to the xth power
4	double log(double x) Returns the natural logarithm (base-e logarithm) of x.
5	double sqrt(double x) Returns the square root of x.
6	double pow(double x, double y) Returns x raised to the power of y.

Library of standard input & output functions

Header file: stdio.h

Example:

1	printf()	This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen
2	scanf()	This function is used to read a character, string, and numeric data from keyboard.
3	getc()	It reads character from file
4	gets()	It reads line from keyboard

Library of String functions:

Header file: string.h

Example:

Functions	**Descriptions**
strlen()	Determines the length of a String
strcpy()	Copies a String from source to destination
strcmp()	Compares two strings
strlwr()	Converts uppercase characters to lowercase
strupr()	Converts lowercase characters to uppercase

String Operations:(Length, Compare, Concatenate, Copy):

Definition:

The group of characters, digits, & symbols enclosed within double quotes is called as Strings. Every string is terminated with the NULL (‘\0’) character.

E.g. “INDIA” is a string. Each character of string occupies 1 byte of memory. The last character is always ‘\0’.

Declaration:

String is always declared as character arrays.

Syntax

```
char stringname[size];
```

E.g. char a[20];

Initialization:

We can use 2 ways for initializing.

1. By using string constant

   E.g. char str[6]= “Hello”;

2. By using initialisation list

   E.g. char str[6]={‘H’, ‘e’, ‘l’, ;l’, ;o’, ‘\0’};

String Operations or String Functions

These functions are defined in **string.h** header file.

1. **strlen() function**

   It is used to find the length of a string. The terminating character (‘\0’) is not counted.

   Syntax

   ```
   temp_variable = strlen(string_name);
   ```

   E.g.

s= “hai”;
strlen(s)-> returns the length of string s i.e. 3.

2. **strcpy() function**

It copies the source string to the destination string
Syntax

```
strcpy(destination,source);
```

E.g.
s1=“hai”;
s2= “welcome”;
strcpy(s1,s2); -> s2 is copied to s1. i.e. s1=welcome.

3. **strcat() function**

It concatenates a second string to the end of the first string.
Syntax

```
strcat(firststring, secondstring);
```

E.g.
s1=“hai ”;
s2= “welcome”;
strcat(s1,s2);   -> s2 is joined with s1. Now s1 is hai welcome.

E.g. Program:

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char str1[20] = "Hello";
  char str2[20] = "World";
  char str3[20];
  int  len ;
  strcpy(str3, str1);
  printf("Copied String=  %s\n", str3 );
  strcat( str1, str2);
  printf("Concatenated String is= %s\n", str1 );
  len = strlen(str1);
  printf("Length of string str1 is= %d\n", len );
  return 0;
}
```

Output:

```
Copied String=Hello
Concatenated String is=HelloWorld
Length of string str1is 10
```

4. **strcmp() function**

It is used to compare 2 strings.
Syntax

```
temp_varaible=strcmp(string1,string2)
```

- If the first string is greater than the second string a positive number is returned.

- If the first string is less than the second string a negative number is returned.
- If the first and the second string are equal 0 is returned.

5. **strlwr() function**

   It converts all the uppercase characters in that string to lowercase characters.

   Syntax

```
strlwr(string_name);
```

E.g.

```
str[10]= "HELLO";
strlwr(str);
puts(str);
```

Output: hello

6. **strupr() function**

   It converts all the lowercase characters in that string to uppercase characters.

   Syntax

```
strupr(string_name);
```

E.g.

```
str[10]= "HEllo";
strupr(str);
puts(str);
```

Output: HELLO

7. **strrev() function**

   It is used to reverse the string.

   Syntax

```
strrev(string_name);
```

E.g.

```
str[10]= "HELLO";
strrev(str);
puts(str);
```

Output: OLLEH

String functions

Functions	**Descriptions**
strlen()	Determines the length of a String
strcpy()	Copies a String from source to destination
strcmp()	Compares two strings
strlwr()	Converts uppercase characters to lowercase
strupr()	Converts lowercase characters to uppercase

strdup()	Duplicates a String
strstr()	Determines the first occurrence of a given String in another string
strcat()	Appends source string to destination string
strrev()	Reverses all characters of a string

Example: String Comparison

```
void main()
{
char s1[20],s2[20];
int val;
printf(“Enter String 1\n”);
gets(s1);
printf(“Enter String 2\n”);
gets (s2);
val=strcmp(s1,s2);
if (val==0)
printf(“Two Strings are equal”);
else
printf(“Two Strings are not equal”);
getch();
}
```

Output:

```
Enter String 1
Computer
Enter String 2
Programming
Two Strings are not equal
```

String Arrays

They are used to store multiple strings. 2-D char array is used for string arrays.

Declaration

```
char arrayname[rowsize][colsize];
```

E.g.

```
char s[2][30];
```

Here, s can store 2 strings of maximum 30 characters each.

Initialization

2 ways

1. Using string constants
   ```
   char s[2][20]={“Ram”, “Sam”};
   ```
2. Using initialization list.
   ```
   char s[2][20]={ {‘R’, ‘a’, ‘m’, ‘\0’},
                   {‘S’, ‘a’, ‘m’, ‘\0’}};
   ```

E.g. Program

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

```
char s[3][20];
printf("Enter Names\n");
for(i=0;i<3;i++)
scanf("%s", s[i]);
printf("Student Names\n");
for(i=0;i<3;i++)
printf("%s", s[i]);
}
```

4.5 RECURSION

A function that calls itself is known as a recursive function.

Direct & Indirect Recursion:

Direct Recursion:

A function is directly recursive if it calls itself.

```
A( )
{
….
….
A( );// call to itself
….
}
```

Indirect Recursion:

Function calls another function, which in turn calls the original function.

```
A( )
{
…
…
B( );
…
}
B( )
{
…
…
A( );// function B calls A
…
}
```

Consider the calculation of 6! (6 factorial)
ie 6! = 6 * 5 * 4 * 3 * 2 * 1
6! = 6 * 5!
6! = 6 * (6 - 1)!
n! = n * (n - 1)!

E.g. Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int fact(int);
        int n,f;
        printf(“Enter the number \n”);
        scanf(“%d”,&n);
        f=fact(n);
        printf(“The factorial of a number =%d”,f);
        getch();
}
int fact(int n)
{
        if(n==1)
        return(1);
        else
        return n*fact(n-1);
}
```

OUTPUT

```
Enter the number to find the factorial
5
The factorial of a number=120
```

Pattern of Recursive Calls:

Based on the number of recursive calls, the recursion is classified in to 3 types. They are,

1. Linear Recursion
   Makes only one recursive call.
2. Binary Recursion
   Calls itself twice.
3. N-ary recursion
   Calls itself n times.

4.6EXAMPLE PROGRAM:

COMPUTATION OF SINE SERIES:

program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i, n;
   float x, sum, t;
   clrscr();
   printf(" Enter the value for x : ");
   scanf("%f",&x);
   printf(" Enter the value for n : ");
   scanf("%d",&n);
   x=x*3.14159/180;
   t=x;
   sum=x;
   /* Loop to calculate the value of Sine */
   for(i=1;i<=n;i++)
   {
      t=(t*(-1)*x*x)/(2*i*(2*i+1));
```

```
        sum=sum+t;
    }

    printf(" The value of Sin(%f) = %.4f",x,sum);
    getch();
}
```

OUTPUT:

```
Enter the value for x :45
Enter the value for n: 5
The value of Sin(0.7853980 = 0.7071
```

SCIENTIFIC CALCULATOR USING BUILT-IN FUNCTIONS:

Program:

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

int main(void)

{
int choice, i, a, b;
float x, y, result;
clrscr();
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(“9. %\n110. log10(x)\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;
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;
case 9:
printf(“Enter X: “);
scanf(“%f”, &x);
printf(“\nEnter Y: “);
scanf(“%f”, &y);
result = (x * y) / 100;
printf(“\nResult: %.2f”, result);
break;
case 10:
printf(“Enter X: “);
```

```
scanf(“%f”, &x);
result = log10(x);
printf(“\nResult: %.2f”, result);
break;
}
} while(choice);
getch();
return 0;
}
```

BINARY SEARCH USING RECURSIVE FUNCTIONS:

Program:

```
#include <stdio.h>
int RecursiveBinarySearching(int arr[], int low, int high, int element)
{
    int middle;
    if(low > high)
    {
        return -1;
    }
    middle = (low + high) / 2;
    if(element > arr[middle])
    {
        RecursiveBinarySearching(arr, middle + 1, high, element);
    }
    else if(element < arr[middle])
    {
        RecursiveBinarySearching(arr, low, middle - 1, element);
    }
    else
    {
        return middle;
    }
}

int main()
{
    int count, element, limit, arr[50], position;
    printf("\nEnter the Limit of Elements in Array:\t");
    scanf("%d", &limit);
    printf("\nEnter %d Elements in Array: \n", limit);
    for(count = 0; count < limit; count++)
    {
        scanf("%d", &arr[count]);
    }
    printf("\nEnter Element To Search:\t");
    scanf("%d", &element);
    position = RecursiveBinarySearching(arr, 0, limit - 1, element);
    if(position == -1)
    {
        printf("\nElement %d Not Found\n", element);
    }
    else
    {
        printf("\nElement %d Found at Position %d\n", element, position + 1);
    }
    return 0;
}
```

Output:

```
Enter the Limit of  Elements in Array: 5

Enter 5 Elements in Array:

1 2 3 4 5

Enter Element To Search:3

Element 3 Found at Position 3
```

4.7. POINTERS

Definition:

A pointer is a variable that stores the address of a variable or a function

Advantages

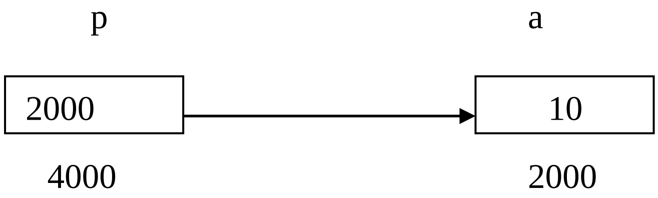
1. Pointers save memory space
2. Faster execution
3. Memory is accessed efficiently.

Declaration

```
datatype     *pointername;
```

E.g)
```
int *p          //p is an pointer to an int
float *fp    //fp is a pointer to a float

int a=10;
int *p=&a;
```



p is an integer pointer & holds the address of an int variable a.

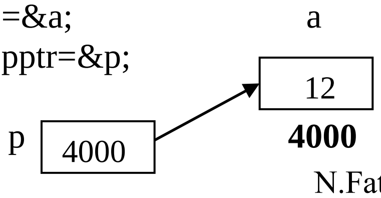
Pointer to pointer

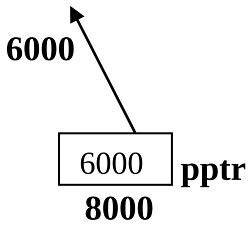
A pointer that holds the address of another pointer variable is known as a pointer to pointer.

E.g.
```
int **p;
```

p is a pointer to a pointer to an integer.

```
int a=12;
int *p=&a;
int **pptr=&p;
```



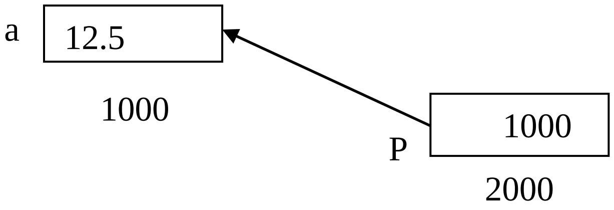


So **pptr=12

Operations on pointers:

1. **Referencing operation**: A pointer variable is made to refer to an object. Reference operator(&) is used for this. Reference operator is also known as address of (&) operator.

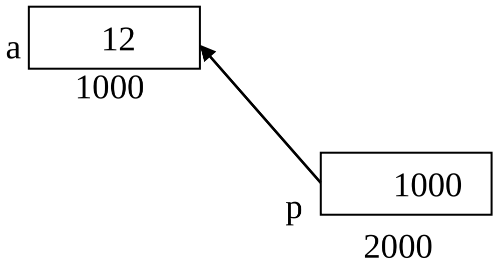
   Eg) float a=12.5;
   float *p;
   p=&a;



2. **Dereferencing a pointer**

   The object referenced by a pointer can be indirectly accessed by dereferencing the pointer. Dereferencing operator (*) is used for this .This operator is also known as indirection operator or value- at-operator

Eg)



```
int b;
int a=12;
int *p;
p=&a;
b=*p;     \\value pointed by p(or)value
                    at 1000=12,
so b=12
```

Example program

```
#include<stdio.h>
void main()
{
        int a=12;
        int *p;
        int **pptr;
        p=&a;
        pptr=&p;
        printf("a value=%d",a);
        printf("value by dereferencing p is %d \n",*p);
```

Note

%p is used for addresses; %u can also be used.

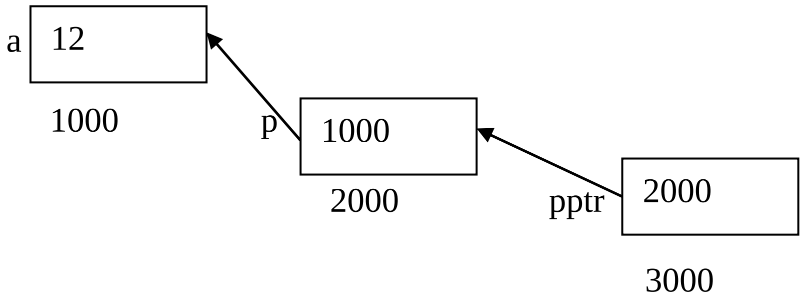
*p=value at p
=value at (1000)=12

*pptr=value at(pptr)
=value at(value at (2000))
=value at (1000)=12

```
        printf(“value by dereferencing pptr is %d \n”,**pptr);
        printf(“value of p is %u \n”,p);
        printf(“value of pptr is %u\n”,pptr);
}
```

Output:

```
a value=12
value by dereferencing p is 12
value by dereferencing pptr is 12
value of p is 1000
value of pptr is 2000
```



4.8. Initialization

1. Pointer can be assigned or initialized with the address of an object.

```
        Eg)     int a=10;
                int *p=&a;
```

2. A pointer to one type cannot be initialized with the address of other type object.

```
        Eg)     int a=10;

                float *p;

                p=&a;  //not possible
```

Because p is a float pointer so it can’t point int data.

4.9. Pointers Arithmetic

Arithmetic operations on pointer variables are also possible.

E.g.) Addition, Increment, Subtraction, Decrement.

1. Addition

(i) An addition of int type can be added to an expression of pointer type. The result is pointer type.(or)A pointer and an int can be added.

Eg) if p is a pointer to an object then

p+1 =>points to next object

p+i=>point s to ith object after p

(ii)Addition of 2 pointers is not allowed.

2. Increment

Increment operator can be applied to an operand of pointer type.

3. Decrement

Decrement operator can be applied to an operand of pointer type.

4. Subtraction

i) A pointer and an int can be subtracted.

ii) 2 pointers can also be subtracted.

S.no	Operator	Type of operand 1	Type of operand 2	Result type	Example	Initial value	Final value	Description
1	+	Pointer to type T	int	Pointer to type T				Result = initial value of ptr +int operand * sizeof (T)
	Eg.	int *	int	int *	p=p+5	p=2000	2010	2000+5*2= 2010
2	++	Pointer to type T	-	Pointer to type T				**Post increment** Result = initial value of pointer **Pre-increment** Result = initial value of pointer + sizeof (T) Value of ptr = Value of ptr +sizeof(T)
	Eg. post increment	float*	-	float*	ftr=p++	ftr=? p=2000	ftr=2000 p=2004	
3	-	Pointer to type T	int	Pointer to type T				Result = initial value of ptr - int operand * sizeof (T)
	E.g.	float*	int	float*	p=p-1	p=2000	1996	2000 – 1 * 4 = 2000-4=1996
4	--	Pointer to type T	-	Pointer to type T				**Post decrement** Result = initial value of pointer **Pre-decrement** Result = initial value of pointer – sizeof(T) Value of ptr = Value of ptr – sizeof(T)
	Eg.pre decrement	float*	-	float*	ftr=--p	ftr=? p=2000	ftr=1996 p=1996	

4.10. Pointers and Arrays

In C language, pointers and arrays are so closely related.

i) An array name itself is an address or pointer. It points to the address of first element (0th element) of an array.

Example

```
#include<stdio.h>
void main()
{
	int a[3]={10,15,20};
	printf("First element of array is at %u\n", a);
	printf("2nd element of array is at %u\n", a+1);
	printf("3nd element of array is at %u\n", a+2);

}
```

10	15	20
1000	1002	1004

Output

```
First element of array is at 1000
2nd element of array is at 1002
3nd element of array is at 1004
```

ii) Any operation that involves array subscripting is done by using pointers in c language.

E.g.) E1[E2]=>*(E1+E2)

Example

```
#include<stdio.h>
void main()
{
	int a[3]={10,15,20};
	printf("Elements are %d %d %d\n", a[0],a[1],a[2]);
	printf("Elements are %d %d %d\n", *(a+0),*(a+1),*(a+2);
}
```

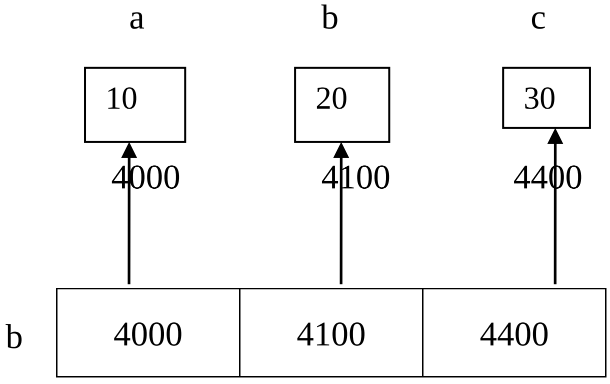
Output:

```
	Elements are 10 15 20
	Elements are 10 15 20
```

Array of pointers

An array of pointers is a collection of addresses. Pointers in an array must be the same type.

```
		int a=10,b=20,c=30;
		int *b[3]={&a,&b,&c};
```



5000 5002 5004

Example Programs:

Sorting of Names:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
static int myCompare (const void * a, const void * b)
{
   return strcmp (*(const char **) a, *(const char **) b);
}
void sort(const char *arr[], int n)
{
   qsort (arr, n, sizeof (const char *), myCompare);
}
int main ()
{
   const char *arr[] = {"GeeksforGeeks", "GeeksQuiz", "CLanguage"};
   int n = sizeof(arr)/sizeof(arr[0]);
   int i;
   printf("Given array is\n");
   for (i = 0; i < n; i++)
   printf("%d: %s \n", i, arr[i]);
   sort(arr, n);
   printf("\nSorted array is\n");
   for (i = 0; i < n; i++)
   printf("%d: %s \n", i, arr[i]);
   return 0;
}
```

Output:

```
Given array is
0: GeeksforGeeks
1: GeeksQuiz
2: CLanguage

Sorted array is
0: CLanguage
1: GeeksQuiz
2: GeeksforGeekss
```

4.11.PARAMETER PASSING:

Function with input and outputs

More than one value can be indirectly returned to the calling function by making use of pointers.

E.g. Program:

Call by reference program

4.11.1 Pass by value & Pass by reference

Argument passing methods in ‘C’ are,

3. Pass by value
4. Pass by reference

2. Pass by value
   - In this method the values of actual arguments are copied to formal arguments.
   - Any change made in the formal arguments does not affect the actual arguments.
   - Once control, return backs to the calling function the formal parameters are destroyed.

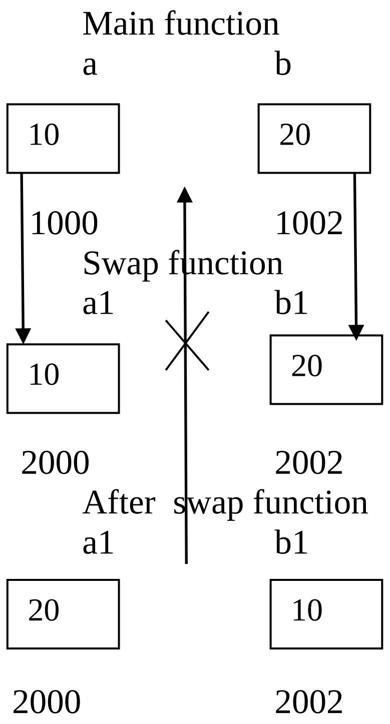
E.g. Program:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
         int a,b;
         void swap(int ,int);
         a=10;
         b=20;
         printf("\n Before swapping: a = %d  and b = %d",a,b);
         swap(a, b);
         printf("\n After swapping: a= %d and b= %d",a,b);
         getch();
        }

void swap(int a1,int b1)
        {
                int temp;
                temp=a1;
                a1=b1;
                b1=temp;
        }
```

OUTPUT:

```
Before swapping: a =10 and b =20
After swapping: a =10 and b = 20
```



2. Pass by reference

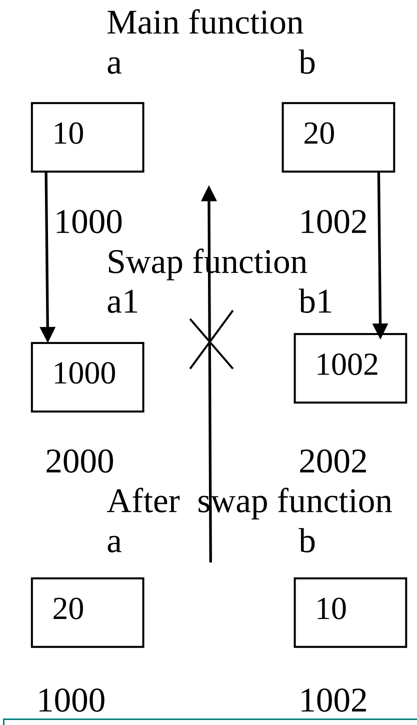
- ➢ In this method, the addresses of the actual arguments are passed to formal argument.
- ➢ Thus formal arguments points to the actual arguments.
- ➢ So changes made in the arguments are permanent.

Example Program:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
          int a,b;
          void swap(int *,int *);
          a=10;
          b=20;
          printf("\n Before swapping: a= %d and b= %d",a,b);
          swap(&a,&b);
          printf("\n After swapping: a= %d and b= %d",a,b);
          getch();
        }
void swap(int *a1,int *b1)
        {
        int t;
        t = *a1;
        *a1 = *b1;
        *b1 = t;
        }
```

OUTPUT:

```
Before swapping: a = 10 and b = 20
After swapping: a = 20 and b = 10
```



a1, b1 points to a and b.

Pass by Value	Pass by Reference
Values of the Actual arguments are Passed to the Formal Arguments.	Addresses of the Actual arguments are passed to the Formal Arguments.
Different Memory Locations are Occupied	Same memory Locations are Occupied.
There is no Possibility of Wrong Data Manipulations	. There is a Possibility of Wrong Data Manipulations.
In this, you are sending a copy of the data.	In this, you are passing the memory address of the data that is stored.
Changes does not affect the actual value.	Changes to the value affect the original data.

4.12.EXAMPLE PROGRAM: SWAPPING OF TWO NUMBERS

Pass by Value:

E.g. Program:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
         int a,b;
         void swap(int ,int);
         a=10;
         b=20;
         printf("\n Before swapping: a = %d  and b = %d",a,b);
         swap(a, b);
         printf("\n After swapping: a= %d and b= %d",a,b);
```

```
            getch();
           }

    void swap(int a1,int b1)
           {
                  int temp;
                  temp=a1;
                  a1=b1;
                  b1=temp;
           }
```

OUTPUT:

```
Before swapping: a =10 and b =20
After swapping: a =10 and b = 20
```

4.13.EXAMPLE PROGRAM:CHANGING THE VALUE OF A VARIABLE USING PASS BY REFERENCE

Pass by Reference:

```
#include<stdio.h>
#include<conio.h>
void main()
      {
        int a,b;
        void swap(int *,int *);
        a=10;
        b=20;
        printf("\n Before swapping: a= %d and b= %d",a,b);
        swap(&a,&b);
        printf("\n After swapping: a= %d and b= %d",a,b);
        getch();
      }
void swap(int *a1,int *b1)
      {
      int t;
      t = *a1;
      *a1 = *b1;
      *b1 = t;
      }
```

OUTPUT:

```
Before swapping: a = 10 and b = 20
After swapping: a = 20 and b = 10
```