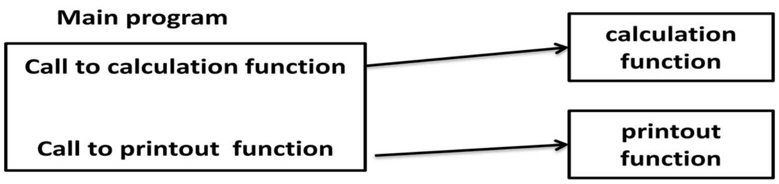
MODULE -3

FUNCTION AND POINTERS

The Concept of a Function (Module)

- ✓ A series of instructions with one specific purpose is called a **module.**
- ✓ A module can be written directly in the main program, but there is another method.
- ✓ We can write a function that consists of the module, then the main body of the program, a single instruction **(known as the call to the function)** utilizes this function in a particular situation.
- ✓ The word **subroutine** or simply **routine** is sometimes used as a synonym for **subprograms** or **functions**.



- ✓ The concept of using module is actually very widespread.
- ✓ Large program can be broken up into a series of well defined **tasks or modules**. Then a separate function can be written for each.
- ✓ We can also reuse the function which is been defined.
- ✓ A large program can be divided into a series of reusable modules and is called as **Modularization.**
- ✓ We are already making use of many library functions such as, **sqrt.** This returns the square root of a given number.

Advantages of Functions

There are two main advantages of using a function.

1) We can separate the mechanics of a function from its particular use in a large program.
   - ✓ **For ex:** the main program does not have to specify the individual steps in finding the square root.
   - ✓ It simply issues a call to the **sqrt** function.

- ✓ In that call, the main program has to specify the value for which it wants to find its square root.
- ✓ This extra information is sent to the function is called a **parameter.**

2) Another advantage is that the set of instructions specifying the task performed by the function is just written once but can be used any number of times.

Highlights: To summarize:

- ➢ A function gives the computer a set of instructions that can be used again with different values.
- ➢ Once the pattern or function is specified, one single statement calls it in a given situation; to apply it three times it takes three calls.
- ➢ To use a function to perform a particular task:
  - Write a function which contains the instructions to perform the particular task.
  - Every time you want to perform the task on actual parameter values, call the function with the values specified.

Using the Library function sqrt

Assume that we have this main program

/* main program which calls the library function sqrt */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
      double w, y;
      w = 16.0;
      y = sqrt(w);
      printf (“%f”,y);
   }
```

- ✓ The first assignment statement gives w the value 16.0 in the next line, this value is sent to the function sqrt.
- ✓ While sqrt is working, the main program is waiting for an answer to be computed(it resumes execution after sqrt computes this answer and sends it back).

- ✓ The function uses some process to find the square root of the number sent to it.
- ✓ In this case, it finds the square root of **16.0,** which is **4.0,** and then sends the answer back to the main program to be put into the variable **y.**
- ✓ At this point the main program resumes execution, prints the value stored in **y**, and stops.

Function Terminology or Components

1) Function Prototyping / Declaration
2) Function Definition
3) Function Call
4) Return Statement

Function Prototyping / Declaration

- ➢ Before using the function, the compiler must know about the number and type of parameters that the function expects to receive and the data type of the value that it will return to the calling program.
- ➢ Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.
- ➢ The general syntax is:

  return_data_type function_name (data_type variable1, data_type variable2, . . .);
- ✓ Here **function_name** is a valid name for the function.
- ✓ A function should have a meaningful name that must clarify the task that it will perform.
- ✓ Every function must have a different name that indicates a particular job that a function does.
- ✓ **return_data_type** specifies the data type of the value that will be returned to the calling function.
- ✓ **data_type variable1, data_type variable2** is a list of variables of specified data types. These are passed from the **calling function** to the **called function** and are known as **arguments or parameters.**
- ✓ After the declaration of every function, there should be a semi colon. If semi colon is missing then compiler will generate an error message.
- ✓ Use of arguments names is optional.

- ✓ **Ex:** int func (int, char, float); **or** int func(int num, char ch, float fnum);

Function Definition

- ✓ When a function is defined, space is allocated for that function in the memory.
- ✓ A function definition comprises of two parts.

1) **Function header**
2) **Function body**

The syntax of a function definition is:

return_data_type function_name (data_type variable1, data_type variable2, . . .)

{

. . . .

statements;

. . . .

return (variable);

}

- ✓ The number and order of arguments in the function header must be same as that given in the function declaration statement.
- ✓ The statements written in pair of { } is known as function body which contains code to perform specific task.
- ✓ The function header is same as function declaration. The only difference between the two is that a function header is not followed by a semicolon.
- ✓ The list of variables in the function header is known as the **formal parameter list.**

Function Call

- ✓ The function call statement invokes the function. When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function.
- ✓ Once the function is executed, the program control passes back to the calling function.

The syntax of a function call is:

function_name (variable1, variable2 . . .);

- ✓ when the function declaration is present before the function call, the compiler can check if the correct number and type of arguments are used in the function call and the returned value, if any is being used reasonably.

Return statement

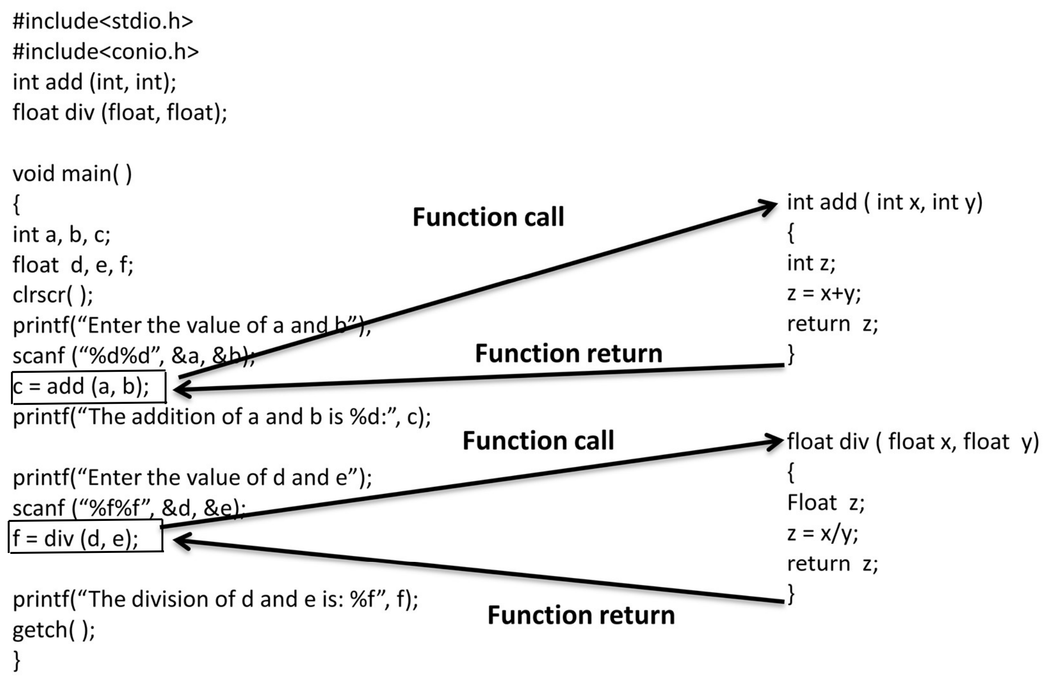
- ✓ The return statement is used to terminate the execution of a function and returns control to the calling function.
- ✓ When the statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- ✓ A return statement may or may not return a value to the calling function.

The syntax of a return is:

return <expression>

- ✓ The expression, if present is converted to the type returned by the function.

/* Example program to add & div two numbers using functions */

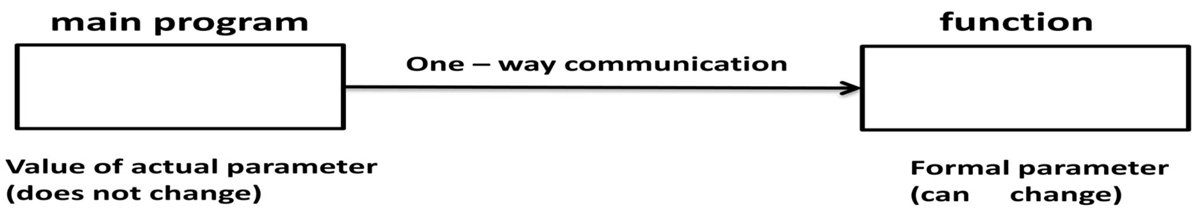

```
#include<stdio.h>
#include<conio.h>
int add (int, int);
float div (float, float);

void main( )
{
int a, b, c;
float  d, e, f;
clrscr( );
printf("Enter the value of a and b");
scanf ("%d%d", &a, &b);
c = add (a, b);
printf("The addition of a and b is %d:", c);

printf("Enter the value of d and e");
scanf ("%f%f", &d, &e);
f = div (d, e);

printf("The division of d and e is: %f", f);
getch( );
}
```

```
int add ( int x, int y)
{
int z;
z = x+y;
return  z;
}
```

```
float div ( float x, float  y)
{
Float  z;
z = x/y;
return  z;
}
```

Parameter Passing Technique --- Call by value

- If a function uses arguments, it must declare variables that accept the values of the arguments. These variables are called as function parameters.
- During execution of the program, the function is sent the ***value*** of the actual parameter, and this value is placed in the formal parameter.
- This method of sending a parameter to a function is called ***passing a parameter by value or parameter transmission by value.***
- Because all parameter is passed by value, if there is a change in the formal parameter inside the function, the value of the corresponding actual parameter does not change.
- In C all parameters are passed by **value.**
- The formal parameters are **dummy parameters** of the function.



Note: In the **function**, you should not refer to the name of an **actual parameter**. Similarly in the **main program**, you should not refer to the name of a **formal parameter**.

/ program to pass argument by value */*

```
#include<stdio.h>
int add(int, int);
void main( )
{
int a,b,c;
printf("Enter the value of a and b:");
scanf ("%d %d", &a, &b);
c = add (a, b);
printf("The result of addition is :%d", c);
}

/*function definition*/
int add (int x, int y)
{
int z;          /* local variable */
z = x + y;
return z;
}
```

Output: The result of Addition is: 30

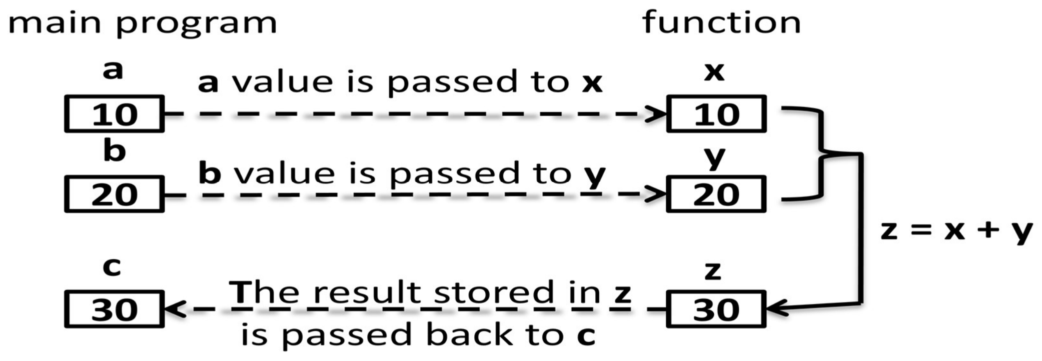


Fig: Graphical representation of pass by value or call by value

Parameter Passing Technique --- Pass by reference

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to, by their arguments.

Let us now call the function swap() by passing values by reference as in the following example –

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
void swap(int *x, int *y) {

   int temp;

   temp = *x; /* save the value of x */
   *x = *y;    /* put y into x */
   *y = temp; /* put temp into y */

   return;
}
```

Output:

```
Before swap, value of a : 100

Before swap, value of b : 200

After swap, value of a : 200

After swap, value of b : 100
```

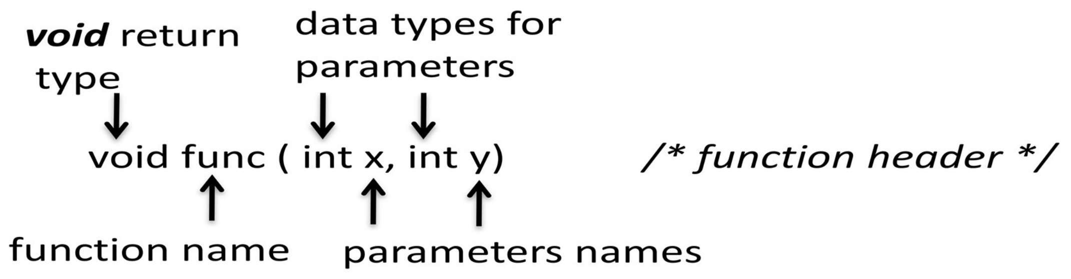
VOID AND PARAMETERLESS FUNCTIONS

- ✓ The basic purpose of each function is to compute and return the answer.

- ✓ However, a function can perform one or more actions (including printing) without returning a particular value. This type is called a void function.
- ✓ Void functions truly are functions.
- ✓ The basic purpose of a function is to perform a well-defined task. Void function does just that.

void function header and return

- ✓ The first line or header of a ***void function*** differs only slightly from the first line of one returning a value.
- ✓ At the beginning of the usual function header is the data type for the answer to be returned; in a void function, the data type is replaced by the keyword ***void*** because no explicit value is returned.\here is an example of a header for a function ***func*** receives two **int** parameter **x** and **y:**



- ❖ There are another feature of void function is the use of ***return*** statement to return a value.
- ✓ In a void function, no value is returned to indicate that control should return to the main program, just use the keyword ***return,*** and then end the function with closing braces.

  return;

  }

Note: the word return can be omitted in a function. The closing brace then ends the function. However, it is better to mark the end of the function with a return statement.

void function prototype and call

- ✓ The function prototype for a void function can just look like the function header with the addition of a semicolon.

- ✓ We can also omit the names of the parameters

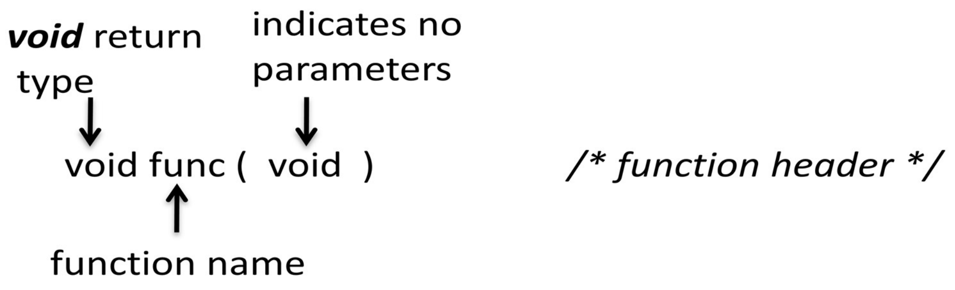
  void printmaxnum (**int, int**); *\/* function prototype *\/*
- ✓ To call a void function, just write its name with the list of all arguments.
- ✓ Since void function does not return a value, we cannot include its call as part of an expression.
- ✓ The only way to call a void function is to write the function name with the list of parameters.

A complete program containing a void function

```
     #include < stdio.h>
          void printmaxnum ( int, int); /* function prototype*/
     main( )
     {
          int a, b;
          a = 5, b = 3;
          printfmaxnum (a, b);          /* function call*/
          }
/*function definition */
     void printmaxnum(int x, int y)
     {
          if ( x > y )
               printf("%d is large ", x );
          else
               printf("%d is large ", y );
     return;
     }
```

Parameter less void function

- ✓ A ***parameter less void*** function does not receive any parameters.
- ✓ Such a void function cannot communicate with the main program. Therefore, it has to perform entirely on its own.
- ✓ The header of the parameter less void function uses the keyword void in two places: at the beginning to indicate that there is no answer to ***return,*** and inside the parentheses to show that there is no parameters.



- ✓ The function header can also omit the keyword void from the parentheses. Here is a header for ***func*** with no parameter.

```
          void func ( )       /* function header */
/*example for parameter less function */
          void printheading ( void );   /* function header with
     no parameter*/
     main ( )
     {
          printheadings ( );   /*   function   call     with   no
     parameter*/
     }
```

Note: in this program there is variables declared or print any result. In fact the only thing it does is to call the void function.

Functions that have no parameters and return an answer

- ✓ It is also possible to write a function that has no parameters but returns an answer. For such a parameter less functions, the function call looks like this:

  varname = func ( ) ;
- ✓ Where varname is the name of a variable holding the result of the function call, and func is the name of the function.
- ✓ Here the function does not receive any parameters, but it does return an answer.

The main program as a void, parameter less function

- ✓ In our programs generally we don't return any value for main ( ). so we write :

  void main ( )
- ✓ But the fact is main ( ) will also take a parameter called as command line arguments.
- ✓ In our example, main has been a void, parameter less function.
- ✓ If we don't use void then compiler will generate a warning saying,"*function doesn't return a value"*.
- ✓ To avoid the warning we have to use void main ( ).
- ✓ The default return type of main ( ) is integer type.

  Ex: ***int main (void)***

  Then the program has to return a value so: ***return 0;*** has to be written.

RECURSION

- ➢ A function which calls itself **or** a function call to a second function which eventually calls the first function is known as ***Recursive function***.
- ➢ Recursion as the name suggest, revolves around a function calling itself.
- ➢ Recursive function is those in which there is at least one function call to itself.
- ➢ Two important conditions which must satisfy by recursion functions are:
  1) Each time a function calls itself it must be neared, in some sense to a solution.
  2) There must be decision criteria for stopping the process or computation.

/* program to find the factorial of a given number using recursion */

```
     #include < stdio.h>
            int fact( int);        /* function prototype*/
     void main( )
     {
int n, result;
printf("Enter the number ");
scanf ("%d", &n);
result = fact (n);
printf("The factorial of a given number is: %d", result);
}

/*  function definition */
int fact ( int num)
{
     if (num = = 0 )
          return 1;
     else
          return ( num * fact( num – 1 )); /* Recursive call */
}
```

Output: Enter the number: 5

The factorial of a given number is: 120

/* program to find Fibonacci Series using recursion in C*/

```
#include<stdio.h>
void printFibonacci(int n)
{
 int n1=0,n2=1,n3;
   if(n>0){
         n3 = n1 + n2;
         n1 = n2;
         n2 = n3;
         printf("%d ",n3);
         printFibonacci(n-1);
    }
}

int main(){
    int n;
    printf("Enter the number of elements: ");
    scanf("%d",&n);
    printf("Fibonacci Series: ");
    printf("%d %d ",0,1);
    printFibonacci(n-2);
//n-2 because 2numbers are already printed
  return 0;
 }
```

Output:

```
Enter Enter the number of elements:15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

POINTERS

- ✓ A pointer is a variable that contains the address of a variable. Pointers are much used in C, because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code.
- ✓ Pointer may point to a variable of any data type **like int, float, char etc.**

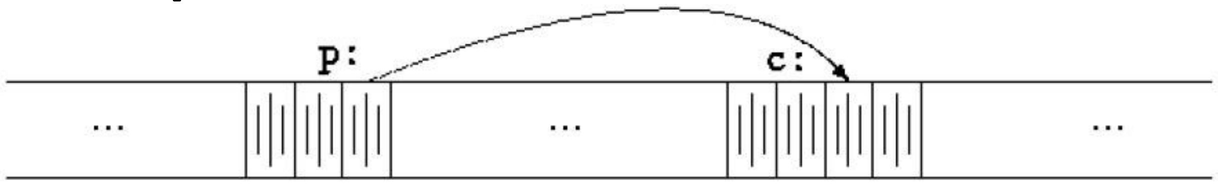
Pointers and Addresses (declaration and Initialization)

- ✓ A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups.
- ✓ One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`.
- ✓ A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `P` is a pointer that points to it, we could represent the situation this way:

datatype *name;

- ✓ The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```



- ✓ Assigns the ***address*** of `c` to the variable `p`, and `p` is said to **``point to"** `c`.
- ✓ The `&` operator only applies to objects in memory: variables and array elements.
- ✓ It cannot be applied to expressions, constant variables.
- ✓ The unary operator `*` is the ***indirection*** **or** ***dereferencing*** operator; when applied to a pointer, it accesses the object the pointer points to.

/* ***Example program demonstrating the concept of pointers*** */

```
#include<stdio.h>
void main ( )
{
int sum = 5;
int *ptr;
ptr = &sum;
printf (" Sum = %d *ptr = %d\n", sum, *ptr );
}
```

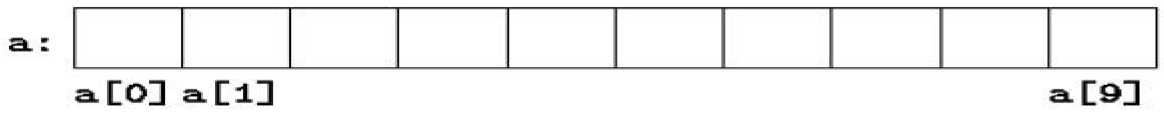
Output: sum = 5, *ptr = 5

Pointers and Arrays

- ✓ In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously.
- ✓ Any operation that can be achieved by **array subscripting** can also be done with **pointers**. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.
- ✓ The declaration

```
int a[10];
```

- ✓ Defines an array of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, ...,`a[9]`.



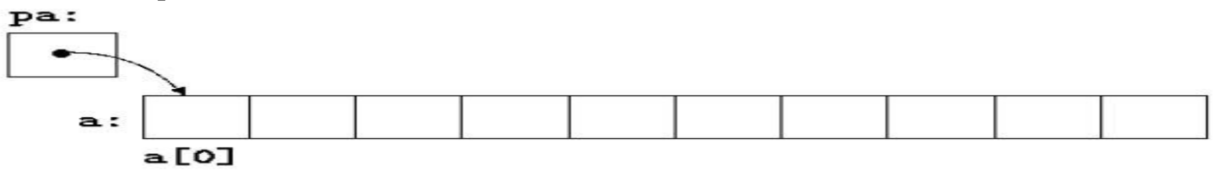
- ✓ The notation `a[i]` refers to the `i`-th element of the array. If `pa` is a pointer to an integer, declared as

```
int *pa;
```

- ✓ then the assignment

```
pa = &a[0];
```

- ✓ sets `pa` to point to element zero of `a`; that is, `pa` contains the address of `a[0]`.



Now the assignment

```
x = *pa;
```

- ✓ Will copy the contents of `a[0]` into `x`.
- ✓ If `pa` points to a particular element of an array, then by definition `pa+1` points to the next element, `pa+i` points `i` elements after `pa`, and `pa-i` points `i` elements before.
- ✓ Thus, if `pa` points to `a[0]`,

```
*(pa+1)
```

- ✓ refers to the contents of `a[1]`, `pa+i` is the address of `a[i]`, and `*(pa+i)` is the contents of `a[i]`.
- ✓ These remarks are true regardless of the type or size of the variables in the array `a`. The meaning of ”adding 1 to a pointer," and by extension,
- ✓ All pointer arithmetic is that `pa+1` points to the next object, and `pa+i` points to the `i`-th object beyond `pa`.
- ✓ By definition, the value of a variable or expression of type array is the address of element zero of the array.

Thus after the assignment

```
pa = &a[0];
```

`pa` and `a` have identical values.

/ **Example program demonstrating the concept of pointers and arrays***/*

```
#include<stdio.h>
   void main ( )
   {
         int num[10];
         int *ptr;
         printf(" Type first element of array:");
         scanf ("%d", &num[0]);
         ptr = &num[0]; /* assign address of first element to pointer variable */
         printf("num[0] is %d stored at memory location : %u", *ptr, ptr);
ptr = num;             /* assign address of first element to pointer variable */
         printf("num[0] is %d stored at memory location : %u", *ptr, ptr);
getch( );
   }
```

<u>Output:</u>

```
Type first element of array: 26
num[0] is :26 stored at memory location :65606
num[0] is :26 stored at memory location :65606
```

<u>**Address Arithmetic(**Pointer Arithmetic **)**</u>

- ✓ Pointers are just like any other variables and hence may be used in expression.
- ✓ It can be incremented, decremented and a value can be added or subtracted to a pointer.
- ✓ However a pointer cannot be **multiplied** or **divided** by another number.
- ✓ Most common arithmetic operation done using pointers is incrementing by 1.

```
#include<stdio.h>
void main( )
{
      int digits [ ] = { 2, 4, 6, 8 };
      int *ptr;
      ptr = digits; /* pointer now pointing to digits */
      printf (" Before increment: %d", *ptr);
ptr++;
      printf (" After increment: %d", *ptr);
```

Output:

```
Before increment: 2
After increment: 4
```

Pointer to a Pointer

- ✓ A pointer is a variable that holds address of memory location instead of variable's value itself.
- ✓ Accessing a variable's value or contents via a pointer is called direct addressing.
- ✓ However a pointer being a variable can point to another pointer variable is called indirect addressing.
- ✓ Let us consider the following pointer declaration:

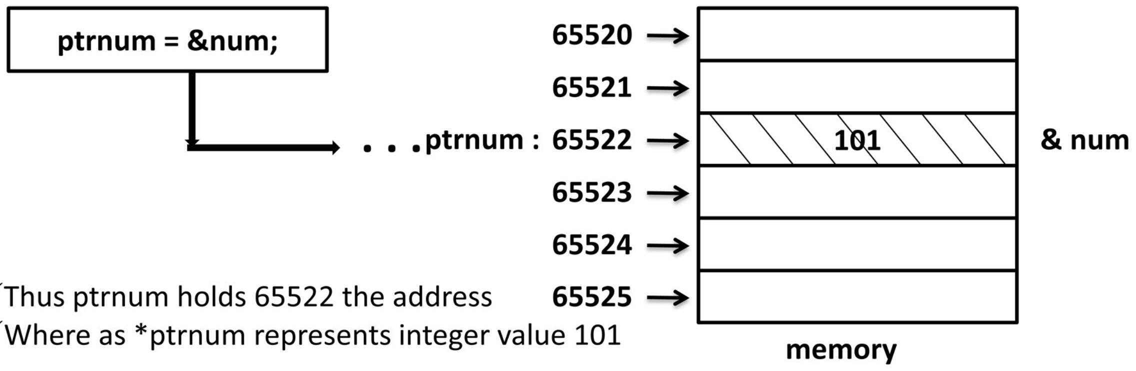
```
int  *ptrnum;
```

- ✓ Is a pointer declaration where ptrnum is pointer to an int data type value.

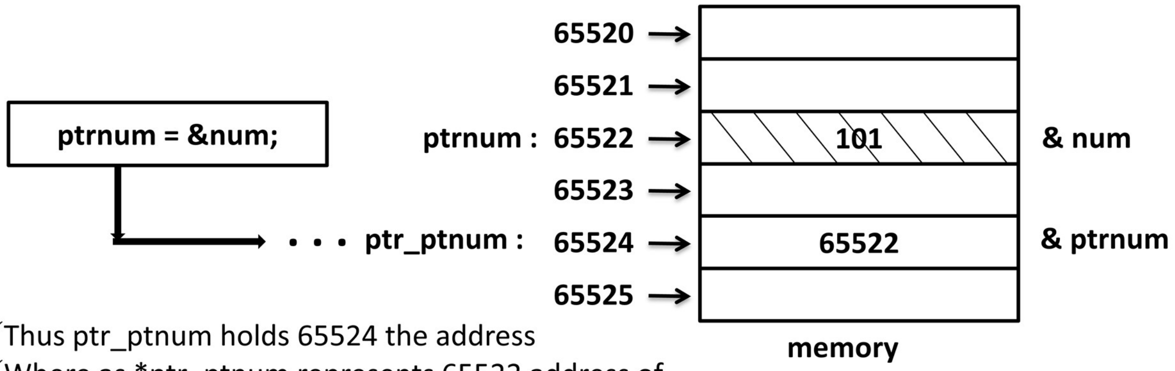
int * *ptr, ptr_ptnum;

- ✓ Is a pointer – to – pointer declaration where prefixing * *
- ✓ For ptr indicates that ptr_ptnum is an int pointer variable that can hold address of another int pointer variable say num.

- ✓ Accordingly, the corresponding address assignment also called as "Pointer initialization" and its meaning in pictorial form can be given as follows:



✓Thus ptrnum holds 65522 the address
✓Where as *ptrnum represents integer value 101



✓Thus ptr_ptnum holds 65524 the address
✓Where as *ptr_ptnum represents 65522 address of memory location of integer value 101

✓ **ptr_ptnum represents (ie points to) the value 101 by way of address of (65524) address of(65522) memory location contents is 101 i.e., indirect addressing.

```
/* Example program demonstrating the concept of pointer to pointer */
#include<stdio.h>
   void main ( )
   {
         int num;            /* simple int variable */
         int *ptrnum; /* an int pointer to num */
         int **ptr_ptnum; /* a pointer to pointer variable */
printf(" Enter an integer number :");
scanf ("%d", &num);
        ptrnum = &num; /* assign address of num to pointer variable ptrnum */
ptr_ptnum = &ptrnum; /* assign address of ptrnum to ptr_ptnum pointer */
printf(" value of num = %d pointed to at memory location:%u",
*ptrnum,ptrnum);
printf(" value of num = &d pointed to at %u by %u by %u a pointer to pointer
", **ptr_ptnum, *ptrnum, ptr_ptnum);
printf("i.e., indirect addressing by **ptr_ptnum");
printf(" i.e, address of address of the memory location holds the data");
getch( );
}
```

Output:

```
Enter an integer number: 101
value of num = 101 pointed to at memory location: 65522
value of num = 101 pointed to at 65522 by 65524 a pointer to pointer
i.e., indirect addressing by **ptr_ptnum
i.e, address of address of the memory location holds the data
```

Assignment Questions

1. Define function. Explain the terms function declaration/prototype, function definition & function call with related example
2. Explain with example parameter passing techniques available in ‘C’
3. What is recursion? Write a ‘C’ program to generate the Fibonacci series using recursion
4. Write a C program to find factorial of a number using recursive function
5. What is a pointer? Explain how pointer variables are declared and initialized
6. Explain pointer arithmetic with example program
7. Explain how arguments are passed as a parameter to functions using pointers

***********************End of Module-3************************