Chapter 8

Functions

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Introduction
- Working with Function
- The function main()
- Parameter Passing
- Functions Returning Multiple Values
- Functions Returning Pointers
- Pointers to Functions
- Nesting of Functions
- Recursion
- Passing Arrays to Functions
- Assignments

Introduction

- **A 'C' program is a collection of functions.**

  Knowingly or unknowingly, we have used many functions (at least one) in *every* 'C' program that we have done so far. e.g., `main()`, `printf()`, `scanf()` etc.

  In this chapter, we will explore the various aspects of functions in detail.

- **What is a Function?:**

  - **A function is a self contained** (independent) **block of statements that perform a coherent** (same type) **task of some kind.**

- **Types of Functions:**

  1. **Library / Pre-Defined Functions: Functions defined by the C compiler.**

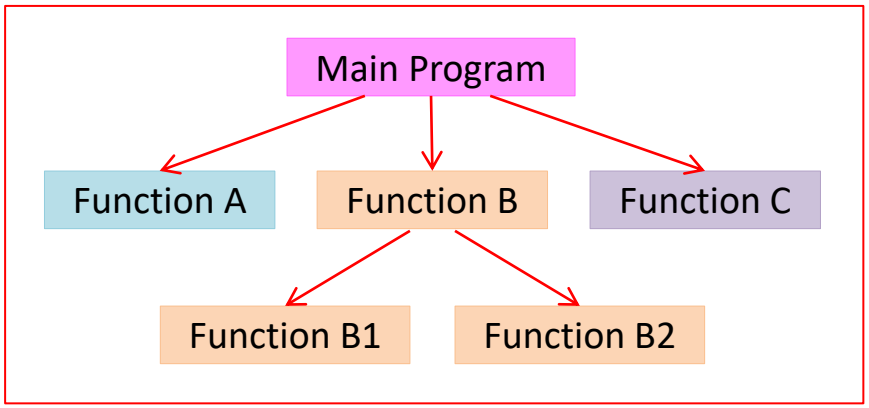
     For example: `printf()`, `scanf()` etc.

  2. **User-Defined Functions: Functions defined by the user.**

     [NOTE]: In this chapter, we will primarily focus on user-defined functions.

- **Why Use Functions?:** Using functions has the following advantages:

1. It facilitates top-down programming approach i.e., if a program is divided into functions, then each function may be coded independently and later can be combined together to fix the main problem.



2. Using functions makes it easier to understand, debug, and test the program. Hence, it reduces the programming complexity.
3. By using functions, we can avoid repetition of writing codes and hence can reduce the length of source code.
4. Functions incorporates code reusability.
5. Functions facilitates multi file programming.

Working with Functions

- **Let us understand how to work with functions** (user-defined functions)**, with the help of a simple program that finds the average of 3 integers. In order to mark the difference, the same program is written using two approaches:**
  1. **Without using a user-defined function** (This is the approach that we have followed so far).
  2. **By using a user-defined function named, "Average".**
- **Approach 1 (Without using a user-defined function):**

```
# include <stdio.h>

void main()
{
    int a, b, c;
    float avg;
    printf("\nEnter three integers: ");
    scanf("%d %d %d", &a, &b, &c);
    avg = (float)(a+b+c)/3;
    printf("\nThe average is: %f", avg);
}
```

Output

```
Enter three integers: 12 15 2
The average is: 9.66
```

Approach 2 (By using a user-defined function named, "Average"):

```
/* PR8_1.c: Program to calculate the average of three numbers by
using a user-defined function */

# include <stdio.h>

/* Function Definition */
float Average(int x, int y, int z)
{
     float avg;
     avg = (float)(x+y+z)/3;
     return avg;
}

/* Function Declaration (or, Function Prototype) */
float Average(int x, int y, int z);

void main()
{
    int a, b, c;
    float avg;
    printf("\nEnter three integers: ");
    scanf("%d %d %d", &a, &b, &c);
    avg = Average(a, b, c); /* Function Call */
    printf("\nThe average is: %.2f", avg);
}
```

Output

```
Enter three integers: 12 15 2
The average is: 9.66
```

Here,

- `main()` is known as the *calling function.*
- `Average()` is known as the *called function.*

- **It can be noticed that working with functions involves three activities (in order):**
  1. **Defining the function**
  2. **Declaring the function**
  3. **Calling / Invoking the function**

  We will discuss them one-by-one in detail next.

[NOTE]: The above activities i.e., defining a function, declaring a function and calling a function is not limited to the user-defined functions only. NO function (whether user-defined or library) can be called / used without being defined and declared.

- For user-defined functions, we are doing these tasks exclusively (in that order).
- So far, the library functions are concerned, they are pre-defined by the C compiler. There *definition part* is stored in the *library files* (in binary format) and their *declarations* (along with in which library file they are actually located) are stored in the *header files*.

  When we intend to use / call any of these library functions in our program, we have to exclusively include their declarations in our program through the pre-processor directive "`#include`".

Defining the Function (The Function Definition)

- In the function definition, we specify, through a set of statements, what the function takes as input, what it does, and what it returns as output.
- **Syntax:** The function definition takes the following general form:

Function Header

Function Body

```
return_type function_name(type arg1, type arg2, ...)
{
  statement(s);
}
```

e.g.,

Function Header

Function Body

```
float Average(int x, int y, int z)
{
    float avg;
    avg = (float)(x+y+z)/3;
    return avg;
}
```

Description of Individual Components:

1. Return Type:

- A function may return a value. The `return_type` is the data type of the value the function returns (to the calling function).
- If a function doesn't return anything, then its return type should be explicitly specified is `void`.
- Mentioning the return type is optional. If no return type written, then C assumes that it is `int` (i.e., the default return type in C is `int`).

2. Function Name:

- It is a valid 'C' identifier.
- The function name should be always followed by a pair of parentheses.
- Both the function name as well as the pair of parentheses (in that order) are compulsory.

3. Argument / Parameter List (Within the Parenthesis):

- **They specify the number, order, and type of inputs accepted by the function.**
- **Each argument must be preceded by its data type.**
- **If there are more than one augments, they should be separated by commas.**
- **Grouping of arguments of same data type is not permitted. i.e.,** `float Average (int x, int y, int z)` **can't be substituted by** `float Average (int x, y, z)`.
- **The argument list is optional** (it is the case when the function doesn't take any input). **An empty argument list can also be represented by writing the key word** `void` **within the pair of parentheses.**
  e.g., `int abc()` **is same as writing** `int abc(void)`.

[NOTE]: In C, the arguments / parameters are often classified as "formal arguments" and "actual arguments".

- They arguments / parameters used in the function definition and prototypes are called formal arguments / parameters.
- The arguments used in function call are called actual arguments / parameters.

4. The Opening and Closing Curly Braces:

- **The opening curly brace "{" marks the beginning of the function body and the closing curly brace "}" marks the end of the function body.**
- **Both the braces are compulsory.**

5. The Statement(s) (Within the Curly Braces):

- **The set of statement(s) are optional i.e., a function body may not even contain any statements.**

```
void Demo()
{

}
```

[A function in its simplest form (no return values, no arguments, and no statements): This function does nothing when called. But this is a valid function in 'C'.]

- **The body of the function may contain a `return` statement.**

About the `return` statement:

- **Use / Control Flow: The `return` statement is used (inside the called function) to transfer the control immediately back to the calling function.**
- **Syntax 1:**

```
return (expression); or return expression;
```

This form of the `return` statement is used when the called function returns a value to the calling function. Here, `expression` is a valid 'C' expression whose *result* is returned to the calling function.

For Example: in PR8_1.c, slide #8.6, the called function "Average()" returns the average of three integers to the calling function "main()". So, we have written the statement "`return avg;`". We can even write this statement as "`return (float)(a+b+c)/3`".

- **Syntax 2:**

```
return;
```

This form of the `return` statement is used, when the called function DOESN'T return any value to the calling function, but we explicitly want that the control should be transferred to the calling function.

So, writing "`return;`" just before the closing curly brace is same as omitting it.

For Example:

```
/* We want that this function shouldn't return  any value. It should only print the sum of 'a' and 'b'.  */
void Sum(int a, int b)
{
    printf("The sum is: %d", a+b);
    return;  /* Optional */
}
```

- **In the absence of a `return` statement, the closing curly brace acts as "`return;`".**
- **A `return` statement returns exactly one value. e.g., the following return statement is illegal.**

```
return(a, b); /* illegal */
```

- **The type of data returned by the return statement must be *same* as the return type of the function.**

```
void Test(...)
{
    ...
    return; /* OK*/
}
```

```
Test(...)
{
    ...
    return 0; /* OK*/
}
```

```
Test(...)
{
    ...
    return (5+3); /* OK*/
}
```

```
float Test(...)
{
    ...
    return 5; /* illegal*/
}
```

- **A function may have more than one `return` statements, but only one `return` statement should be made to execute (by writing suitable conditional statements). e.g.,**

```
if(x==0)
   return 0;
else
   return 1;
```

- **[Note - The Placement of the Function Definition]: A function can be defined either before or after the calling function. Even a function can be defined in a different file** (in that case we have to link this file to the current file in which the calling function is present). **But a function can't be defined within another function.**

```
void main()
{
   float Average(int x, int y, int z) /* Error */
   {
      ...
   }
}
```

Declaring the Function (The Function Declaration / Function Prototype)

- **Like variables, a function must be declared, before it is invoked / called.**
- **Syntax:**

```
return_type function_name (type arg1, type arg2, ...);
```

e.g.,

```
float Average(int x, int y, int z);
```

Notice that, the function declaration / prototype is noting but the function header followed by a semicolon (;).

- **The function declaration / prototype gives necessary information regarding the called function to the calling function(s). The following information are given**
  - **A function named `function_name` exists.**
  - **The number, order, and type of parameters accepted by the function.**
  - **The type of data returned by the function.**

- **Notes:**

1. **It is mandatory to put the function declaration before its call.** (Recall that the function definition can be placed anywhere, either before or after the calling function).

2. **Like local and global variable declarations, a function declaration can be made either inside the calling function (locally) or outside the calling function (globally).** (Recall that putting a function definition within another is not permitted).

   However, it is advisable to declare a function outside all functions (globally), because in that case it is *visible* to all calling functions.

```
void main()
{
   ...
   float Average(int a, int b, int c); /* OK */
}
```

2. The number, order, and type of arguments in the function prototype should *exactly match* with that of the function definition. If there is a mismatch the compiler will produce an error.

3. Writing the parameter *names* inside the prototype is optional. e.g.,

```
float Average(int a, int b, int c);
```

is same as

```
float Average(int, int, int);
```

4. The prototype itself is optional where the function definition precedes the function call. **For Example:** in PR8_1.c, slide #8.6, it is perfectly OK if we omit the function declaration.

However, it is advisable to write the function prototype always.

Calling the Function (The Function Call)

- **Syntax:**

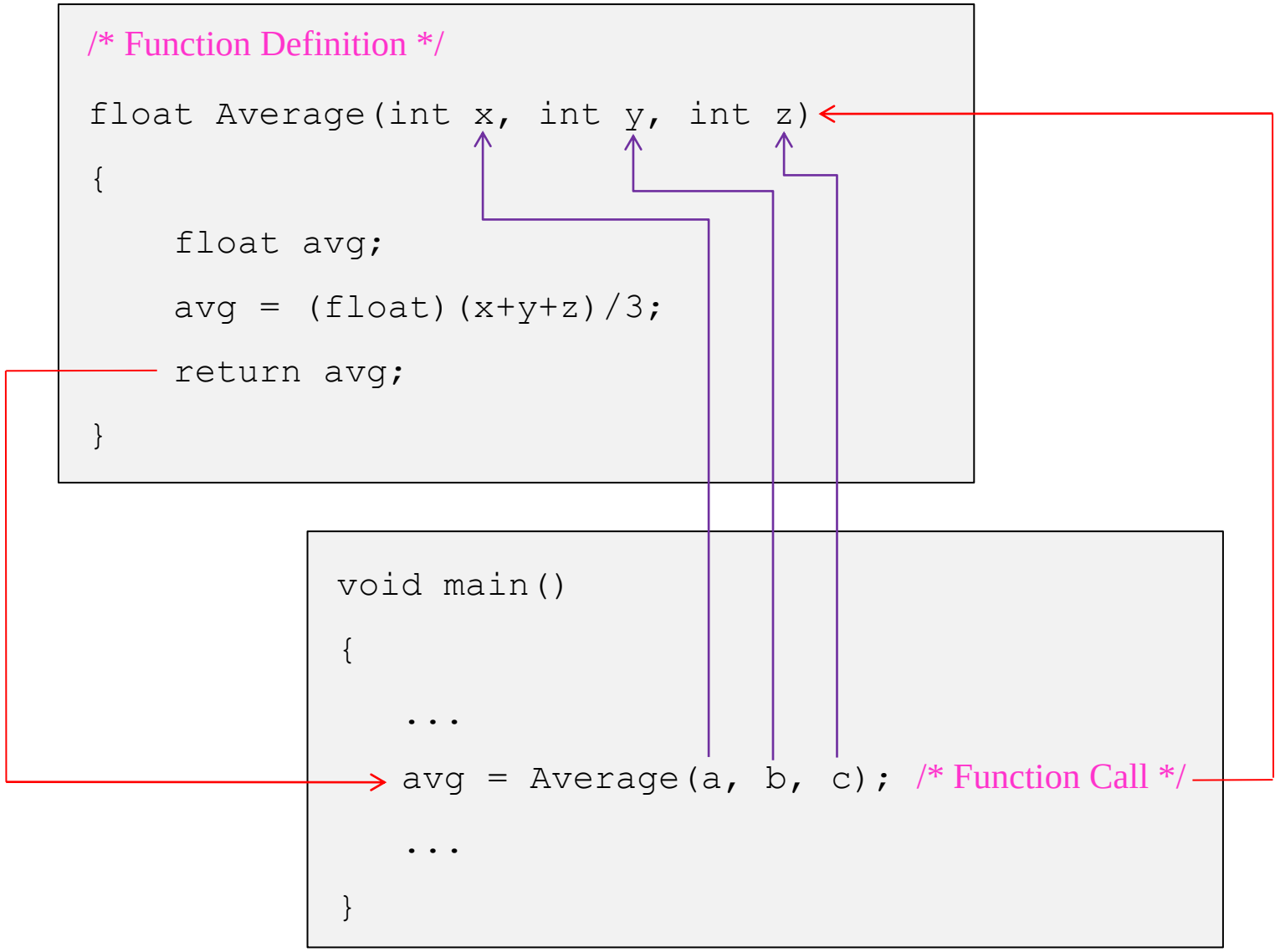
```
function_name (arg1, arg2, ...);
```

e.g.,

```
Average(a, b, c);
```

- **The Control Flow: What Happens When a Function is Called?**
  - When the compiler encounters a function call, first the values of the actual arguments (if any) are *copied* to the corresponding formal arguments. The control is then transferred to the called function.
  - The called function is then executed line by line until a `return` statement is encountered or the closing curly brace is reached.
  - At this point the control is transferred back to the calling function, to the exact place where the function call was made.

[Refer to the figure, next slide]


/* Function Definition */
float Average(int x, int y, int z)
{
float avg;
avg = (float)(x+y+z)/3;
return avg;
}
void main()
{
...
avg = Average(a, b, c); /* Function Call */
...
}

- **Notes:**

1. **The number, order, and type of the actual arguments** (arguments in the function call) **should exactly match with that of the formal arguments** (the arguments in the function definition and prototype), **otherwise a compilation error will occur.**

2. **It doesn't matter whether we use a different set of *names* or the same set of *names* for the actual and formal arguments.**

   i.e., In the program shown in the previous slide we have used a different set of *names* for the actual and formal arguments: the actual arguments are "a, b, c", whereas the formal arguments are "x, y, z". However, we can choose the same set of names "a, b, c" for both actual and formal arguments. (generally, same names are taken).

3. **No matter, whether we use the same set of names or a different set of names for the actual and formal arguments, they are actually a different set of variables.**
    - **The actual arguments "`a, b, c`" are local to the calling function (the called function is unaware of them).**
    - **The formal arguments "`a, b, c`" are local to the called function (the calling function is unaware of them)**

Some Programming Examples

- **Programming Example 1:**

```
/* PR8_2.c: A program using function to find the largest among three integers*/

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

/* Function Definition */
int Largest(int a, int b, int c)
{
   int largest;

   if(a>b && a>c)
       largest = a;
   else if(b>a && b>c)
       largest = b;
   else
       largest = c;

   return largest;
}
```

[Cont.]

```
/* Function Declaration (or, Function Prototype).
We can omit it, since function definition precedes function call. */
int Largest(int a, int b, int c);

void main()
{
    int a, b, c, largest;
    printf("Enter three integers: ");
    scanf("%d %d %d", &a, &b, &c);

    largest = Largest(a, b, c); /* Function call */

    printf("\nThe largest number is: %d", largest);
    getch();
}
```

Output

```
Enter three integers: 12 15 2
The largest number is: 15
```

Programming Example 2:

```
/* PR8_3.c: A program using function to evaluate the equation y = x^n when 'x' is any number and
'n' is a non-negative integer */

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

/* Function Prototype.
Here we CAN'T omit this prototype, because function definition is placed after the function call */
float Equate(float, int);

void main()
{
    float x;
    int n;
    printf("Enter the values of x and n: ");
    scanf("%f %d", &x, &n);

    printf("\nx = %f; n = %d", x, n);
    printf("\nx to power n = %f\n", Equate(x, n)); /* Function call */

}
```

[Cont.]

```
/* Function Definition */
float Equate(float x, int n)
{
    int i;
    float result = 1.0;

    for(i=1; i<=n; i++)
    {
        result = result*x;
    }

    return result;
}
```

Output

```
Enter the values of x and n: 12 5
x = 12.000000; n = 5;
x to the power n = 248832.000000
```

The Function `main()`

- **We have used the function `main()` in *every* 'C' program that we have done so far, without knowing much about it. In this section, we will explore it in detail.**
- `main()` **is a special user-defined function. We have used the term "special" because in `main()` the function header is pre-defined whereas the function body is user-defined.**

```
void main()  /* Pre-defined */
{
   statement(s);  /* User-defined */
}
```

- **Importance of `main()` : Program execution always starts from `main()`.** (A 'C' program is a collection of functions. So, which function should execute 1st when the program runs? Program execution always starts from `main`). **Other functions (library or user-defined) are called by `main` as and when needed.**

- **Every C program must have exactly one `main` function** (if there are more than one `main`, the compiler can't understand which one marks the beginning of the program. So, it generate an error message.).
- **So far, we have used main in just one form "`void main()`". However, C permits following forms of `main`:**

```
main()
int main()
main(void)
int main(void)
```

All are same
(Takes no arguments, returns an integer value *to the compiler*).

```
void main()
void main(void)
```

Both are same
(Takes no arguments, returns nothing *to the compiler*)

```
int main(int argc, char* argr[])
void main(int argc, char* argr[])
```

Used for accepting *command line arguments* (discussed in chapter "Files and Command Line Argument.)

Programming Example:

```
/* PR8_4.c: A program to compute the GCD of two numbers*/

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

/* Function Definition */
int FindGCD(int a, int b)
{
    int i, larger;

    if(a>b)
        larger = a;
    else
        larger = b;

    for(i=larger;i>=1;i--)
    {
        if(a%i==0 && b%i==0)
        {
            return i;
        }
    }
}
```

[Cont.]

```
int main(void)
{
    int a, b;
    printf("\nEnter two integers: ");
    scanf("%d %d", &a, &b);
    printf("\nThe GCD of %d and %d is: %d", a, b, FindGCD(a, b));
    getch();

    return (0); /* If we don't write this there will be a compilation error, because here
                   the return type of main is int.*/
}
```

Output

```
Enter two integers: 16 24
The GCD of 16 and 24 is: 8
```

Parameter Passing

- The technique used to pass data from one function to another (from the calling function to the called function) is known as parameter passing.
- It can be done in two ways:
  - Call by value
  - Call by reference / Call by pointer

Call By Value

- This is the technique that we have used so far to pass the arguments from a calling function to a *user-defined* called function.

- **The Technique:** In call by value, the values of the actual arguments (arguments in the function call) are copied to the formal arguments (arguments in the function definition).
  - Therefore, the called function works on the copy and not on the original values of the actual arguments.
  - This ensures that the original data in the calling function can't be changed by the called function.

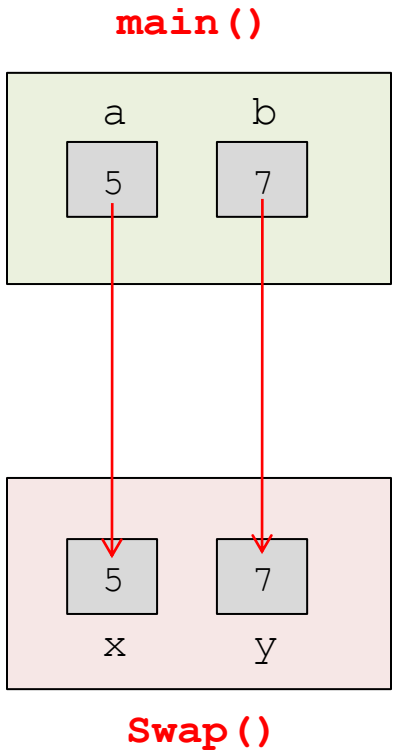
A program given in the next slide will help understand the concept.

Programming Example:

```
/* PR8_5.c: Swapping of two integers by using call by value */

# include <stdio.h>

/* Function Prototype */
void Swap(int, int);

void main()
{
     int a=5, b=7;
     printf("\nBefore swapping: a=%d, b=%d", a, b);
     Swap(a, b);  /* Function call - By value */
     printf("\nAfter swapping: a=%d, b=%d", a, b);
     getch();
}

/* Function Definition */
void Swap(int x, int y)
{
     int t;
     t = x;
     x = y;
     y = t;
}
```

Output

```
Before swapping: a=5, b=7
After swapping: a=5, b=7
```



Call By Reference / Call By Pointer

- **The Technique:** In call by reference, the memory addresses of the variables rather than the copies of the values of the variables are sent to the formal arguments.
  - Therefore, the called function directs works on the original data in the calling function.
  - This ensures that any changes made to the values of the variables by the called function reflects in the calling function.

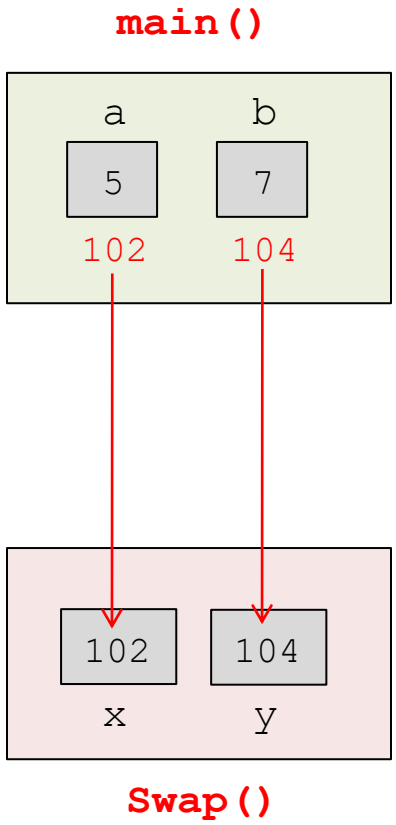
Let us rewrite the same *swapping* program by using call by reference and see what happens.

Programming Example:

```
/* PR8_6.c: Swapping of two integers by using call by value */

# include <stdio.h>

/* Function Prototype */
void Swap(int*, int*);

void main()
{
     int a=5, b=7;
     printf("\nBefore swapping: a=%d, b=%d", a, b);
     Swap(&a, &b); /* Function call - By reference */
     printf("\nAfter swapping: a=%d, b=%d", a, b);
     getch();
}

/* Function Definition */
void Swap(int *x, int *y)
{
     int t;
     t = *x;
     *x = *y;
     *y = t;
}
```

Output

```
Before swapping: a=5, b=7
After swapping: a=7, b=5
```



[NOTE]: When to use call by value and call by reference?

- **Call by value is OK if we want that the called function SHOULD NOT directly manipulate the variables of the calling function.**

  In other words, the called function should work as follows:

  Take some values → Process those values → Return a value

- **Call by reference is used when we exclusively want that the called function should directly manipulate the variables of the calling function.**

  So, it is openly used to manipulate arrays, strings, or when we want to return multiple values from the calling function.

Functions Returning Multiple Values

- In the section "Working with Functions" we have learnt that a function can return just one value using a return statement.
- However, what happens if we want that the called function should return more than one values to the calling function.
- This situation could be handled by using the call by reference approach, in which we can make the called function to *indirectly* return multiple values.

  For Example: The Swap() function in the last program 'PR8_6.c' (call by reference approach) indirectly returns two values to the calling function.

One More Programming Example:

```
/* PR8_7.c: Program using function to calculate the sum and difference of two integers*/

# include <stdio.h>

/* Function definition (this function indirectly returns two values: sum, diff) */
void SumAndDifference(int x, int y, int *sum, int *diff)
{
     *sum = x+y;
     *diff = x-y;
}

void main()
{
     int a, b, s, d;
     printf("\nEnter two integers: ");
     scanf("%d %d", &a, &b);
     SumAndDifference(a, b, &s, &d); /* Function call - By value & reference */
     printf("\nSum=%d, Difference=%d", s, d);
     getch();
}
```

Output

```
Enter two integers: 5 9
Sum=14, Difference=-4
```

Functions Returning Pointers

- **Since pointers are also data types in C, we can make a function to return a pointer.**
- **Programming Example:**

```
/* PR8_8.c: Program to calculate the larger of two integers */
# include <conio.h>
/* Prototype (Function that returns an integer pointer) */
int* FindLarger(int*, int*);

void main()
{
    int a, b, *p;
    printf("\nEnter two integers: ");
    scanf("%d %d", &a, &b);
    p = FindLarger(&a, &b); /* Return value is collected in a integer pointer */
    printf("\nThe larger is: %d", *p);
}
```

[Cont.]

```
/* Function definition */
int* FindLarger(int *x, int *y)
{
    if(*x>*y)
        return(x); /* Address of 'a' */
    else
        return(y); /* Address of 'b' */
    getch();
}
```

Output

```
Enter two integers: 55 11
The larger is: 55
```

Pointers to Functions

- **Like a variable, a function also occupies a space in memory and has a memory address. Therefore, it is possible to declare a pointer to a function.**

  [NOTE]: This is the reason why we have defined a pointer as "A variable that can store the address of another variable or *function*".

- **Declaration:**
  - **Syntax:**

    ```
    return_type (*pointer_name)();
    ```

    e.g.,

    ```
    int(*p)();
    ```

    The above declaration tells the compiler that, `'p'` is a pointer that can point to a function whose return type is `int`.
  - **NOTE: The parenthesis around `'p'` is necessary, because a declaration like "`int *p();`" would tell the compiler that `'p'` is a function returning an integer pointer.**

- **Initialization: Demonstrated through the following example**

```
double Ratio(int, int); /* Function prototype */
double (*fp)(); /* Pointer declaration */
fp = Ratio; /* Initialization (This initialization is possible because, the function name
               itself corresponds to the address of the function ) */
```

- **Function Call (through pointer): We can call the function "`Ratio`" through the function pointer "`fp`" as follows:**

`(*fp)(a, b);` **This is equivalent to:** `Ratio(a, b);`

- **A NOTE: Though it is possible to declare a pointer to a function and call function through its pointer, it is practically never used. Because calling a function by its name (like `Ratio(a, b);`) is more *understandable* than calling a function by its pointer (like `(*fp)(a, b);`)**

Nesting of Functions

- **C permits nesting of functions. i.e., `main()` calls `function1`, which calls `function2`, which calls `function3`, ... and so on.**
- **Programming Example:**

```
/* PR8_9.c: A program that takes four integers 'a', 'b', 'c', and 'd' as input and displays the value of
(a+b)/(c-d) if (c-d) ≠0, otherwise it will display "(c-d) = 0, The result is undetermined." */

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

/* Function Definition for "Difference" */
int Difference(int x, int y)
{
    if(x==y)
        return 0;
    else
        return 1;
}
```

[Cont.]

```
/* Function Definition for “Ratio” */
float Ratio(int a, int b, int c, int d)
{
    if(Difference(c, d))  /* Call to function "Difference” */
        return ((float)(a+b)/(c-d));
    else
        return (0.0);
}

void main()
{
    int a, b, c, d;
    float result;
    printf("\nEnter four integers 'a' , 'b', 'c', 'd': ");
    scanf("%d %d %d %d", &a, &b, &c, &d);
    result = Ratio(a, b, c, d);  /* Call to function “Ratio” */

    if (result)
        printf("\nThe value of (a+b)/(c-d) is: %.2f", result);
    else
        printf("\n(c-d) = 0, The result is undetermined.");

    getch();
}
```

Output:

```
Run 1:
Enter four integers 'a' , 'b', 'c', 'd': 12 3 8 3
The value of (a+b)/(c-d) is: 3.00

Run 2:
Enter four integers 'a' , 'b', 'c', 'd': 12 3 8 8
(c-d) = 0, The result is undetermined.
```

- **Notes:**

1. **In principle, there is no limit how deeply functions can be nested.**
2. **Function calls can also be nested.**

   For example: a statement like the following is perfectly legal.

```
p = mul(mul(5, 2), 6)
```

Recursion

- **[Definition]: Recursion is a special case where a function calls itself.**

e.g.,

```
void main()
{
   printf("\nHi");
   main();
}
```

This program prints "Hi" indefinitely.

- **[NOTE]: When we write recursive functions, we must have a conditional statement somewhere in the function that force the function to return without the recursive call being executed. Other wise the function will never return, and the program will execute indefinitely** (like the above example).

Programming Example 1:

```
/* PR8_10.c: A program to print 1 to 10 without using any loop structures */

#include <stdio.h>

int i = 0;  /* Global variable declaration */

void main()
{
    printf("%d ", ++i);
    if(i==10)
        exit(0);
    main();
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

Programming Example 2:

```
/* PR8_11.c: A program to compute the factorial of a number*/

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

int Factorial(int n)
{
    if(n==1)
        return (1);
    else
        return(n*Factorial(n-1));
}

void main()
{
    int n;
    printf("\nEnter a non-negative integer: ");
    scanf("%d", &n);
    printf("\nThe factorial of %ld is: %d", n, Factorial(n));
    getch();
}
```

Output

```
Enter a non-negative integer: 5
The factorial of 5 is: 120
```

Passing Arrays to Functions

Passing One-Dimensional Arrays to Functions

- **Explained through the following example**

```
void main()
{
    int a[3] = {10, 7, 15};
    int n = 3;
    Sort(a, n); /* Function Call */
}
```

- To pass a 1D array to a called function, it is sufficient to pass the array name 'a' and its size 'n' as the actual arguments.
- The variable 'n' must be assigned to the size of 'a' prior to the function call (as we have done through `int n=3;`), otherwise a compilation error will be there.
- At the place of 'n' we can directly write down '3' (i.e., `Sort(a, 3)`).
- Passing 'n' is also optional if we can know the size of the array explicitly in the called function.

```
/* Function definition */
void Sort(int b[], int size)
{
   ...
}
```

- Here, the variable `size` collects a *copy* of the array size 'n'.
- The array 'b' refers to the same array 'a' (not a copy of 'a').
  - It is because, in the function call we have passed the array 'a' by its name (i.e., Sort (a, n)).
  - As we have learnt in the chapter "Pointers", an array name actually refers to the base address of the array. Therefore, by passing the array name, we are in fact, passing the address of the array to the called function (call by reference).

  Hence, any changes made in the array 'b' in the called function will be reflected in the original array 'a'.
- Since 'b' and 'a' refers to the same array in the memory, generally the array in the called function is given the same name as the array in the calling function (i.e., the function header could be `void Sort(int a[], int size)`).
- [NOTE]: We can't pass a *whole* array by value as we did for ordinary variables.

Programming Example

```
/* PR8_12.c: A program to sort an array of 5 integers */

#include <stdio.h>

/* Function that swaps two integers */
void Swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}

/* Function that sorts an array of integers in ascending order */
void SortArray(int a[], int n)
{
     int i, j;
     for(i=0;i<n-1;i++)
     {
          for(j=i+1;j<n;j++)
          {
               if(a[i]>a[j])
                    Swap(&a[i], &a[j]);
          }
     }
}
```

[Cont.]

```
void main()
{
    int i, arr[5];
    printf("\nEnter 5 integers to sort: ");
    for(i=0;i<5;i++)
    {
        scanf("%d", &arr[i]);
    }

    SortArray(arr, 5); /* Function call */

    printf("\nThe integers after sorting: ");
    for(i=0;i<5;i++)
    {
        printf("%d ", arr[i]);
    }
    getch();
}
```

Output

```
Enter 5 integers to sort: 55 21 7 0 11
The integers after sorting: 0 7 11 21 55
```

Passing Two-Dimensional Arrays to Functions

- **Explained through the following example**

```
void main()
{
    int a[3][2] = {1, 2, 3, 4, 5, 6};
    int m=3, n=2;
    Average(a, m, n);   /* Function Call */
}
```

- To pass a 2D array to a called function, it is sufficient to pass the array name 'a', its row size 'm', and its column size 'n' as the actual arguments.
- The variables 'm' and 'n' must be assigned to the row size and column size of 'a' respectively prior to the function call (as we have done through `int m=3, n=2;`), otherwise a compilation error will be there.
- At the place of 'm' and 'n' we can directly write down '3' and '2' respectively (i.e., `Average(a, 3, 2)`).
- Passing 'm' and 'n' are also optional if we can know the row and column sizes of the array explicitly in the called function.

```
/* Function definition */
void Average(int b[][COL_SIZE], int p, int q)
{
   ...
}
```

- Here, the variables 'p' and 'q' collects the *copies* of the row size 'm' and column size 'n' respectively .
- `COL_SIZE` is a constant (not a variable) which is equal to the value of 'n' (or 'q'). `COL_SIZE` can be directly mentioned here (i.e., `void Average(int b[][2], int p, int q)`), or can be indirectly specifies through a #define directive (i.e., `#define COL_SIZE 2`, then `void Average(int b[][COL_SIZE], int p, int q)`).
- The array 'b' refers to the same array 'a' (not a copy of 'a'). Therefore, any changes made in the array 'b' in the called function will be reflected in the original array 'a'.
- Since 'b' and 'a' refers to the same array in the memory, generally the array in the called function is given the same name as the array in the calling function (i.e., the function header could be `void Average(int a[][2], int p, int q)`).
- [NOTE]: We can't pass a *whole* array by value as we did for ordinary variables.

Programming Example

```
/* PR8_13.c: A program to find the average of all integers in a matrix. */
#include <stdio.h>
#define COL_SIZE 2
/* Function that finds the average of all integers in a matrix */
float FindAverage(int a[][COL_SIZE], int m, int n)
{
    int i, j, sum = 0;
    float avg;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            sum = sum+a[i][j];
        }
    }
    avg = (float)sum/(m*n);
    return avg;
}
```

[Cont.]

```
void main()
{
    int i, j, a[3][2];
    float avg;
    printf("\nEnter the elements of a 3X2 matrix: ");
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    avg = FindAverage(a, 3, 2); /* Function call */
    printf("\nThe average is: %.3f", avg);
    getch();
}
```

Output

```
Enter the elements of a 3X2 matrix: 4 22 8 6 77 11
The average is: 21.333
```

Passing Strings to Functions

- **A string is a 1D array of characters terminated by a NULL character. Therefore, the rules for passing strings to functions is similar to those for passing 1D arrays to functions.**
- **The following example demonstrates:**

```
void main()
{
    char name[] = "San";
    Display(name); /* Function Call  (we only pass the string name, NOT its length,
                      because it can be easily found out in the called function by
                      using the strlen() function) */
}
```

```
/* Function definition */
void Display(char name[])
{
   ...
}
```

Assignments

Complete the experiments given in "Lab Manual - Section 10"**.**

End of Chapter 8