Chapter 4: C Functions

In this chapter you will learn all about:

- Use of Functions
- Defining and declaring functions & Function prototypes
- Function calling and returning – Working mechanism from CPU perspective – Making sense of total processing time
- Passing parameters by value and reference
- Writing Recursive Functions
- C Pre-processor: Writing Macros
- Storage Class - Extern
- **Annexure**

==========

.

A. Use of Functions

- Modularity in the design process
- Stand-alone code for designing an algorithm/strategy
- Ease of working in teams, especially on large-scale workflow processes
- Ease of debugging/error locating and portability

B. Function declaration & Definitions

General form of a Function Definition

Return Data-type **func_name** (type1 arg1, type2 arg 2, …, type n arg n)

```
{
...<your code here>…
return; /* return statement depending on the type declared */
}
```

Default type (if not explicitly declared) – int

Remarks: It is a good programming practice to have the **return** statement even though some functions do not return anything.

Function Prototypes: This is an alert to the **compiler** that the function being accessed will be defined later in the program; This is essential as otherwise it will be confusing for the compiler when function access precedes the function definition.

Return Data-type **func_name** (type1 arg1, type2 arg 2, …, type n arg n)**;**

/* observe the semi-colon at the end */

The arguments/parameters in this prototype are called as "**dummy variables**" or "local variables". You need not use the names of the variables you are actually passing to the functions (rather, should not!) from the caller program. This makes sense, as we may need to pass different variables at different instances in the code.

The compiler would establish a correspondence (map) between the variables passed to the function and the dummy variables. This mapping means both variables would mean accessing to the same data item.

Note: It is a good practice to put all the function prototypes **after** the header files systematically (before the main() starts).

Example 4.1: Observe this example and declarations.

```
# include<stdio.h>
void myadder(int a, int b); /* Function prototype */
/* = = == == == */
int main(){   /* Note – our main is also a function!! */
   int x,y;
   x =10;
   y = 20;
   myadder(x,y);  /* calling the function myadder */
   printf("Bye Bye from main function");
   return(0);
}
/* ====Actual func definition below ===== */
void myadder(int a, int b)
{
   int sum;
   sum = a+b;
   printf("Result of addition (from Func): %d\n",sum);
   return;
}
```

Example 4.2:

```
void name (int x, int y)  {
int z;
z = (x >= y) ? x : y;
printf("Max value is: %d", z);
return;
}
```

Example 4.3:

```
# include<stdio.h>
# include<math.h>
# define SEED 12345

int compute(int x, float z); /* function prototype */
void play(void); /* function prototype */

void main() {
. . . .
return;
}
/*------------ */
void play (void) /* function definition */
{
. . . . . .
return ;
}
/*------------ */
int compute(int x, float z) /* function definition */
{
.......
return(m+n); /*returns the sum m+n */
}
```

B. Working mechanism – Calling and returning modes

A function call can happen anywhere in the main code. It is important to understand the sequence of events happen when a function call is made.

Example 4.4: Function call-return mechanism

```
main() {
6       . . .
7       . . .
8       printf (…);
9       CALL FuncA(); / * call to function A happens here  */
10      Y = x*q/r;  /* next statement to execute after the CALL */
        . . .
20      p = r +t;
21      CALL FuncA(); / * call to function A happens here  */
22      gets();  /* next statement to execute after the CALL */
        . . .
}
```

What exactly happens during CALL & RET events? – A high-level description can be found in the Annexure on Page 14. We will describe this at the end of this chapter. *Stay tuned!*

B.1 Passing Parameters to a function by value

Tutorial 4.1 – *Passing by value*

- *Pass a variable with a value to a function; modify it in the function;*
  - *(i) Do not return the value but check the updated value from the called program;*
  - (ii) *Return the value and check the value from the called program; What do you observe in both cases?*

    Try running the code.
- *What is the main disadvantage of this method?*
- *When can use this method of passing by value?*

B.2 How to pass arrays to functions?

To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call.

When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets; The size of the array is not specified within the formal argument declaration; See the example below.

Example 4.5: Observe the call and return statements.

```
float average (int a, float mylist[]); /* function prototype */
void main() {
int n;
float My_avg, numbers[100];
. . . . . .
My_avg = average(n, numbers);
/* actual array name should appear here; */
. . . . . (rest of the code here) ….. }

float average(int a, float mylist[]) { /* function definition */
 float x;  . . . <code here> . . . return(x);  }
```

Some remarks – Integer "a" and "mylist" are formal arguments; The function declarations establish the fact that "a" is an integer and "mylist" is a one-dimensional floating point array. Thus there is a clear correspondence between the actual argument n and actual array numbers;

B.3 Passing by Reference – Array is a special case!

Tutorial 4.2: Do the following and see – Define an integer array and initialize it; Print the values of the array from the main(); Then pass the array to a function and modify it in the function, i.e., modify the array values; Print them within the function; Exit from the function; Now from the main() print the array; ***What do we observe?***

The result is that the array values are modified; Thus, the passing of array values seems to get modified when we return from the function!!

This is clearly a different behaviour when compared to our earlier example (Tutorial 4.1). *Why is this behaviour?*

Concept - The key to this behaviour is due to the fact that the array values are passed by ***reference*** as opposed simply by ***values***.

Array is a special case! When an array is passed to a function the array name is interpreted as the address of the first array element (that is, the address of the memory location containing the first element in the array). This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer ***to the*** first element of the array element. Thus, the locations where the values are stored are directly accessed rather than their copies! So any change to the value is made ***directly in the location where the values are stored***. So we see a change in the values of the array elements even after the function exists.

Q: What about Two-dimensional arrays?

C. Passing Two-Dimensional Arrays

When 2D arrays need to be passed it is the number of columns that need to be specified explicitly. **Thus**:

Example 4.6: *Test this code and see !*

```
void myprint2(int array[][COLS])
{
 int i, j;

 for (i=0; i<ROWS; i++)
 {
  for (j=0; j<COLS; j++)
  {
   printf("%d ",array[i][j]);
  }
  printf("\n");
 }
 return;
}
/* ========================= */
/* Declare here the required header stuff */
int main()
{
 int x[ROWS][COLS] = {{0,1,2},{3,4,5},{6,7,8}};
 myprint2(x);  /* passing just by its name */
 return 0;
}
```

D. Writing Recursive Functions

C functions may be used recursively; Recursive code is compact; Often used in tree data structures;

What is recursion?

Recursion is the process by which a function calls itself repeatedly. To set up a recursion, we need to identify two cases – a base case and a recursive case. Identifying a base case helps to stop the problem. For instance, in the case of a factorial computation, we need to go until n=1 and we need to stop. Thus, n=1 becomes a base case of the problem. Following tutorial demonstrates the functionalities.

Tutorial 4.3 – (Recursive functions)

(a) Write a program to add n natural numbers using a recursive call.

(b) Write a program to compute factorials.

Execution of the recursive function calls does not happen immediately. They are placed on a *stack* until the condition for termination happens and then the function calls are executed in the reverse order, as they are popped off the stack! This is a *characteristic of all functions that are executed recursively.*

E. The C Pre-processor

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do the required pre-processing before the actual compilation.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.

We have seen some of these rights from the beginning: #include<…>, #define XYZ 20, etc. These are pre-processor commands.

E.1. MACROS

Sometimes while programming, we stumble upon a condition where we want to use a value or a small piece of code many times in a code. Also there is a possibility that the in future, the piece of code or value would change. Then changing the value all over the code does not make any sense. There has to be a way out through which one can make the change at one place and it would get reflected at all the places. This is where the concept of a macro fits in.

A Macro is typically an abbreviated name given to a piece of code or a value. It is "like" a function, but not exactly!

#define MACRO_NAME(arg1,arg2,…,) (code here)

Example 4.7.

1. #define INC(x) x++
2. #define MULT(x,y) x*y
3. #define ADD_TEN(x) x+10

[DIY] Try the above in your programs. In problem (2) above, what do you observe if you use, MULT(5+3,2+3) in your main program? Similarly, what do you observe if you use ADD_TEN(3)*5 in your main program?

```
void main(){ /* define the macro and run */
int p=5, q=2;
q = MULT(p,q); /* q will be assigned 10 */
return;
}
```

Remarks: Multi-line macros are possible; Interested students can read further;

F. Storage Class - Extern

When variables are passed between functions their values may not be retained when they "fly" out of the defined "scope"! In such situations it is desirable to classify and organize the variables in the required form so that they are used by other functions, as needed. Storage class helps to carry out this ease of handling variables across functions.

==========

Example:

extern float root1; - *Not confined to single functions; scope extends from the point of declaration to the remainder of the program. So, they usually span 2 or more functions – They are referred to as "global variables". They retain the values assigned and hence values assigned by one function can be used in other functions;*

Example 4.8: *(communication using external arrays – global arrays)*

```
#include<stdio.h>
extern int owner[200]; /* global array - declared outside the main */
extern int items[200]; /* global array -declared outside the main*/
extern char date_found[9] = "08-06-06"; /* declared outside the main */
void find(void); /* func prototype */
/* ---------------------------------- */
void main()
{
        int i, j;
        <statements>
        owner[i] = 456; /* global array accessed here */
        return;
}
void find(void) /* function definition */
{
        int k;
        <statements>
        owner[k] = 123; /* global array accessed here */
        strcpy (date_found, "27-08-08");
        return;
}
```

Very important Note: Modification of values of the global(external) variables by the functions and retaining the modified values (permanently) is true ONLY for array elements. A demo example in the lecture will clarify that this is **NOT** true for variables (non-array elements).

Tutorial 4.4 – Assume the following flow of calls and returns initiated from your main() program. Assume each CALL and a RET consumes 5 msecs. Each function, other than calls, on an average, takes 80 msecs to execute. The main() program, excluding the CALLS, take 1200msecs. Compute the ratio of the overall time overhead due to call-return events to the total execution time.

Program flow is as follows:

main() {… F_A, F_B, F_A… }

F_A() {… F_B…F_C…}

F_B() {… F_C…F_D…}

F_C {….} F_D {. . . }

Overhead, in our example here, is the amount of time used in shuttling between functions (Call-Ret events) in the memory.

Remarks: (Multi-file compilation)

Under MS Visual Studio platform you can try the following. Instead of writing a long winding main() program with all the functions in it, you can choose to do the following which helps in a modular design.

- Keep your main file separate; In the main function declare all function prototypes with a keyword **extern** explicitly.
- Create as many separate files as you need (depending on the number of functions you have) using function names as file names with "xxxx.**c**" under source files folder.
- Build the main() function file as usual and compile; It will create the necessary executable after building (compiling, linking, etc). Then run the main() file.

Try this example:

```
/* This code is in the main file – say, main.c */
#include <stdio.h>
extern int myprint(int p); /* func prototype */
int main(){
        int k = 10, q;
        printf("Printing this msg from the main file\n");
        q = myprint(k);
        printf("Printing q = %d from the main file\n",q);
        return 0;
}

/* This is in a separate file called "myprint.c" */
int myprint(int p) {
        printf("Printing this msg from the myprint file\n");
        printf("Printing the received value from the main file: %d\n",q);
        return(p);
}
/* Note: Sometimes if you see any unusual warning, try inserting the
stdio.h header file in the myprint.c too. Note that you should include
any  other header files needed specifically for the function. */
```

ANNEXURE

(A) Bonus [DISCUSSION / DIY] problem to think over! This exercise is meant for you to practice **on your own** in order to – (i) understand how to design a modular code, (ii) what are the essential functions required? (iii) what are the parameters to pass to those functions, etc.

Consider Tutorial 3.4 on graph analysis. Rewrite using functions. Decide the functions you need to design and how the flow must go!

(B): Steps during a Function call & Return events[1]: (High-level description)

(i) As soon as a call is made from a program (say, main() program), the status of the memory, CPUs internal registers, current status of the execution (which is captured in a program status word (PSW) register in the CPU), etc, are all **pushed** to a local **stack**. Address where this main() resides in the memory and the return address of the instruction in the main program is also saved;

(ii) Control is handed over to the function to be executed;

(iii) After the function completes executing, the return statement at the last line will enable to restore the original status of the main program back (**popped**) to the respective registers from the stack; Thus, putting a **return** statement is very important;

(C) Storage Classes Types – *Automatic, External, Static*

Only external variables can be initialized and not local (automatic) variables

auto int a, b, c; - *Must be declared within the function in which they are to be used; Any variable declared within a function must be interpreted as a an auto variable. Keyword "auto" is strictly not required; An auto variable does not retain its value once*

[1] You may learn these steps in detail in your EE2028 and other Computer Architecture related courses; For now, this understanding is more than sufficient.

control is transferred out of its defining function. This means, the value assigned to an auto variable is lost once the function is exited.

extern float root1; - *Not confined to single functions; scope extends from the point of declaration to the remainder of the program. So, they usually span 2 or more functions – They are referred to as "global variables". They retain the values assigned and hence values assigned by one function can be used in other functions;*

static int count =9; - *In single-file program (all functions in one file called as "source file) the static variables behave very much as auto variables; However, they retain the values throughout the life of the program. So, when control comes back to a function the former values of static variables are retained and not lost; This allows the program to retain permanently the values throughout the execution of the program. Static variables can be utilized within the function, however, they cannot be accessed outside of their defining function;*