Introduction to Functions, prototypes and math functions

Modularizing Programs in C

- Most computer programs that solve real-world problems are much larger than the programs presented so far.
- The best way to develop and maintain a large program is to construct it into smaller pieces of programs, each of which is more manageable than the original program. When I say manageable, it means smaller programs are easy to write, easy to find problems or defects or bugs.
- This technique is called divide and conquer.
- We will talk about some of the key features of the C language that facilitate the design, implementation, operation and maintenance of large programs.

- Functions are used to modularize programs
- C programs are typically written by combining new functions you write with *prepackaged* functions available in the C standard library.
- The C standard library provides a rich collection of functions for performing common *mathematical calculations, string manipulations, character manipulations, input/output*, and many other useful operations.

- The functions printf, scanf are standard library functions.
- You can write your own functions to define tasks that may be used at many points in a program.
- These are sometimes referred to as programmer-defined functions.
- The statements defining the function are written only once, and the statements are hidden from other functions.
- Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the called function needs to perform its designated task

Math Library Functions

- Math library functions allow you to perform certain common mathematical calculations.
- you should use #include <math.h>
- Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the argument (or a comma-separated list of arguments) of the function followed by a right parenthesis.
- For example, a programmer desiring to calculate and print the square root of 900.0 you might write

```
printf("%.2f", sqrt(900.0));
```

- When this statement executes, the math library function sqrt is called to calculate the square root of the number contained in the parentheses (900.0).

- The number 900.0 is the argument of the sqrt function.
- The preceding statement would print 30.00.
- The sqrt function takes an argument of type double and returns a result of type double.
- All functions in the math library that return floating-point values return the data type double.
- Note that double values, like float values, can be output using the %f conversion specification.
- Function arguments may be constants, variables, or expressions.
- If c1 = 13.0, d = 3.0 and f = 4.0, then the statement

```
printf( "%.2f ",    sqrt (c1 + d * f) )  ;
```

- calculates and prints the square root of 13.0 + 3.0 * 4.0 = 25.0, namely 5.00.

Some examples of math functions are :

ceil (x) - rounds to smallest integer not less than x.

- ceil (7.1) returns 8.0
- ceil (-7.8) returs -7.0

fabs (x) - absolute value of x as a floating point number

- fabs (8.9) returns 8.9
- fabs (-8.9) return 8.9

exp (x) - exponential function e power x .

floor (x) - rounds to the largest number not greater than x

sin (x), cos (x) , tan (x) - Trigonometric functions

Functions in C:

- Functions allow you to modularize a program.
- All variables defined in function definitions are local variables—they can be accessed *only* in the function in which they're defined.
- Most functions have a list of parameters that provide the means for communicating information between functions.
- A function's parameters are also local variables of that function.

- There are several motivations for creating many functions in a program.
- The divide-and-conquer approach makes program development more manageable.
- Another motivation is software reusability—using existing functions as *building blocks* to create new programs. A core feature in object orient programming.
- Software reusability is a major factor in the *object-oriented programming* movement that you'll learn more about in languages derived from C, such as C++, Java and C# (pronounced "C sharp").
- We use abstraction each time we use standard library functions like printf, scanf and pow.
- A third motivation is to avoid repeating code in a program.
- Packaging code as a function allows the code to be executed from other locations in a program simply by calling the function

The code written in a function shouldn't be hundreds of lines, rather very concise and brief to perform the task. The function name should also reflect the objective of the function. For instance sin (x) , the name of the math function reflect it is computing the sin of x.

Take this simple program

```
int square ( int ) ; // prototype
```

–The int in parentheses informs the compiler that square expects to *receive* an integer value from the caller.

- the int before the name informs the compiler that square will return a value of type int

```
main ( )
{
    printf ( " The square of 10 = %d \n",  square ( 10 ) ) ;
}

int square ( int x )  // function definition matches the prototype.
{
   return x * x ;   // body of the function
}
```

Whenever you define a function like square here, you have to declare the function prior to the definition. This declaration is known as prototype. In the prototype, we declare the signature. When the compiler sees the prototype declared, it will remember that the definition will come later so it won't complain when the function is called in the main function. Prototype is essential when the function is defined later than the call or the function is defined in another file.

- The compiler refers to the function prototype to check that any calls to square contain the *correct return type*, the *correct number of arguments*, the *correct argument types*, and that the *arguments are in the correct order*.
- The format of a function definition is

```
 return-value-type function-name(parameter-list)
{
  definitions
  statements
}
```

- The *function-name* is any valid identifier.
- The *return-value-type* is the data type of the result returned to the caller.

- The *return-value-type* void indicates that a function does not return a value.
- Together, the *return-value-type, function-name* and *parameter-list* are sometimes referred to as the function header.
- The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called.
- If a function does not receive any values, *parameter-list* is void.
- A type must be listed explicitly for each parameter.

- The *definitions* and *statements* within braces form the function body, which is also referred to as a block.
- Variables can be declared in any block, and blocks can be nested

Return values;

- There are three ways to return control from a called function to the point at which a function was invoked.
- If the function does *not* return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement

    return;

- If the function *does* return a result, the statement

    return *expression;*

returns the value of *expression* to the caller

Prototype

- An important feature of C is the function prototype.
- This feature was borrowed from C++.
- The compiler uses function prototypes to validate function calls.
- Early versions of C did *not* perform this kind of checking, so it was possible to call functions improperly without the compiler detecting the errors.
- Such calls could result in fatal execution-time errors or nonfatal errors that caused subtle, difficult-to-detect problems.

- Take this function prototype for maximum

// **function prototype**

int maximum(int x, int y);

- It states that maximum takes two arguments of type int and returns a result of type int.

the actual function is written as

```
int maximum( int x, int  y )
{
   if ( x >= y ) return x ; else return y ;
}
```

- Notice that the function prototype is the same as the first line of maximum's function definition.
- A function call that does not match the function prototype is a compilation error.
- An error is also generated if the function prototype and the function definition disagree.
- For example, if the function prototype had been written

void maximum(int x, int y);

- the compiler would generate an error because the void return type in the function prototype would differ from the int return type in the function header.

Argument Coercion and "Usual Arithmetic Conversion Rules"

- Another important feature of function prototypes is the coercion of arguments, i.e., the forcing of arguments to the appropriate type.
- For example, the math library function sqrt can be called with an integer argument even though the function prototype in <math.h> specifies a double parameter, and the function will still work correctly.
- The statement

printf(**"%.3f\n", sqrt(4));**

correctly evaluates sqrt(4) and prints the value 2.000.

- The function prototype causes the compiler to convert a *copy* of the integer value 4 to the double value 4.0 before the *copy* is passed to sqrt.
- In general, *argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called*.

- These conversions can lead to incorrect results if C's usual arithmetic conversion rules are not followed.
- These rules specify how values can be converted to other types without losing data.
- In our sqrt example above, an int is automatically converted to a double without changing its value. We saw this a type promotion.
- However, a double converted to an int *truncates* the fractional part of the double value, thus changing the original value.
- Converting large integer types to small integer types (e.g., long to short) may also result in changed values.
- The usual arithmetic conversion rules automatically apply to expressions containing values of two or more data types (also referred to as mixed-type expressions) and are handled for you by the compiler.
- In a mixed-type expression, the compiler makes a temporary copy of the value that needs to be converted then converts the copy to the "highest" type in the expression—the original value remains unchanged.
- The usual arithmetic conversion rules for a mixed-type expression containing at least one floating-point value are:
  > –If one of the values is a long double, the other is converted to a long double.
  > –If one of the values is a double, the other is converted to a double.
  > –If one of the values is a float, the other is converted to a float.

Function : variables - types and duration

- Earlier, we used identifiers for variable names.
- The attributes of variables include name, type, size and value.
- Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.
- C provides the storage class specifiers: auto, register, extern and static.
- An identifier's storage class determines its storage duration, scope and linkage.
- An identifier's storage duration is the period during which the identifier exists *in memory*.
- Some exist briefly, some are repeatedly created and destroyed, and others exist for the program's entire execution

Scope:

- An identifier's scope is where the identifier can be referenced in a program.
- Some can be referenced throughout a program like global variables, others from only portions of a program like local variables in a function

Linkage:

- An identifier's linkage determines for a multiple-source-file program whether the identifier is known only in the current source file or in any source file with proper declarations.

Duration:

- The storage-class specifiers can be split into automatic storage duration and static storage duration.
- Keyword auto is used to declare variables of automatic storage duration.
- Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.

Local Variables

- Only variables can have automatic storage duration.

- A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration.
- Keyword auto explicitly declares variables of automatic storage duration.
- Local variables have automatic storage duration by *default*, so keyword auto is rarely used.
- For the remainder of the text, we'll refer to variables with automatic storage duration simply as automatic variables

Static Storage Class

- Keywords extern and static are used in the declarations of identifiers for variables and functions of static storage duration.
- Identifiers of static storage duration exist from the time at which the program begins execution until the program terminates.
- For static variables, storage is allocated and initialized *only once*, *before* the program begins execution.
- For functions, the name of the function exists when the program begins execution.
- There are several types of identifiers with static storage duration: *external identifiers* (such as global variables and function names) and local variables declared with the storage-class specifier static.
- Global variables and function names are of storage class extern by default.
- Global variables are created by placing variable declarations *outside* any function definition, and they retain their values throughout the execution of the program.
- Global variables and functions can be referenced by any function that follows their declarations or definitions in the file.
- This is one reason for using function prototypes—when we include stdio.h in a program that calls printf, the function prototype is placed at the start of our file to make the name printf known to the rest of the file.
- Local variables declared with the keyword static are still known only in the function in which they're defined, but unlike automatic variables, static local variables retain their value when the function is exited.
- The next time the function is called, the static local variable contains the value it had when the function last exited.
- The following statement declares local variable count to be static and initializes it to 1.
- **static int count = 1;**
- All numeric variables of static storage duration are initialized to zero by default if you do not explicitly initialize them.

Here is a simple program using static variable

```
#include <stdio.h>
int getMyAge ( )
{
  static int  age = 80 ;
       //   age is static , it is initialized to 80 when the function is called for the first time.

  age++;   //   age is incremented to 81

  return age;  //   81 is returned.
}


main ( )
{

      printf (" Last year, My Age was %d \n", getMyAge ( )  )  ;
      printf ("This  year my age is %d \n", getMyAge ( ) ) ;
      printf ("Next year, my age would be %d \n", getMyAge ( ) ) ;

}
```

Pass by Value and Reference

Function - Passing variables by value

- In many programming languages, there are two ways to pass arguments—pass-by-value and pass-by-reference.
- When arguments are *passed by value*, a *copy* of the argument's value is made and passed to the called function.
- Changes to the copy do *not* affect an original variable's value in the caller.
- When an argument is passed by reference, the caller allows the called function to modify the original variable's value.
- Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable.
- When we pass the address of a variable, we call the variable is passed by reference. When we pass the value of a variable, we call the variable is passed by value.
- A good example of pass by reference is scanf function we used a lot. The function requires the address of the variable. When the function returns, it updates ie modifies the value of the variable.
- a good example of pass by value is the printf function we used a lot. The function requires just the value of the variable to be printed. It serve no purpose for it to have the address of a variable to print.

Here is the code snippet of a function swap

```
void swap (int , int  ) ;   // prototype of a function.  It takes two parameters of type int.
//  it returns nothing.  so we say void

void swap ( int a , int b )
{
   int temp = a ;
   a = b ;
```

```
   b = temp ;
   printf (" swap : The numbers are %d and %d \n ",  a,  b ) ;
}

int main ( )
{
     int x, y;
     printf ("Enter two numbers \n");
     scanf("%d %d", &x, &y);   // variables are Passed by reference
     swap (x, y) ;   // variables are passed by value
     printf ("main after calling swap:  %d and %d \n", x, y );
    return 0 ;
}
```

In the function swap, we swap the values, but the values in the main function are still not swapped. This is because we only sent the values to the function swap.

Function - Passing variables by reference

Example :

```
#include <stdio.h>
// function swap_1: pass by reference or pass address
void swap_1 ( int *, int * ); // prototype

void swap_1 ( int *x, int *y )
{
```

```
    int temp ;
    temp = *x ;
    *x = *y ;
    *y = temp ;
}

int main ( )
{
int a = 10, b = 20 ;
printf ( "Before swapping a=%d b=%d \n", a, b);
swap_1( &a, &b ) ; // watch passing the address
printf ( "After  swapping a=%d b=%d \n", a, b);
}
```

```
$ gcc ref1.c
$ ./a.out
Before swapping a=10 b=20
After  swapping a=20 b=10
```

Function - Passing Arrays

When you pass an array, you always have to pass the number of elements in that array to the function. For instance

```
void print_array ( int data [ ] ,  int count )
{
   int i ;
   // to traverse the array , you use the count variable in a loop
  for ( i = 0 ; i < count ; i++ )
    printf ( " %d  \n",   data [ i ]  )  ;
}
int main ( )
{
  int myDataArray [ 10 ] = { 4 } ;  //  assign 4 to index 0,  0 to rest of the cells
  print_array ( myDataArray ,  10 ) ;
 }
```

Shall we do another example with passing arrays ?

```
void bubble_sort ( int data [  ]  , int count )
{
    int i, j , temp ;
    for ( i = 0 ; i < count ; i++ )
        for ( j = 0 ; j < count - 1 ; j++ )
                if (  data [ j ] >  data [ j + 1 ] )
                     {
                         temp = data [ j ] ;
                         data [ j ] = data [ j + 1 ] ;
                         data [ j + 1 ] = temp ;
                      }
}
```

```
int main ( )
{
     int sort_data [ ] = {  5, 2, 9, 8, 3 }  ;
    bubble_sort ( sort_data, 5 )  ;
     print_array ( sort_data , 5 )   ;
}
```

Pass Arrays to Functions

- To pass an array argument to a function, specify the array's name without any brackets and also pass the number of cells in the array.
- For example, if array WeeklySalaries has been defined as

```
int  WeeklySalaries[ 52 ];
```

the function call

```
ComputeAnnualSalary( WeeklySalaries,  52)
```

passes array WeeklySalaries and its size to ComputeAnnualSalary.

- C automatically passes arrays to functions *by reference* —the called functions can modify the element values in the callers' original arrays.
- We know the name of the array evaluates to the address of the first element of the array.
- Because the starting address of the array is passed, the called function knows precisely where the array is stored.
- Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their *original* memory locations.

- For a function to receive an array through a function call, the function's parameter list must specify that an array will be received.
- For example, the function header for function ComputeAnnualSalary (that we called earlier in this section) might be written as
- **void** ComputeAnnualSalary **(int** data **[ ] , int size)**

indicating that ComputeAnnualSalary expects to receive an array of integers in parameter data and the number of array elements in parameter size.

- The size of the array is not required between the array brackets. See we have data with empty [ ]
- If it's included, the compiler checks that it's greater than zero, then ignores it.
- Specifying a negative size is a compilation error.
- Because arrays are automatically passed by reference, when the called function uses the array name data, it will be referring to the array in the caller (array WeeklySalaries in the preceding call)

```
void ComputeAnnualSalary   (  int data [  ] , int size )
{
        for ( i = 0 ; i < size ; i++ )
             annualSalary += data [ i ] ;
        printf ( "The annual Salary is %d \", annualSalary ) ;
}
```

Version 2: We know arrays are just like pointers, we could rewrite the function as

```
void ComputeAnnualSalary   (  int * data  , int size )
{
        for ( i = 0 ; i < size ; i++ )
             annualSalary +=  * ( data + i )  ; // NOTE IT.
        printf ( "The annual Salary is %d \", annualSalary ) ;
}
```

How individual cells are passed ?

- Although entire arrays are passed by reference, individual array cell elements are passed by value exactly as simple variables are.
- To pass an cell element of an array to a function, use the indexed name of the array element as an argument in the function call

```
PrintCellValue (  int cellValue ) {
   printf ( " The value is %d \n ", cellValue ) ;
}
```

.

```
//  The caller would call this function as

   for ( i = 0 ; i < 52 ; i ++ )

          PrintCellValue (  WeeklySalaries [ i ]  ) ;
```

Executing functions via Pointers

Now we will declare a pointer to a function , and execute the function using the pointer.

There are three steps in executing a function using a pointer.

Step 1 : Declare the pointer stating it points to a function and define the parameters and return types of the function.

for example, consider the following pointer definition.

```
int  ( * ptr )  ( int, int ) ;   // STEP 1
```

ptr is a pointer, it points to some function, the function takes two parameters of type int and returns a type of int.

Step 2 : Point the pointer to the actual function

```
  ptr = add ;      // STEP 2
```

add is a function that takes two parameters. Our ptr is now pointing to the function add.

Step 3: call the pointer with the argument values to execute the function.

```
        ptr ( 10, 5 ) ;    // STEP 3
```

now we are actually calling the function add passing two variables : 10 and 5

// DEMO EXAMPLE OF A POINTER TO A FUNCTION

```
#include <stdio.h>

int add ( int x , int y )

{
```

```
  return x + y ;
}

int sub ( int x, int y )
{
   return x - y ;
}

int main ( )
{
    // define the pointer that points to a function which takes two parameters and returns an int
    int  ( * ptr )  ( int, int ) ;   // STEP 1

   //  if we remove the (    ) around ptr,  like
   // int  *ptr  ( int , int )
  //  then  ptr is a function that takes two parameters and returns int pointer.  That is not what you want.  Putting ( ) around would make ptr is pointer to a function.

  // NOW YOU KNOW how to define a pointer to a function.

   ptr = add ;      // STEP 2
  int sum = ptr ( 10, 5 ) ;    // STEP 3


   ptr = sub ;     // STEP 2
   int minus = ptr ( 10 , 5 ) ;  // STEP 3
```

```
  printf ( "%d %d \n", sum, minus);



}


$ gcc ptr.c
$ ./a.out
15 5
```

Designing function parameters

When you write function, you have to ask :

What kind of variables should be function take as input - pass by value

what kind of variables should this function take as output/input - pass by reference

What kind of values should this function return: int, float, char, double , short, ...

what kind of address should this function return : char *, int *, float *, ...

Why do functions define const in front of the variables

Without asking these questions, you cannot write any functions. So, research these functions before attempting to complete the function.

Some example of function definitions.

A function may take a variable as reference as a parameter. When that happens, it predominantly means the function will change the value. Passing by reference means you are actually passing an address of that variable. An example:

char *strcat(char *dest, const char *src);

The const means the function will not change the value of src. The contents pointed by dest is changed by copying src to dest.

Another example :

char *strcpy(char *dest, const char *src);

The purpose seems to be same as strcat. dest is changed so it is passed as reference.

A function may take a variable as value as a parameter. When that happens, it predominanly means the function will not change the value, it merely wants a copy. Here is an example:

int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);

A function may take have two or more variables as parameters. Some variables may be passed by reference, some variables may be passed by value.

char *strncpy(char *dest, const char *src, size_t n);

- n is passed by value

- dest is defined as pointers

A function may return a value.

int rand(void);

A function may not return a value - void.

void srand(unsigned int seed);

A function may return an address

char *strtok(char *str, const char *delim);

A function may return an address of type void *

void *malloc(size_t size);

One bad programming: DO NOT RETURN THE ADDRESS OF LOCAL VARIABLES. Why ?

```
int * test (  )
{
   int x = 10;
   return &x ;  // why is this BAAAAAAAAD
}
```