Chapter 3

Functions

In the first-two chapters, we have come across programs that were not much complex both in terms of problem definitions and program/code development. However, realistic requirements would engage the programmer in developing large programs, larger in terms of the lines of code. Software engineering observations have been that the best way to develop and maintain large-sized programs is to construct it from smaller components or pieces each of which is easier to solve than the original problem. In algorithm terminologies this is called the divide and conquer strategy. Modules in C/C++ are called functions.

C/C++ programs are developed by combining both new programmer defined functions and predefined functions from the standard library. The standard library is quite exhaustive including commonly used functions for mathematical computations, standard input and output, string manipulations and so on. Such functions reduce the effort required on the programmer's side and thereby reducing development time and chances of errors. Good programming practice is to make use of available predefined functions to the extent possible and avoid reinventing the wheel. Another hard fact to be accepted is that programmers should spare their efforts in modifying or rewriting the standard library functions to make them efficient.

A function is called or invoked by a function call. The function call specifies the function name and provides (arguments) which the called function needs to carry out its task. The real world analogy is the hierarchical form of management. A boss (caller function) asks a worker (called function) to perform a task and return (report back) the results to the boss when the work is done. The boss/caller function is totally unaware of how the worker or the called function had performed its task. The worker function can again call other helper or worker functions.

However, for the main caller or boss function such details are hidden and looks as if the work was done by the immediately called worker function. Well in the lighter vein of it, this information hiding may lead to one person assuming credit for work of the other. However, this is accepted universally as a rule of nature. In fact, the main advantage with this approach is that the responsibility lies with the caller function and not with the called function in case of mishaps [Very rarely does this mishap occur but its occurrence does have a chance]. Whether or not this information hiding offers advantages in real life, it does offer key benefits in software engineering that will be explained later in Data Abstraction, an OOP feature.

3.1 Math Library Functions

The math library functions allow the programmer to perform certain mathematical computations. In this section, we shall look at a few such functions. Functions are called by writing the

name of the function followed by a left parenthesis, then the argument list is comma-separated and finally followed by a right parenthesis. For example, to compute the square root of a number using the predefined `sqrt()` function, the programmer would call it in this manner of `sqrt(625.0)` which would return the square root of the number. Here, `625` is the argument to the function `sqrt()`; the square root function accepts and returns a double argument.

All functions of the math library header file return data type double. Math library functions are made use of in the code by including the header file `<math.h>` that contains the declarations for various mathematical functions such as `pow()`, `sin()`, `cos()`, etc. Function arguments can be constants, variables or expressions. For example, the same sqrt function can be called `sqrt(a+d*f);` assuming `a, d, f` have been suitably declared and initialized as `a=24, d=5` and `f=5`. A few other available math functions are shown in Table 3.1. The above list is not exhaustive. Readers should refer to the `<math.h>` header file for an idea about the entire range of such math functions available with the language.

Table 3.1: Math library functions

Function	*Description*	*Example*
`ceil(x)`	Round `x` to the smallest integer `>= x`.	`ceil (5.2) = 6`
`cos(x)`	Trigonometric cosine of `x` in radians.	`cos ( 0 ) = 1`
`pow(x,y)`	`x` raised to the power `y`	`pow (3,5) =243`

Functions allow programmers to modularize the program. Variables declared within a function definition are called local variables, referring to the fact that they are known only within the function in which they are declared. Such variables are referenceable (accessible) only from within the function. Most functions have a parameter list (called arguments), they provide the means for communicating information between functions. A function's parameters are also local variables.

Key advantages of functions/program modularization are as follows:

1. Divide and conquer makes program size more manageable.
2. Modularization promotes software reusability with programs being created from standard functions that perform the required tasks rather than being created by the user.
3. Avoid code repetition. Codes that are repetitive in nature can be packaged as function(s) and executed from several locations by invoking/calling the function.

A good coding practice is to have functions performing limited or well-defined task and the name of the function, also called function identifier should be meaningful and express the task that it is performing. This promotes readability and reusability. For example, a function that is intended to perform the sum of matrix elements passed to it should be named as `matrix_sum()` [variable names/ function names or identifiers can have underscores in them]. Reusability that is promoted by good modularization is one of the key features of object-oriented programming.

Readers in fact have been already exposed to the concept of defining functions when they defined the function `main()`. Tasks which the user wanted to perform (either single line instructions or statements or subtasks or functions were defined in function `main`. `Main()` is one compulsory function that cannot be given any other name. `main()` will have statements [assignment or computational, control and repetitive structures, function calls, user-defined or predefined functions] specified or defined in them. In fact, in the first chapter as a part

of header file inclusion concepts, we had a detailed discussion on function declarations and function definitions. Those concepts are applicable here as well. The following example shown in Figure 3.1 will define a function square to compute the square of an integer argument passed to it.

```
1.  #include<stdio.h>
2.  int square ( int );
3.  int main()
4.  {
5.  int number;
6.  printf("Enter the no for which you want the find square \n");
7.  scanf ("%d", &number);
8.  printf("Square of %d is %d", number, square (number));
9.  return 0;
10. }
11. int square ( int a )
12. {
13. return ( a * a );
14. }
```

Figure 3.1: C code to compute square of a number using functions.

3.2 Code Interpretation

Line 2 is the declaration of the function square or prototype for function square. As has been already explained `int square (int)` identifies that the function square would accept one integer argument and at the end of processing return an integer as the result of the computation. Function declaration's main purpose is to handle mismatched function calls at compilation stage itself rather than leaving it till run-time. Note that in function declaration variable name or identification is unnecessary because it is only the number and data type of each argument that is important than the variable name.

They act as what are called place holders and it is the place (number, data type of arguments) that is important. However, function declaration is not required if the function definition appears before the first use of it in the program, in which case the definition also acts as a prototype. The void return data type indicates that the function returns nothing (does not return any value).

3.3 Possible Coding Errors with Functions

Possible programming errors while defining functions are as follows:

- Returning a value from a function whose return type has been declared as void
- Forgetting to return a value from a function that is supposed to return value
- Missing out on return type of a function while defining it.

The parameter list of a function definition is comma-separated containing the declarations of the parameters the function receives when it is called.

A function that does not accept any arguments can clearly indicate that the argument list is empty with the keyword void or by leaving it simply empty such as `int square (void);` or `int square();` Both function declarations mean the same. All the arguments in the parameter list should have an explicit data type mentioned. Errors that are possible with function declarations are:

Not mentioning explicit data types for all the arguments in the parameter list. For example, a function header of the form float `somefunc(float x, y)` will cause a syntax error. The float data type is applicable to the argument x only and not to y which is the case with normal variable declarations. The normal variable declaration syntax is not applicable for declaring the arguments of a function header and requires individual data types. The correct form of the above function header will be as follows:

```
float somefunc ( float x, float y)
```

Placing a semicolon at the end of the function header while defining a function will cause syntax errors. Declaring/defining a function parameter (argument) as a local variable within the function definition leads to syntax errors).Avoid having the same names for the arguments passed to a function and the variables within the definition; this may lead to ambiguity. Missing out on the braces (parentheses) during function `call` or `invoke` causes syntax errors.

The declarations and statements within the { } of a function forms the function definition. It is also referred to as the body of the function or block. A block is generally a compound statement that includes declarations and other statements. Blocks can be nested and contain variable declarations in them. But then a function cannot be defined inside another function, it leads to syntax errors. Another possible syntax error occurs if the function prototype, function header and function calls do not agree in the number, type and order of arguments and parameters, and in return value type.

Software engineering observations have been that a function requiring many arguments is probably task-extensive and is an ideal candidate for further subdivision into smaller functions. Programs coded as collection of small functions are easier to maintain, modify and debug. We have already seen that smaller functions promote software reusability. The `return` keyword is used to return value from a function. An empty `return` statement (void return type) simply passes the program control to the caller function to resume execution from the next instruction or statement onwards (in sequence). A `return` statement followed by an expression or value passes the value to the caller function and stores it in the corresponding LHS variable. Let us now develop a simple program to compute the maximum of three numbers using functions, code for which is shown in Figure 3.2.

In function `main` three integers are declared and accepted from the user. The three integers are passed to the function `maximum` which computes the `maximum` (logic is pretty simple) and returns the computed `max (local_x)` to the `main` function which is captured in the variable `max` and then finally displayed along with the three integers input by the user.

One important feature of function prototypes is the coercion of arguments forcing the arguments to the appropriate data type required. Most often the arguments passed by users of predefined or standard library functions will not be in exact match with their respective function prototypes available in header files. One cannot enforce the restrictions that the arguments passed by end-users of predefined functions match exactly. Hence the requirement

```
1.  #include<stdio.h>
2.  int maximum ( int, int , int );
3.  int main ()
4.  {
5.  int num1, num2, num3, max;
6.  printf("Enter the three numbers \n");
7.  scanf ( "%d %d %d",&num1,&num2,&num3);
8.  max = maximum ( num1, num2, num3);
9.  printf ("The maximum of %d , %d & %d is %d",num1,num2,num3,max);
10. return 0;
11. }
12. int maximum ( int a , int b , int c)
13. {
14. int loc max= a;
15. if ( b > loc max)
16. loc max=b;
17. if (c > loc max)
18. loc max=c;
19. return loc max;
20. }
```

Output:

Enter the three numbers

20 40 60

The maximum of 20,30 & 40 is 40.

Figure 3.2: C code to compute maximum using functions.

to convert the arguments of end users to the proper type before the respective function is called.

Conversion from lower byte consumption to higher byte consumption is called promotion and vice versa is called demotion. Such promotions/demotions should follow the rules of the language. For example, the `int` argument passed to the `sqrt()` function gets coerced to double type without changing the value. However, the vice versa conversion would truncate the fractional part and will result in changed values. More exhaustive programming examples involving functions shall be explored in the further sections.

3.4 Storage Classes

In the previous sections, we have seen that a variable gets identified by a name and a data type. In fact, it is based on this identification that variable names are also referred to as identifiers. In addition to these properties of name and data type, each identifier in a program has other attributes/features of storage class, scope and linkage. In this section, we shall elaborate on each of these concepts. The rest of this section to follow reviews the variable

declaration concepts. The storage classes available are: `auto` (automatic), `register`, `extern` (external) and `static`. C++ in addition to the above specifiers supports a mutable storage class specifier which will also be explained here, purely from a better understanding point of view.

An identifier's storage class determines the period or duration during which the identifier exists in memory. Certain identifiers exist in memory for a short period. Certain identifiers are repeatedly created and destroyed [allocation followed by deallocation] while certain identifiers exist for the entire duration of program execution. An identifier's scope identifies the places in a program where an identifier can be referenced. Certain identifiers can be referenced throughout the program while others can be referenced only from within a block / limited portions of the program.

An identifier's linkage determines the reference ability of variables or identifiers across source files, a concept that shall be explored with multiple source file development. The storage class specifiers are grouped in two classes, namely the automatic storage class and the static storage class. The keywords `auto` and `register` are used to declare variables of the automatic storage class type. Such variables are created when the block in which they are declared is entered, exist as long as the block is active and they are destroyed when the block is exited. Automatic storage class specifier is applicable only for variables.

A function's local variables and parameters are of the automatic storage class by default, explicitly declares variables of the automatic class. The syntax is as follows: `auto int x,y;` explicitly declares variables `x` and `y` to be of integer data type and automatic storage class. Such variables exist only in the function body where their declaration or definition appears.

A function's local variables that are of automatic storage class type by default are in short referred to as Automatic Variables. The nature of automatic variables conserves memory space. It is based on the principle of least privilege, not to have unreferenced or unnecessary variables in memory. Data in the machine language version of a program are loaded in registers for calculations and other processing. The storage class specifier register can be placed before an automatic variable declaration to denote that the variable is maintained in high speed registers than in memory.

Access from registers is fast in comparison to access from main memory. Frequently accessed variables those that require frequent updation (read and write) can be maintained in hardware registers thereby eliminating the overhead of repeatedly loading variables from memory to register and back to memory after updation. Multiple storage class specifiers applied for identifiers cause syntax errors.

Variables such as loop counter, cumulative total are better off to be stored in registers rather than in main memory to avoid the above-mentioned overhead. In case of insufficient registers, the compiler might ignore the register declarations and store it in memory itself. The `register` keyword is applicable only with local variables and function parameters. The syntax for a register type variable as follows: `register int count = 1.`

In fact, with the modern optimizing compilers frequently referenced variables are automatically placed in registers even without the register declaration. The next storage class type of static is declared using the key words `extern` and `static`. Static storage class variables exist from the point of program execution. Static variables have storage allocated and initialized when the program begins execution. Even though such variables and functions exist from the point of execution scope still has weightage regarding the `reference` ability of the variables or functions.

Two types of external identifiers with static storage class are global variables and function names, and local variables with static storage class specifier. Global variables and function names default to the `extern` storage class specifier. Global variables are created by placing variable declarations outside the function definition. Such global variables retain their values throughout the program execution. Global variables or functions can be referenced by functions that follow their declaration and definition in the source file. Global variables when compared with local variables allow unintended side effects to occur when a function that does not need access accidentally modifies it. They are similar to goto statements and should be avoided to the extent possible and used only in exceptional cases.

Variables that will be accessed only within functions should be declared as local rather than global variables. Local variables (within function) declared with static storage class specifier are known only in the function where they are defined, but unlike automatic variables static variables retain their values when the function is exited so that the next time the same function is called again static variables contain values they had when the function last exited. All static storage class variables are initialized to zero, whereas auto storage class identifiers are initialized with garbage or junk values.

3.5 Scope Rules

The portion of a program where an identifier has meaning is known as its scope. When we declare a local variable in a block it can be referenced only from within the block or blocks nested within it. This is referred to as block scope. The other scopes available for identifiers are function scope, file scope, function prototype scope and block scope. An identifier declared outside any function has file scope. Such an identifier is known in all functions from the point of declaration until the end of the file. Goto labels (identifiers with :) are the only variables that have function scope. Labels cannot be referenced outside the function body. Labels are also made use of in `switch` structures as case labels. Identifiers declared inside a block have block scope.

Block scope begins at the right brace (}) of the block. Local variables declared at the beginning of function definition and function parameters (local variables of functions) have block scope. When blocks are nested and an identifier in an outer block has the same name as that of an identifier in the inner block, the outer block identifier is hidden till the inner block terminates. Within the inner block it is the inner block's local identifier that is applied and the identifier in the enclosing outer block with the same name is ignored (hidden) within the inner block. As we have already stated, the static local variable that exists for the entire duration of program execution still has only block scope.

Storage class does not influence the scope of an identifier. Identifiers used in the parameter list of a function prototype have function prototype. For such identifiers names are not required, as they serve only as place-holders, the compiler ignores such names and only the individual types are required. Identifiers used in function prototype can be reused in the program without any ambiguity. Though it is syntactically allowed, it is a better practice to avoid identical identifier names within nested blocks. Such duplicate identifiers though allowed if not referenced properly can cause serious logic errors. The following example shown in Figure 3.3 helps in better understanding of storage and scope rule concepts.

```
1.  #include<stdio.h>
2.  void function1(void);
3.  void function2(void);
4.  void function3(void);
5.  int glob var a=1;
6.  int main ()
7.  {int a = 5;
8.  printf ("Local a in outer scope is %d",a);
9.  {
10. int a=8;
11. printf ("Local a in inner scope is %d",a);
12. }
13. printf ("Local a in outer scope is %d",a);
14. function1();
15. function2();
16. function3();
17. function1();
18. function2();
19. function3();
20. printf("Local a in main is %d",a);}
21. void function1(void)
22. {int a =30;
23. printf ("Local a on entering function1 is %d",a);
24. ++a;
25. printf ("Local a on leaving function1 is %d",a);}
26. void function2(void)
27. {static int a = 40;
28. printf("Local static a on entering function2 is %d",a);
29. ++a;
30. printf("Local static a on leaving function2 is %d",a);}
31. void function3 (void)
32. {printf("Global a on entering function3 is %d", a);
33. a = a * 20;
34. printf("Global a on leaving function 3 is %d", a);}
```

Figure 3.3: C code to illustrate storage classes and scope.

Before function `main`, a global variable `a` is declared and initialized (`a = 1`). Blocks where the identifier `a` is redeclared, the global value of `a` is hidden hence the global value is alive and active. Within function `main`, an auto local variable `a` is declared (initialized with 5). Within an inner block, of `main`, `a` is again redeclared and initialized with 8. Within each inner block, the enclosing outer block, value of `a` is hidden. The function set [`function1, function2, function3`] are called twice [basically to differentiate the initialization logic associated with global, auto local and static local variables].

Within `function1` a local automatic variable is declared and initialized with 20. This variable is incremented. This operation is to distinguish the starting point values with each storage class type. Within `function2`, a local static copy of `a` is created and initialized with 40. It is again incremented for reasons explained earlier. In the last `function3`, there is no local copy of `a` that is created and the references to `a` actually refer to the global value of `a` defined outside `main()`. Note that if a global variable is not defined and `function3` [that does

not have a local a defined] refers to a, then a syntax error of the form variable undeclared is thrown. Output of the code shown in Figure 3.3 is shown in Table 3.2.

Table 3.2: Output of the code shown in Figure 3.3

Output:
Local a in outer scope 5
Local a in inner scope 8
Local a in outer scope 5
Local a on entering function1 20
Local a on leaving function1 21
Static a on entering function2 40
Static a on leaving function2 41
Global a on entering function3 1
Global a on leaving function3 20
Local a on entering function1 20
Local a on leaving function1 21
Static a on entering function2 41
Static a on leaving function2 42
Global a on entering function3 20
Global a on leaving function3 400
Local a in main 5.

Note that the local automatic variable a in `function1` is created and destroyed each time the function is entered or exited. That is the reason for the incremented value (done in first call of `function1`] not reflected in the second call of `function1`. But the static local variable a in `function2 ()` retains its value when `function2 ()` is exited and reentered the second time. That is the reason for the modified value of a getting reflected in the second call of `function2 ()`. References to a within `function3` refers to the global value of `a =1` which is modified by the two calls to `function3`. The last statement in `main` refers to the local a defined in the scope of `main`. References to a outside `main` and in functions other than `function1` and `function2` refers to the most recently updated global a.

3.6 Recursion

All along the programs were developed by calling other functions to perform the task. However, in certain occasions it may be required to have function calling themselves within their definition. This is referred to as Recursion. Two types of recursion are: direct and indirect recursion. A function calling itself as an explicit statement is direct recursion. A function calling another function, which in turn calls the earlier or boss function, is indirect recursion. Examples of the above types of recursion are shown in Table 3.3.

Recursion can also be viewed as a divide and conquer strategy. A recursive function has solution to the problem only for the simple or what are called base cases [most often it is the smallest problem size]. The recursive function keeps dividing the problem [to smaller subproblems] until the base case is reached for which solution is known. The recursion step normally involves a return statement since its results will be combined with the portion of the problem, which lead to this call. All the solutions of the n recursive calls are combined to

Table 3.3: Types of recursion

Direct recursion	*Indirect recursion*	
`void function1 ()`	`void function2()`	`void function3`
`{`	`{`	`{`
⋮	⋮	⋮
`funtion1();`	`function3();`	`function2();`
⋮	⋮	⋮
`}`	`}`	`}`

form the overall solution to the problem and pass it back to the original caller function, which would normally be `main()`.

To make recursion solvable, the newly created problem [divided subproblems] must resemble the original problem in nature but should be slightly simpler or smaller than the original problem in terms of problem size. Since the new problem resembles the original problem, the function launches [calls] a fresh copy of itself to work on the smaller problem. This is the recursive step or recursive call. The recursive call or recursion step executes while the original call to the function [first call] is still open and not yet finished executing. The recursion step can result in many more such recursive calls as the function keeps dividing each new subproblem with which the function is called into smaller pieces.

As we have already stated that this division continues until the base case is reached which is also referred to as recursion bottoming out. At that bottoming out point, the function recognizes the base case, returns a result to the previous copy of the function and a sequence of such returns happens all the way up the line until the original call of the function eventually returns the final result to the main function. As an example let us compute the factorial of a number using a recursive program. Towards the latter half a non-recursive solution for the same problem is implemented.

Problem: Develop a function called factorial to compute the factorial of an integer passed to it. The mathematical formula `n!= n(n-1)!` is made use of to implement a recursive solution to the above problem. The base case of the recursion is `n=1` for which the factorial is `1`. Let us now see the program to perform `n!`, code for which is shown in Figure 3.4. The list of recursive calls for `5!` is shown in Figure 3.5 for better understanding. The forward arrows indicate the recursive calls till the base case of `n=1` is reached. When the base case is reached, the last recursive call returns the result of the base case to the previous call and such returns proceed up till the original call in main is reached. The reverse arrows indicate this backward return.

A non-recursive solution to the above problem could be achieved with the following `for` structure:

```
(for int ctr=number; ctr >=1; ctr --) factorial = factorial * ctr;
```

The above `for` structure should be inserted in a valid C code. The loop structure is based on `5!= 5 * 4 * 3 * 2 * 1 = 120`.

```
1. #include<stdio.h>
2. int factorial(int);
3. int main()
4. int number,result;
5. printf("Enter the number for which the factorial
   is to be computed\n");
6. scanf("%d",&number);
7. result= factorial(number);
8. printf("Factorial of %d is %d",number,result);
9. return 0;
10. }
11. int factorial (int no)
12. if (no<=1)
13. return 1;
14. else
15. return no * factorial ( n-1 );
16. }
```

Figure 3.4: C code using recursion to compute n!.

5!→	5*4!→	4*3!→	3*2!→	2*1!→	1!
120	←5*24	←4*6	←3*2	←2*1	←1

Figure 3.5: Trace of `n!` using recursion for `n=5`.

3.6.1 Fibonacci Series using Recursion

The fibonacci series 0,1,1,2,3,5,8,13,21 begins with a 0 and 1 and the subsequent numbers are the sum of the previous two-fibonacci numbers. Fibonacci series can be recursively defined as

```
fib( n ) = fib ( n-1 ) + fib ( n-2 )
```

Base cases:

```
fib ( 0 ) = 0, and fib ( 1 ) = 1;
```

Now using the above recursive and base cases let us develop the code to generate the *i*th fibonacci number ; `i` being an user input. Code for recursive version of fibonacci series generation is shown in Figure 3.6.

A pictorial representation for the `call fib (5)`, that is to generate the 5th fibonacci number is shown in Figure 3.7.

The recursive calls may be processed left to right or right to left. The programmer should make no assumptions regarding the processing (left–right) or (right–left). In most cases the results would be the same, however, in certain cases the results will not be the same. The first call to a recursive function cannot be counted as a recursive call. It is the subsequent calls (including the base case call) that make up the total set of recursive calls.

```
 1. # include<stdio.h>
 2. int fib(int);
 3. int main ()
 4. {
 5. int number,result;
 6. printf("Enter the number \n");
 7. scanf("%d",&number);
 8. result = fib (number);
 9. printf("The %dth fibonacci number is %d",number,result);
10. return 0;
11. }
12. int fib (int no)
13. {
14. if ( no == 0 ||no ==1)
15. return no;
16. else
17. return fib(no -1)+fib(no-2);
18. }
```

Figure 3.6: C code to generate fibonacci series using recursion.

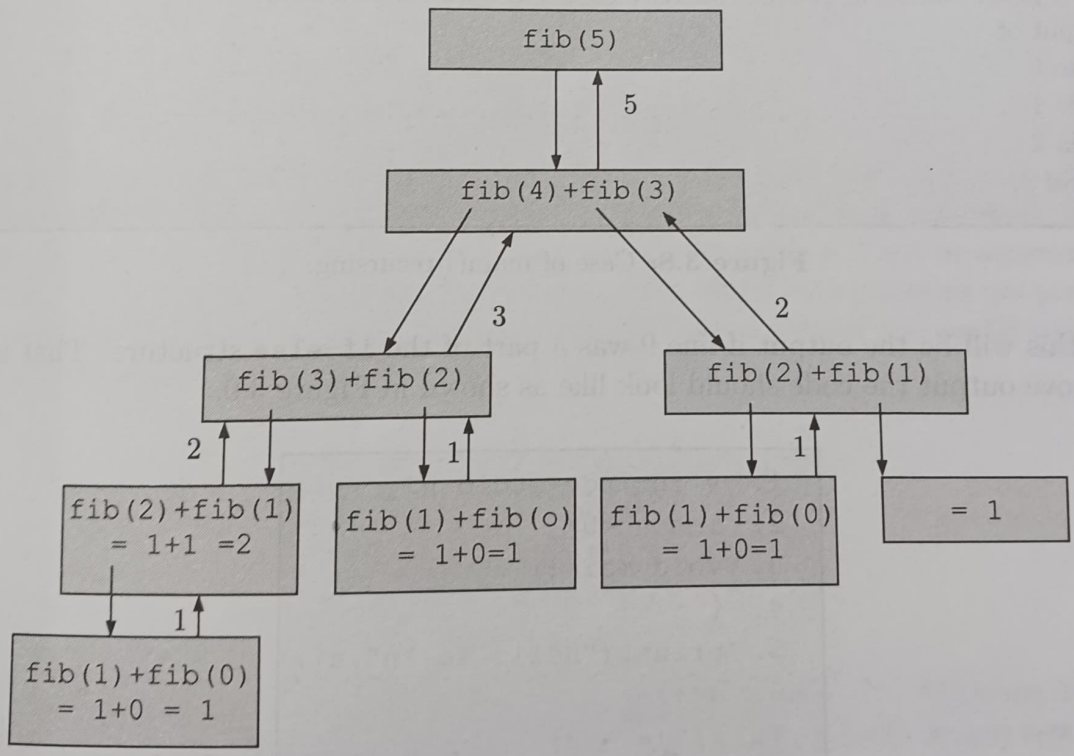

Figure 3.7: Trace of fibonacci code for `n=5`.

Developing programs that depend on the order of evaluation of operators (subexpressions) can lead to errors since it is a compiler specific issue and varies across compilers. Hence systems built using such programs will not remain stable. In the above pictorial representation, the forward arrows denote the recursive call to function `fib` and the reverse arrows indicate the

value returned after each recursive call terminates. Let us now explore two examples that shall strength our understanding of the concepts of recursion. Readers are advised to walk through the code, trace through each recursive call separately to predict the output.

3.6.2 Another Example for Recursion

What is the output of the following code shown in Figure 3.8?

```
1. # include<stdio.h>
2. int a =0;
3. void main()
4. {
5. printf("Hello %d ₩n",a);
6. a++;
7. if (a < 3)
8. main();
9. printf("World %d \n",a);
10. }
```

Most programmers at the first instance come out with the following output of
Hello 0
Hello 1
Hello 2
World 3.

Figure 3.8: Case of main() recursing.

Well, this will be the output if line 9 was a part of the `if-else` structure. That is to say for the above output the code should look like as shown in Figure 3.9.

```
1. # include<stdio.h>
2. int a =0;
3. void main()
4. {
5. printf("Hello %d \n",a);
6. a++;
7. if (a < 3)
8. main();
9. else
10. printf("World %d \n",a);
11. }
```

Figure 3.9: Case of main() recursing.

But then this was not our original code. The world `printf()` is not a part of the `if-else` construct, but it is a part of the definition of `main()`. So, the number of times the function `main` gets called, this `printf` will also get executed that many number of times. By default `main` is called once by the compiler and twice as a result of the recursive calls [if - part]. Thus, in all `main` gets called three times. Thus, the `printf("World %d",a);` will also get executed three times. In fact, the actual output is

```
Hello 0
Hello 1
Hello 2
World 3
World 3
World 3
```

Having already established that the `world` display occurs three times, let us now look at the program control flow in the above recursion exercise, for which to understand we need to trace through each recursive call separately. The recursion sequence and return of control to the previous function can be understood from the diagrammatic representation given in Figure 3.10.

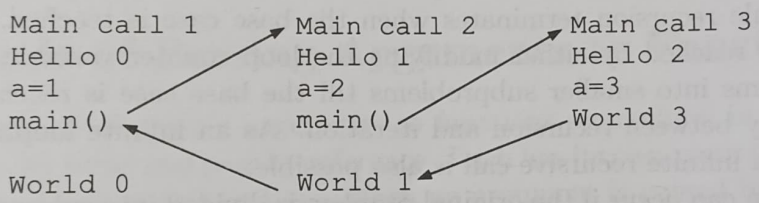

Figure 3.10: Trace of code shown in Figure 3.8.

The above trace should clarify the transfer of program control flow. Truly speaking the `world` string that appears on the screen should have the values of a associated with the respective recursive call. But then since a is a global variable, there will not be separate copies of a created in each call, but there is one global copy over which the updations are performed. Thus, the final stable state of `a (3)` is reflected in each of the display of the `world` string three times. Thus, the output of the above code is as follows:

```
Hello 0
Hello 1
Hello 2
World 3
World 3
World 3
```

This is a classical example to understand the nuances of recursion. The following interesting code shown in Figure 3.11 will be left as an exercise to the reader. Kindly do not test the code on a compiler. The whole charm of it will be lost.

3.6.3 Recursion vs. Iteration

Programmers should be aware of the fact that tasks that are solved by recursion can also be solved in a non-recursive (iterative) manner. Hence there is a need to differentiate recursion

```
1. # include<stdio.h>
2. int a =0;
3. void main()
4. {
5. printf("Hello %d \n",a);
6. a++;
7. if (a < 3)
8. printf("World %d",main());
9. return a;
10. }
```

Figure 3.11: Recursion exercise.

from iteration. Iteration and recursion both are based on control structures. Iteration uses a repetitive control structure, while recursion uses a selection control structure. The repetition in recursion is achieved through repetitive function calls. There is a termination test involved in both.

Iterative structures stop when the loop continuation condition specified in the iterative structure fails, while recursion terminates when the base case is reached. The terminating criterion in each is reached by either modifying the loop counter variable or by subdividing the original problems into smaller subproblems till the base case is reached. In fact, there is a lot of similarity between recursion and iteration. As an infinite looping is possible with iteration so does an infinite recursive call is also possible.

Infinite recursion can occur if the original problem is divided into subproblem(s) that never reach the base case or infinite recursion can as well occur as a result of no condition governing the recursive call. The overhead in terms of the processor time and memory space associated with recursion is more or large when compared with their counterparts, namely iteration. Each recursive call creates a fresh copy of the function (only the variables) that can consume reasonable amount of memory. And also the control transfer or return in recursion could be more.

All such bottlenecks are not associated with iteration. Thus it is more often iterative solutions that is favoured in comparison with recursive solutions. However, to ease out code understanding and debugging recursion may be opted. And also when an apparent iterative solution is not visible and recursion is a natural reflection in the problem definition (subdivisional nature), recursion may be the choice. But never choose recursion when system performance is a key issue.

Note: This being an OO text, in the initial few chapters we have explored structured programming, elaborating on the advanced structured programming concepts available only in C++. The following discussions and in fact, the earlier discussions on function prototypes are with C++ in mind.

3.7 Empty Parameter List Functions

As we have already stated functions that do not accept any arguments is indicated by an empty braces () or explicitly mentioning the keyword `void` within the parentheses [`(void)`].

Both syntaxes are valid and mean the same. However, in C an empty parentheses [not using the keyword `void`] would disable all type checking with respect to this function call and this allows the function to be called with any number or type of argument.

3.8 Inline Functions

Function calls lead to transfer of control and return. This transfer or return of control can cause substantial execution time overhead. To avoid this transfer or return of program control C++ provides inline functions. A function that has been declared to be inline in the function header of a function definition instructs the compiler not to transfer control while paste the function definition at the invocation or call point.

The trade-off is that multiple copies of the same code get inserted at its various invocation points rather than maintaining a simple copy and transfer or return control to or from it. The inline qualifier is chosen normally for smaller functions (small in terms of lines of code of function definition). If used with larger sized function it can improve upon execution time but increase the program size. It is a user- or programmer-dependent issue and it is he who should make a judicious choice between execution time or size trade-off.

3.9 Methods of Passing Arguments to Functions

Two common methods of passing arguments to functions or modules in many programming languages are pass by value and pass by reference. Each has its own merits and demerits, which we shall explore in detail in this section. When an argument is passed by value, the function definition creates a copy of the arguments passed (local copy) to it. References to variables or modifications are done over this duplicate copy of the variables or arguments defined in the caller function. In fact, two types of parameters associated with functions are formal and actual parameters.

Formal parameters are the variable names in the function header of function definition for the purpose of referring to within the function definition because functions do not know the actual arguments or variables until they are called. Thus, the parameters with which a function are gets called in the caller or boss function are called actual parameters, while the parameters that are used in the function header of function definition for manipulation within the definition are called formal parameters.

Thus, in pass by value a duplicate copy of the arguments passed is created and it is this local copy that is referenced within the function definition. For example, a function `swap (a,b)` [intended to exchange the values of `a` and `b`] in which `a` and `b` are actual parameters creates a duplicate copy of `a` and `b` when mapped with the formal parameters of the function definition. The swapping is done over this duplicate copy and as a result the swapped values are reflected within the function definition and not in the caller function. The C code to swap by value is shown in Figure 3.12.

Thus, it can be seen clearly that the swapping effect is reflected only within function definition [the `printf` statement within the function swap] and not reflected in the caller or main function [as shown by the `printf` at line 9]. The advantage with pass by value mechanism is that changes are made only over the copy and do not affect the original values in caller. This prevents the accidental side effects that are so much crucial to good program development.

```
1. #include<stdio.h>
2. void swap (int,int);
3. int main()
4. {
5. int number1,number2;
6. printf("Enter values for number1,number2 \n");
7. scanf ("%d %d",&number1,&number2);
8. swap(number1,number2);
9. printf('The numbers after calling swap in caller function
   is %d %d',number1,number2);
10. return 0;
11. }
12. void swap ( int forno1,int forno2)
13. {
14. int temp;
15. temp = forno1;
16. forno1=forno2;
17. forno=temp;
18. printf("Swapped value in function swap is %d %d
   \n",forno1,forno2);
19. }
```

Output:

Enter values for number1, number 2

20 30

Swapped Values in function swap is 30 20

Figure 3.12: C code to perform swap by value.

The disadvantage is that creation of duplicate copies can cause reasonable wastage of memory space.

Thus, it is the programmer who has to decide which is more important. In cases where memory is important, the second mechanism of pass by reference (address) is used. In this case, the address of the actual parameter is made available for the called function, that is it can access the data directly and hence modifications or updations are reflected in both function definition and the caller function. In fact, there is only one copy that is maintained in the caller function. The `swap` function when coded by pass by reference looks as shown in Figure 3.13.

The outputs clearly reflect the modifications in this case, swapping is reflected not only in the function definition but in the caller function `main` as well, since there is only one copy of `number1` and `number2` that is referenced within the function definition `swap()` and in `main()`. Pass by references are advantageous in that there is no wastage of memory since there is no duplicate copy creation. But the disadvantage is that the original variable is modified and hence accidental changes [side effects] can prove to be a bottleneck.

```
 1. #include<stdio.h>
 2. void swap (int &, int &);
 3. int main()
 4. {
 5. int number1,number2;
 6. printf("Enter values for number1,number2 \n");
 7. scanf("%d %d",&number1,&number2);
 8. swap(number1,number2);
 9. printf("The numbers after calling swap in caller function is
    %d %d",number1,number2);
10. return 0;
11. }
12. void swap ( int &forno1,int *&forno2)
13. {
14. int temp;
15. temp = forno1;
16. forno1=forno2;
17. forno2=temp;
18. printf("Swapped value in function swap is %d %d
    \n",forno1,forno2);
19. }
```

Output:
Enter values for number1, number 2
20 30
Swapped Values in function swap is 30 20
The numbers after calling swap in caller function is 30 20.

Figure 3.13: C code to perform swap by reference.

Note: The advantages of both pass by value and pass by reference can be combined by placing the `const` qualifier before the arguments passed by reference. Such a mechanism would not waste memory since there is no duplication and it being a constant reference, modifications (accidental changes—side effects] will as well not be allowed. This is a syntactic exploitation of the provisions in C and is not a separate passing mechanism.

The following function declaration `Void fn2 (const int & a1,const int &a2)` provides only read access over arguments `a1` and `a2` for `function fn2`. This can be made use of when the called or worker or helper functions need only read permissions over the caller functions variables and will not perform any modifications over it. Hence the following conventions are widely adopted while passing arguments:

1. Small, non-modifiable arguments are passed by value.
2. Large-numbered, non-modifiable arguments are passed as constant references.

3.10 Aliases

References can also be used as aliases for other variables within a function. Reference variables must be initialized in their declaration.

For example:

```
int b =1;
int &a =b;//alias for b
++a; //increment b using alias.
```

Both `a` and `b` refer to the same location. The alias is simply another name for the original variable and all operations supposedly performed on alias are performed actually on the original variable. A reference argument must be an lvalue and not a constant or expressions that return a value.

3.10.1 Possible Coding Errors with References

A few of the possible coding errors involving reference data and manipulation over reference variables are as follows:

1. Not initializing a reference variable at declaration point is syntax error.
2. Declaring multiple references in one statement and assuming that & distributes across a comma-separated list might cause programming errors. For example, `int &a,b`, declares `a` to be a reference variable and `b` to be a normal integer variable. If `b` should also be a reference variable then `&` should be explicitly applied to it like `int &a,&b`. Assuming that `&` distributes across a list of variables declaration might cause logical errors since there would be a mismatch between the way the compiler and the programmer treat the variable. For example, in a statement `int &a,b`, the compiler treats `a` as a reference variable and `b` as a normal variable while the programmer assuming that `&` distributes treats `b` also as a reference variable. Hence, programmers should stick to the syntax of explicitly applying (prepending) the `&` operator for reference variables before their names or identifiers.
3. Functions can also return references but this should be done with extra care. When returning a reference to a variable declared in the called function, the variable should be declared static within that function, otherwise, the reference refers to an automatic variable that is destroyed or discarded when the function terminates. Such a reference variable is said to be undefined and the programs behaviour would be unexpected. In fact, such references are also called dangling or dancing references. One more logical error is attempting to reassign a previously declared reference to be an alias for another variable. This would just copy the other variable's value to the location for which the reference is already an alias. Returning dangling references will not most often be caught at compilation stage, it leads to serious logical errors.

3.11 Default Arguments

We have already seen that caller functions pass arguments to called functions. In certain cases, the programmer can specify that the function is called with default arguments if arguments are not explicitly specified by the caller of the function. Such arguments called default arguments when omitted by the caller function are automatically replaced by the compiler and passed in the call. Default arguments must be the rightmost or trailing arguments in a function's parameter list.

When calling a function with two or more default arguments, if a missed argument is not the rightmost argument in the argument list, all the arguments to the right of that missed arguments must also be omitted. Default arguments should be specified with the first occurrence of the function name mostly the prototype. Default arguments can also be used with inline functions, constants and global variables. For better understanding of defaulting argument, let us look at a programming example for the same as shown in Figure 3.14.

```
1.  #include<stdio.h>
2.  int simpleinterest(int =1,int =1,int=1);
3.  int main()
4.  {
5.  int si;
6.  printf("Simple Interest function called with no user specified
    and all default values");
7.  si=simpleinterest();
8.  printf("Simple Interest function called with 1 user specified
    and 2 default values");
9.  si=simpleinterest(10);
10. printf("Simple Interest function called with 2 user specified
    and 1 default value");
11. si=simpleinterest(10,2);
12. printf("Simple Interest function called with 3 user specified
    and no default values");
13. si=simpleinterest(10,5,3);
14. return 0;
15. }
16. int simpleinterest(int prin,int yrs,int rate)
17. {
18. return prin*yrs*rate;
19. }
```

Figure 3.14: C code to demonstrate default arguments to functions.

Note that in lines 6, 8 and 10, the arguments with missing values are replaced with default values as specified in the function header of function prototype. In the first case (line 6), all three arguments are missing and replaced by default values. In the second case (line 8), one argument (`prin`) is specified by the caller and the remaining two arguments are defaulted. In the third case (line 10), two values are caller specified and the third argument is defaulted. In the last case (line 13), all the three arguments are user or caller specified and there is no defaulting that is performed by the compiler.

3.12 Global Variable Access

We have already seen that functions can declare variables local within them (local variables) or refer to variables defined or declared outside, called global variables. We have also seen that

or function header of function definition encodes the function name or identifier and data types of its parameters. This process is normally referred to as name mangling or name decoration and this ensures type safe linkage, to ensure that a function gets called with the correct data type(s).

Type safe linkage ensures that the correct overloaded function confirming to the argument types is invoked. Overloaded functions can have different return types but must mandatorily differ in their parameter lists. Let us now see a programming example for better understanding of function overloading. To compute the cube of an integer and float argument passed to a function called cube(), the C code is as shown in Figure 3.16.

```
1. #include<stdio.h>
2. int cube (int=1);
3. float (float=1.0);
4. int main()
5. {
6. int intno,floatno;
7. printf("Enter the values for integer and float number
\n");
8. scanf("%d %f",&intno,&floatno);
9. printf("Cube of the integer no %d is
%d",intno,cube(intno));
10. printf("Cube of the float no %f is
%f,intno,cube(floatno));
11. return 0;
12. }
13. int cube(int x)
14. {
15. return x*x*x;
16. }
17. float cube(float y)
18. {
19. return y*y*y;
20. }
```

Figure 3.16: C code to demonstrate function overloading.

The compiler uses only the parameter lists to resolve overloaded function calls. Hence functions with identical parameters, but list differing in return types, will not be function overloading. It only leads to syntax errors. There is no compulsion that overloaded functions must match in the number of parameters, but they definitely need to differ in the data types or precisely the signature. However, care should be taken while using function overloading with function defaulting values for its arguments.

A function with default arguments omitted may be identical to some other overloaded function, in which case it will be a syntax error. Of course, it should lead to a syntax error for the compiler would not be able to distinguish the two functions if this is allowed. Also

a program in which there is a function accepting no arguments (declared explicitly as not accepting any arguments and there is another function that contains all default arguments. This again will cause syntax error for the same reason mentioned earlier. Therefore, in a call of the function that has no arguments specified by the caller, the compiler will not be able to distinguish or resolve the two function definitions.

3.14 Function Templates

Well, function overloading was used to deal with functions that perform similar operations but involved different logic on differing data types. Now what if the case when the programming logic is also the same and it is only in the data types that such functions differ. This is an ideal situation for use of function templates even though it can be very well achieved using function overloading as well. But function templates provide a more compact and convenient solution for achieving the same.

The programmer writes a single function template definition, the generic function not relating to any specific data type. This function template is the generic specification of the programming logic involved with the function. Having completed the generic function specification, the compiler automatically generates separate template functions [functions generated out of or from function templates] to handle function calls differing in data types appropriately.

The compiler performs code generation on behalf of the programmer. Hence effectively the lines of code of the function definition are duplicated for the respective data type, or in other words, it is function overloading behind the scenes that is performed by templates with the difference being that the programmer need not specify the definitions for all the overloaded versions. Whereas he needs to complete the function definition for a generic data type and leave it to the compiler to generate data type specific definition of the function.

Thus, defining a single function template defines a whole or entire family of functions. There is no performance enhancement with the use of templates in comparison with user overloaded functions. It is just an efficient code generation strategy and nothing more than that. However, it makes programs more compact and readable. All function templates definitions begin with the keyword templates followed by a list of formal parameters as we had with normal function definitions. The formal parameter list is enclosed within <> angle brackets. The formal parameters could be built in or user defined types [structures or classes which shall be dealt with later], then the function definition follows which is similar to normal function definition. Let us now look at an example to swap two numbers, code for which is as shown in Figure 3.17. These numbers could be integers or float or double.

The above function template declares a single formal type parameter T. Then for each specific invocation of swap, the compiler substitutes the T in the function template specification with actual data type passed. This generated function (template function) is then compiled. As we have already seen it is only code generation performed by the compiler on the programmer's behalf. Once the codes have been generated for each separate invocation of the function template or when all the template functions have been generated from there on it is based on function overloading concepts that further actions take place.

```
1. #include<stdio.h>
2. template <class T>
3. void swap (T &v1 ,T &v2 )
4. {T temp;
5. Temp =*v1;
6. *v1=*v2;
7. *v2=Temp; }
8. int main()
9. {int int1,int2;
10. float no1,no2;
11. char a1,a2;
12. printf("Enter the integer numbers \n");
13. scanf("%d %d",&int1,&int2);
14. printf("Enter the float numbers \n");
15. scanf("%f %f",&no1,&no2);
16. printf("Enter the characters \n");
17. scanf("%c %c",&a1,&a2);
18. swap(int1,int2);
19. printf("Integers after swapping are %d %d",int1,int2);
20. swap(no1,no2);
21. printf("Float numbers after swapping are %f %f",no1,no2);
22. swap(a1,a2);
23. printf("Characters after swapping are %c %c",a1,a2);
24. return 0;}
```

Figure 3.17: C code to illustrate the use of function templates.

Review Questions

1. List the benefits of a function-based code development approach.
2. Differentiate the terms function declaration and function definition.
3. Differentiate an interface from implementation.
4. Develop a function cube which returns the cube value of the integer argument passed to it.
5. Develop a C program that uses functions to test if an input square matrix is a magic square or not. (Note a magic square is a square matrix in which the sum of all columns, sum of all rows, sum of main and trailing diagonals all equate to the same value.)
6. Develop a C program that uses functions to generate a magic square of a user-specified order.
7. Develop a C program that generates the determinant value of a square matrix.
8. Differentiate a static variable from an automatic variable.
9. Identify a few coding situations when the choice of registers is justified.

10. What do you understand by the term scope of a variable?
11. Differentiate scope from storage class.
12. Can you redeclare a variable in C? Justify your answer.
13. Differentiate recursion from iteration.
14. What is the mathematical principle that forms the basis of recursion? Explain with respect to the problem of factorial computation.
15. Develop a C program using recursion to solve the Towers of Hanoi problem. (There are three pegs, A, B and C. A is the source peg and C is the destination. A is initially composed of n disks with the smallest one on top and the largest in terms of size at the bottom. Objective is to transfer the disks from A to C using B as the intermediate peg, always satisfying the condition that a smaller disk is placed on top of a larger one.)
16. Differentiate call by value from call by reference with an example.
17. Explain the notion of default arguments supported with functions in C++.
18. Develop a C(/C++) program to generate the product of three numbers accepted from the user. The user could enter all integers or all floating point numbers. Use the concept of function overloading.
19. Argue the case for function templates over function overloading. Develop a function template to solve the problem defined in the earlier question.
20. Develop a C program using functions to compute the greatest common divisor of two numbers passed to it.