PROGRAMMING IN C

Chapter 4: Functions

Study Notes for Engineering Students

4.1 Functions

What is a Function?

A function is a self-contained block of program statements that performs a specific, well-defined task and can be executed whenever it is required in a program. Every C program consists of at least one function, main(), from which execution begins. Large and complex problems are broken down into smaller, manageable sub-tasks, and each sub-task is implemented as a separate function. In C, a function has a name, may accept a set of input values called parameters or arguments, executes a sequence of statements, and may send back a single value to the point from which it was called.

Need for Functions

Writing every program as one long sequence of statements inside main() quickly becomes unmanageable as the size of the program grows. Functions solve this problem and provide several practical benefits:

- Modularity: A large program is divided into smaller, logically independent units, which makes the overall design easier to plan and understand.
- Reusability: Once a function is written and tested, it can be called any number of times from different parts of the same program, or even from other programs, without rewriting the code.
- Avoiding code repetition: Instead of writing the same set of statements repeatedly, a function is written once and invoked wherever that task is needed.
- Easier debugging and testing: Since each function performs one task, errors can be isolated and corrected within that function without disturbing the rest of the program.
- Abstraction: A function hides the internal implementation details from the caller; the user of a function only needs to know what it does and how to call it, not how it is internally coded.
- Easier maintenance: Modifying the logic of a task usually requires changes in only one place, i.e., inside the function definition.
- Team development: Different functions of a large project can be assigned to and developed independently by different programmers.

Prototype Declaration

A function prototype (also called a function declaration) tells the compiler about a function's name, its return type, and the number and types of its parameters, before the function is actually used or defined. It allows the compiler to perform type checking on function calls even though the full definition of the function appears later in the program, or in another file altogether.

The general syntax of a function prototype is:

```
return_type function_name(parameter_type_1, parameter_type_2, ...);
```

For example:

```
int add(int, int);        // prototype
float average(float, float, float);
```

Parameter names in a prototype are optional; only the types are mandatory for the compiler. Prototypes are normally placed before main(), or in a header file, so that the compiler can check every call to the function for the correct number and type of arguments, and can correctly convert the return value where necessary.

> ***Note:*** *If a function is used without a prior prototype or definition, older compilers may assume it returns int, which can lead to undetected errors. Modern C compilers require a visible prototype or definition before a function is called.*

Scope and Lifetime of Variables

Scope refers to the region of the program within which a variable can be accessed or referred to by its name. Lifetime (also called storage duration) refers to the period during which a variable remains allocated in memory, i.e., from the time memory is allotted to it until it is deallocated.

Types of Scope

- Local (block) scope: A variable declared inside a function or a block (within { }) is a local variable. It is accessible only within that function or block and does not exist outside it.
- Global (file) scope: A variable declared outside all functions, usually at the top of the program, is a global variable. It is accessible from any function in the file, from the point of its declaration to the end of the file.
- Function prototype scope: Parameter names used only inside a function's prototype (declaration) have scope limited to that prototype statement.

Lifetime Categories

- Automatic variables exist only while the function or block in which they are declared is executing; memory is created on entry and destroyed on exit.
- Static variables retain their value between successive calls to the function and exist for the entire run of the program, even though their scope may still be local.
- Global variables exist for the entire duration of program execution, from the start of the program to its termination.

It is important to note that scope and lifetime are two different concepts: a static local variable has local scope (it can only be named inside its function) but a lifetime equal to the whole program.

Defining Functions

A function definition provides the actual body of the function, i.e., the statements that are executed when the function is called. The general form of a function definition is:

```
return_type function_name(parameter list)
{
    declarations;
    statements;
    return value;   // if return_type is not void
}
```

A function definition has three parts:

1. Function header: consists of the return type, function name, and the parameter list enclosed in parentheses.

2. Function body: the block of statements enclosed in curly braces { } that defines what the function actually does.
3. Return statement: sends a value back to the calling function and transfers control back to the caller (omitted or without a value when the return type is void).

Example — a function that returns the larger of two integers:

```
int larger(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Passing Parameter Types

Parameters (also called arguments) are the values passed to a function so that it can perform its task using those values. C supports the following parameter categories:

- Actual parameters: the values or expressions written in the function call statement, e.g., add(x, y).
- Formal parameters: the variable names listed in the function definition/header, which receive copies of, or references to, the actual parameters, e.g., int add(int a, int b).
- No parameters: a function may be defined to accept no arguments at all, in which case the parameter list is left empty or written as void.

The number, order, and data type of the actual parameters in the call must match the formal parameters in the function definition (or be compatible through implicit conversion), otherwise the compiler will report a type mismatch or the program will behave incorrectly.

Function Call — Call by Value and Call by Reference

A function call is a statement that invokes a function, transferring control to it along with any arguments. C provides two conceptual methods of passing arguments to a function:

Call by Value

In call by value, a copy of the actual parameter's value is passed to the formal parameter of the called function. The formal parameter and the actual parameter occupy separate memory locations. As a result, any changes made to the formal parameter inside the function have no effect on the original variable in the calling function. This is the default method of parameter passing in C.

```
void modify(int x)
{
    x = x + 10;     // only the local copy changes
}

int main(void)
{
    int a = 5;
    modify(a);
    printf("%d", a);   // prints 5, a is unchanged
    return 0;
}
```

Call by Reference

C does not have true reference parameters as some other languages do, but the same effect is achieved by passing the address of a variable (a pointer) as the argument. The formal parameter is then declared as a pointer, and the function accesses and modifies the original variable indirectly through this address using the dereference operator (*). Because the function operates directly on the caller's memory location, changes made inside the function are reflected in the calling function.

```
void modify(int *x)
{
    *x = *x + 10;   // changes the original variable
}

int main(void)
{
    int a = 5;
    modify(&a);
    printf("%d", a);   // prints 15, a is changed
    return 0;
}
```

Call by reference is essential whenever a function needs to modify more than one variable of the caller, or when large data structures (such as arrays or structures) need to be passed without the overhead of copying them.

Return Values

The return statement is used to end the execution of a function and, optionally, send a value back to the point from where the function was called. Its general form is:

```
return expression;   // for functions with a non-void return type
return;              // for functions with return type void
```

- A function can return at most one value directly through the return statement; the data type of this value must match the declared return type of the function.
- If a function's return type is void, it performs its task but does not send back any value to the caller.
- A function may contain multiple return statements along different execution paths (e.g., inside if-else branches), but only one of them executes for a given call.
- Once a return statement executes, control immediately passes back to the calling function, and any statements after it in the same function are skipped.
- Although a function can return only one value directly, multiple results can be sent back to the caller indirectly using pointer (call by reference) parameters.

4.2 Storage Classes, Categories of Functions and Recursion

Storage Classes

A storage class in C determines three properties of a variable: its scope (where it can be accessed), its lifetime (how long it exists in memory), and its default initial value. C provides four storage class specifiers:

1. auto

This is the default storage class for all local variables declared inside a function or block. Such variables are created when control enters the block and are automatically destroyed when control leaves it. Their scope is

local, and their default initial value is an unpredictable garbage value. The keyword auto is rarely written explicitly because it is applied automatically.

```
void demo(void)
{
    auto int x = 10;   // same as: int x = 10;
}
```

2. register

The register storage class requests the compiler to store the variable in a CPU register instead of RAM, for faster access. It is typically used for variables that are accessed very frequently, such as loop counters. Its scope and lifetime are the same as an automatic variable, but the address-of operator (&) cannot be applied to it, since a register may not have a memory address. Modern compilers are free to ignore this request and optimize register allocation on their own.

```
for (register int i = 0; i < 1000; i++)
{
    /* fast loop counter */
}
```

3. static

A static variable retains its value between successive calls to the function in which it is declared; it is initialized only once, and its lifetime extends over the entire execution of the program, even though its scope remains local to the block. If a static variable is declared at global level (outside all functions), its scope is restricted to the file in which it is declared, i.e., it becomes invisible to other files. The default initial value of a static variable is zero.

```
void counter(void)
{
    static int count = 0;   // initialized only once
    count++;
    printf("%d\n", count);
}
// Successive calls to counter() print 1, 2, 3, ...
```

4. extern

The extern storage class is used to declare a global variable or function that is defined in another file or later in the same file, making it visible/accessible across multiple source files of a program. An extern declaration does not allocate new memory; it simply refers to a variable that has been defined elsewhere. Its scope is global and its lifetime is the entire program run.

```
/* file2.c */
extern int total;   // refers to 'total' defined in another file
void show(void)
{
    printf("%d", total);
}
```

Summary of Storage Classes

Storage Class	Storage Area	Default Value	Scope	Lifetime
auto	RAM	Garbage	Local	Within the block
register	CPU register	Garbage	Local	Within the block
static	RAM	Zero	Local (or file, if global)	Entire program
extern	RAM	Zero	Global	Entire program

Category of Functions

Depending on whether a function accepts arguments and whether it returns a value, functions in C can be classified into the following categories:

1. No Argument and No Return Value

Such a function neither receives any data from the calling function nor sends any data back to it. It typically communicates with the rest of the program using global variables, or simply performs an independent task such as displaying output.

```
void greet(void)
{
    printf("Hello, welcome to C programming!");
}

int main(void)
{
    greet();      // no arguments passed, nothing returned
    return 0;
}
```

2. No Argument with Return Value

Such a function does not accept any input from the caller but computes and returns a value. This is often used when the function generates or reads a value on its own, for example, reading input from the keyboard or returning a fixed/computed constant.

```
int getValue(void)
{
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    return x;
}

int main(void)
{
    int n = getValue();   // receives a value, sends none
    return 0;
}
```

3. Argument with Return Value

Such a function receives one or more values from the calling function, processes them, and sends a result back. This is the most commonly used and most flexible category of function, since data flows in both directions.

```
int square(int n)
{
    return n * n;
}

int main(void)
{
    int result = square(7);   // 7 passed in, 49 returned
    printf("%d", result);
```

```
    return 0;
}
```

> ***Note:*** *A fourth possible combination, 'argument with no return value', also exists in practice — the function accepts input but performs its task (such as printing) without sending anything back, using a void return type.*

Recursion and Use of Memory Stack

Recursion is a programming technique in which a function calls itself, either directly or indirectly, in order to solve a problem by breaking it down into smaller sub-problems of the same type. A recursive function must always include:

- A base case (terminating condition): a condition under which the function stops calling itself and returns a value directly, preventing infinite recursion.
- A recursive case: a statement in which the function calls itself with a modified (usually smaller) argument, moving the problem closer to the base case.

Classic example — factorial of a number using recursion:

```
int factorial(int n)
{
    if (n == 0)          // base case
        return 1;
    else
        return n * factorial(n - 1);   // recursive case
}
```

Use of the Memory Stack in Recursion

Every time a function (recursive or not) is called, the compiler creates a new activation record (also called a stack frame) on a special area of memory called the call stack. This activation record stores the function's local variables, its parameters, and the return address, i.e., the point in the program to which control must return once the function finishes.

In recursion, each recursive call pushes a new stack frame on top of the call stack, since each call is treated as an entirely new invocation of the function, with its own separate copy of local variables and parameters. As calls continue until the base case is reached, the stack keeps growing. Once the base case is satisfied, the function starts returning; each return pops the topmost stack frame off the stack, and execution resumes at the return address stored in it. This continues until all the pushed frames have been popped and the very first call returns to the original caller.

Since the size of the call stack is limited, excessive recursion (for example, recursion without a proper base case, or with a very large number of recursive calls) exhausts the stack memory and causes a runtime error known as stack overflow.

Types of Recursion

- Direct recursion: A function calls itself directly from within its own body, e.g., function fact() calling fact() again.
- Indirect (mutual) recursion: A function A calls another function B, and function B in turn calls function A, so that the two functions call each other in a cycle.

- Tail recursion: The recursive call is the last statement executed in the function, with no pending operation left after it returns. Tail-recursive functions can often be optimized by the compiler into an iterative loop, saving stack space.
- Non-tail (head) recursion: The recursive call is followed by some additional operation (such as a multiplication or addition) that must be performed after the call returns, so the pending operations must be kept on the stack until the base case is reached.
- Linear recursion: The function makes only a single recursive call to itself during each invocation, e.g., computing factorial or sum of first n natural numbers.
- Tree (multiple) recursion: The function makes more than one recursive call to itself during a single invocation, causing the calls to branch out like a tree, e.g., the naive recursive computation of Fibonacci numbers, where fib(n) calls both fib(n-1) and fib(n-2).

```
// Tail recursion example
int sumTail(int n, int acc)
{
    if (n == 0)
        return acc;
    return sumTail(n - 1, acc + n);   // last operation performed
}

// Tree recursion example
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);   // two recursive calls
}
```

Multiple Choice Questions (15)

Each question carries one correct answer. The answer key is given at the end of this section.

1. Which of the following is the main reason for using functions in a C program?

(A) To increase the size of the source code
(B) To achieve modularity and reusability of code
(C) To make the program run only once
(D) To avoid using variables

2. A function prototype in C primarily helps the compiler to:

(A) Execute the function faster
(B) Allocate memory for global variables
(C) Perform type checking on function calls before the definition appears
(D) Convert the function into a macro

3. A variable declared inside a function, without any storage class keyword, has:

(A) Global scope and program lifetime
(B) Local scope and automatic lifetime
(C) Local scope and static lifetime
(D) File scope only

4. Which part of a function definition specifies the return type, name, and parameter list together?

(A) Function body
(B) Function header
(C) Return statement
(D) Function prototype only

5. In C, the actual parameters are:

(A) The variable names listed in the function definition
(B) The values or expressions passed during the function call
(C) Always pointers
(D) Declared using the extern keyword

6. In call by value, changes made to the formal parameter inside the function:

(A) Permanently change the actual parameter
(B) Have no effect on the actual parameter in the caller
(C) Cause a compilation error
(D) Delete the actual parameter

7. To achieve the effect of call by reference in C, a function must accept:

(A) An array of characters only
(B) The value of the variable directly
(C) The address of the variable, using a pointer parameter
(D) A static variable

8. How many values can a single return statement send back directly to the calling function?

(A) As many as the number of parameters
(B) Only one
(C) Exactly two
(D) Unlimited

9. Which storage class requests the compiler to store a variable in a CPU register for faster access?

(A) auto
(B) extern

(C) static
(D) register

10. A static local variable inside a function:

(A) Is reinitialized every time the function is called
(B) Loses its value once the function returns
(C) Retains its value across successive calls to the function
(D) Cannot be used inside a function

11. The extern storage class is mainly used to:

(A) Make a global variable or function accessible across multiple files
(B) Store a variable only in a CPU register
(C) Give a variable a local scope
(D) Delete unused variables

12. A function that neither takes any argument nor returns a value belongs to which category?

(A) Argument with return value
(B) No argument with return value
(C) No argument, no return value
(D) Argument with no return value

13. Which of the following is essential in every correctly written recursive function?

(A) A loop statement
(B) A base case that stops further recursive calls
(C) A global variable
(D) The register storage class

14. During recursive function calls, information about each call (local variables, parameters, return address) is stored in:

(A) The heap
(B) A stack frame (activation record) on the call stack
(C) A global array
(D) The data segment only

15. A recursive function in which the recursive call is the very last operation performed, with nothing left to compute after it returns, is called:

(A) Tree recursion
(B) Indirect recursion
(C) Tail recursion
(D) Infinite recursion

Answer Key

1. (B)	2. (C)	3. (B)
4. (B)	5. (B)	6. (B)
7. (C)	8. (B)	9. (D)
10. (C)	11. (A)	12. (C)
13. (B)	14. (B)	15. (C)

Broad / Long Answer Questions (10)

1. What is a function in C? Explain the need for functions in program development with suitable points.

2. What is a function prototype? Explain its syntax and significance with the help of an example.

3. Differentiate between the scope and lifetime of a variable. Explain local, global, and static variables with examples.

4. Explain the general structure of a function definition in C. Illustrate with a program that defines and calls a function.

5. Distinguish between call by value and call by reference in C, with suitable example programs for each.

6. Explain the rules governing the return statement in C. Can a function return more than one value? Justify your answer.

7. Describe the four storage classes available in C (auto, register, static, extern) along with their scope, lifetime, and default values.

8. Classify functions in C based on arguments and return values. Explain each category with a suitable example program.

9. What is recursion? Explain, with the help of the factorial function, how the memory stack is used during recursive function calls.

10. Explain the different types of recursion (direct, indirect, tail, non-tail, linear, and tree recursion) with suitable examples for each.