

UNIT 8

User Defined Function

(5 Hours) — C Programming (CACS151), BCA I/II

8.1 Introduction

A function is a self-contained, named block of code designed to perform a specific task. Instead of writing the same logic repeatedly, a programmer defines it once as a function and then calls it wherever that task needs to be performed. A user-defined function is a function that the programmer writes to meet the specific needs of a program, as opposed to a library function that is already provided by C.

Every C program consists of at least one function — `main()` — which is the starting point of execution. Larger programs are usually broken down into several smaller, manageable functions, an approach known as modular programming.

Why Use Functions?

- **Modularity:** Large problems are divided into smaller, logical, manageable units.
- **Reusability:** A function, once written, can be called multiple times from different parts of a program.
- **Reduced code size:** Avoids duplicating the same code repeatedly.
- **Easier debugging and maintenance:** Errors can be isolated and fixed within a specific function.
- **Abstraction:** The caller only needs to know what a function does, not how it does it internally.

8.2 Components of a Function

Using a user-defined function in a C program generally involves four related components working together.

Components of a User-Defined Function

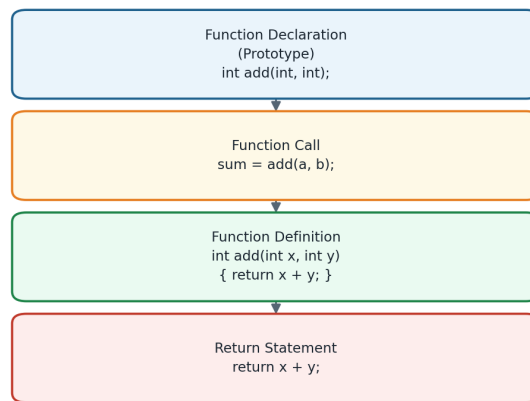


Fig 8.1 — Components of a user-defined function

Component	Description
Function Declaration (Prototype)	Tells the compiler the function's name, return type, and parameter types in advance — usually placed before main().
Function Call	The statement that invokes (executes) the function, optionally passing arguments to it.
Function Definition	The actual body of the function — the block of code that performs the task.
Return Statement	Sends a value back to the calling function and ends the function's execution (optional if the function is void).

Example 8.1 — A Complete Function in Action

```
function_demo.c
#include <stdio.h>
int add(int a, int b);      /* 1. Function declaration */
int main()
{
    int sum;
    sum = add(5, 3);        /* 2. Function call */
    printf("Sum = %d", sum);
    return 0;
}
int add(int a, int b)      /* 3. Function definition */
{
    return a + b;          /* 4. Return statement */
}
```

Note: A function prototype ends with a semicolon, while a function definition is followed by a body enclosed in curly braces { } — this is the key visual difference between the two.

8.3 Function Parameters

Parameters (also called arguments) allow data to be transferred between the calling function and the called function.

Term	Meaning
Formal Parameters	Variables listed in the function definition/declaration that receive the incoming values (e.g. int a, int b).
Actual Parameters	The real values or variables passed by the calling statement (e.g. add(5, 3)).

Based on parameters and return values, functions can be classified into four types:

Type	Description
No arguments, no return value	Function neither receives data nor sends any back — it performs a fixed task.
No arguments, with return value	Function computes and returns a value without needing input parameters.
With arguments, no return value	Function receives data to work with but does not send a result back.
With arguments, with return value	Function both receives input and returns a computed result — the most common and flexible form.

Example 8.2 — Function with Arguments and a Return Value

```
params_demo.c
#include <stdio.h>
int square(int n)    /* 'n' is a formal parameter */
{
    return n * n;
}
int main()
{
    int num = 6;
    printf("Square = %d", square(num)); /* 'num' is an actual parameter
*/
    return 0;
}
```

8.4 Library Function vs. User-Defined Function

Basis	Library Function	User-Defined Function
Definition	Pre-written, pre-compiled function provided by the C standard library	Function written by the programmer to solve a specific problem
Declaration	Declared in header files (e.g. <code>stdio.h</code> , <code>math.h</code>)	Declared/defined explicitly by the programmer in the source file
Modification	Cannot be modified by the programmer	Can be fully modified according to the program's needs
Examples	<code>printf()</code> , <code>scanf()</code> , <code>sqrt()</code> , <code>strlen()</code>	<code>add()</code> , <code>factorial()</code> , <code>isPrime()</code> , etc.
Storage	Stored in library files, linked at compile/link time	Stored as part of the program's own source code

Note: A user-defined function can freely call a library function inside its body, and vice versa — a library function like `qsort()` can even call a user-defined comparison function.

bcalink.com

8.5 Different Forms of Function

The four possible combinations of arguments and return values (introduced in 8.3) give rise to four distinct forms of user-defined functions. Each form suits a different programming need.

Form 1 — No Arguments, No Return Value

Example 8.3

```
form1.c
#include <stdio.h>
void greet()
{
    printf("Hello, welcome to C programming!\n");
}
int main()
{
    greet();
    return 0;
}
```

Form 2 — No Arguments, With Return Value

Example 8.4

```
form2.c
#include <stdio.h>
int getValue()
{
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    return x;
}
int main()
{
    int n = getValue();
    printf("You entered: %d", n);
    return 0;
}
```

Form 3 — With Arguments, No Return Value

Example 8.5

form3.c

```
#include <stdio.h>
void displaySum(int a, int b)
{
    printf("Sum = %d", a + b);
}
int main()
{
    displaySum(7, 4);
    return 0;
}
```

Form 4 — With Arguments, With Return Value

Example 8.6

form4.c

```
#include <stdio.h>
int multiply(int a, int b)
{
    return a * b;
}
int main()
{
    printf("Product = %d", multiply(6, 9));
    return 0;
}
```

8.6 Recursion

Recursion is a technique in which a function calls itself, either directly or indirectly, to solve a problem by breaking it down into smaller sub-problems of the same type. Every correct recursive function must have two essential parts:

- **Base Case:** A condition under which the function stops calling itself and returns a direct answer — without this, recursion never ends.
- **Recursive Case:** The part where the function calls itself with a smaller or simpler version of the original problem, moving it closer to the base case.

Recursion — factorial(4) Call Sequence

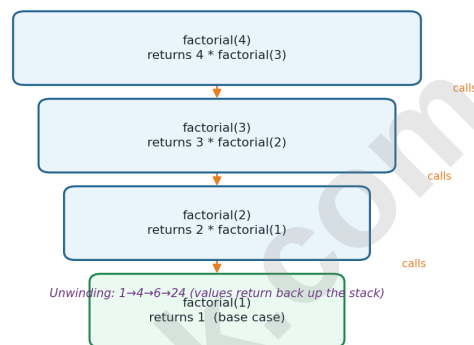


Fig 8.2 — Recursive call sequence for factorial(4)

Example 8.7 — Factorial Using Recursion

```
factorial.c
#include <stdio.h>
int factorial(int n)
{
    if (n == 0 || n == 1)    /* base case */
        return 1;
    else
        return n * factorial(n - 1); /* recursive case */
}
int main()
{
    printf("Factorial of 4 = %d", factorial(4));
    return 0;
}
```

How It Works

- $\text{factorial}(4) = 4 * \text{factorial}(3)$
- $\text{factorial}(3) = 3 * \text{factorial}(2)$
- $\text{factorial}(2) = 2 * \text{factorial}(1)$
- $\text{factorial}(1) = 1$ (base case reached — stops recursing)
- Values then multiply back up the chain: $1 \rightarrow 4 \rightarrow 6 \rightarrow 24$

Basis	Recursion	Iteration (Loop)
Mechanism	Function calls itself	Uses for/while/do-while to repeat
Memory use	Higher — each call uses stack memory	Lower — no extra function-call overhead
Speed	Generally slower due to function-call overhead	Generally faster
Code style	Often shorter and more elegant for problems like trees, factorial	Can be longer, but easier to trace/debug
Termination	Requires a clear base case	Requires a clear loop condition

Note: Without a properly reachable base case, a recursive function results in infinite recursion, which eventually exhausts the call stack and crashes the program with a stack overflow error.

8.7 Passing Array to Function

An array can be passed as an argument to a function. Unlike ordinary variables, C does not copy the entire array — instead, only the base address of the array is passed. This means the function operates on the original array, and any changes made inside the function directly affect the caller's array.

Point	Explanation
Passed by reference (in effect)	The array name decays to a pointer to its first element, so the function always works on the original data.
Size not automatically known	The array's size is lost when passed, so it is common practice to pass the array's length as a separate parameter.
Declaration in function	<code>int arr[]</code> or <code>int *arr</code> are equivalent ways to declare an array parameter.

Example 8.8 — Passing a 1-D Array to a Function

array_func.c

```
#include <stdio.h>
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}
void doubleValues(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        arr[i] = arr[i] * 2;    /* modifies the ORIGINAL array */
}
int main()
{
    int numbers[5] = {1, 2, 3, 4, 5};
    doubleValues(numbers, 5);
    printArray(numbers, 5);    /* prints: 2 4 6 8 10 */
    return 0;
}
```

Note: Because array names decay to pointers when passed, there is no true "call by value" for arrays in C — the function always has direct access to the original array elements.

8.8 Passing String to Function

Since a string in C is simply a character array terminated by a null character ('\0'), passing a string to a function follows the same rule as passing any array: the base address of the character array is passed, and the function can read or modify the string directly. The end of the string is detected by checking for the null terminator rather than needing a separate length parameter.

Example 8.9 — Passing a String to a Function

string_func.c

```
#include <stdio.h>
int stringLength(char str[])
{
    int count = 0;
    while (str[count] != '\0')    /* stop at null terminator */
        count++;
    return count;
}
void toUpperCase(char str[])
{
    int i = 0;
    while (str[i] != '\0')
    {
        if (str[i] >= 'a' && str[i] <= 'z')
            str[i] = str[i] - 32;    /* modifies ORIGINAL string */
        i++;
    }
}
int main()
{
    char name[20] = "bcalink";
    printf("Length = %d\n", stringLength(name));
    toUpperCase(name);
    printf("Upper case: %s", name);    /* prints: BCALINK */
    return 0;
}
```

Note: Standard library functions like `strlen()`, `strcpy()`, and `strcat()` (declared in `string.h`) are themselves examples of functions that accept strings as character-array parameters.

8.9 Accessing a Function: Call by Value & Call by Reference

C provides two distinct ways in which arguments can be passed to a function, differing in whether the function works on a copy of the data or on the original data itself.

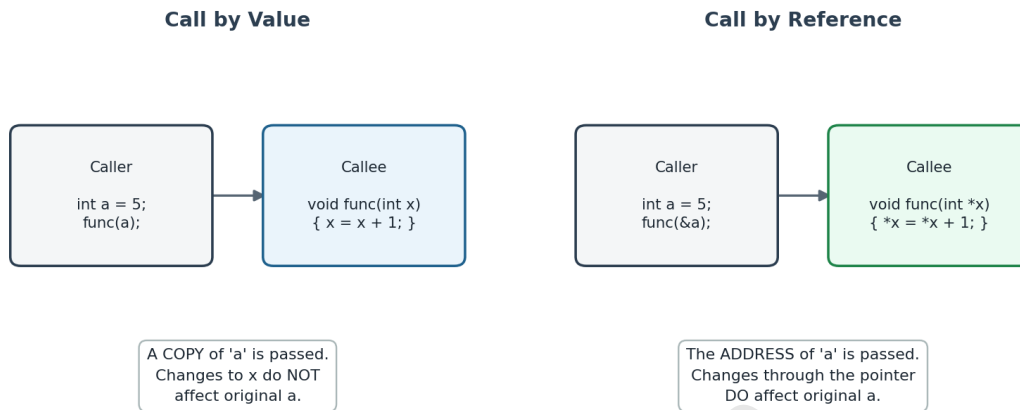


Fig 8.3 — Call by Value vs. Call by Reference

Call by Value

In call by value, a copy of the actual argument's value is passed to the function's formal parameter. Any changes made to the parameter inside the function affect only the copy — the original variable in the calling function remains unchanged.

Example 8.10 — Call by Value

```
call_by_value.c
#include <stdio.h>
void increment(int x)
{
    x = x + 1;          /* modifies only the local copy */
}
int main()
{
    int a = 10;
    increment(a);
    printf("Value of a = %d", a);  /* prints: 10 (unchanged) */
    return 0;
}
```

Call by Reference

In call by reference, the address of the actual argument is passed to the function using pointers. The function then accesses and modifies the original variable directly through its address, so changes made inside the function do reflect back in the calling function.

Example 8.11 — Call by Reference

```
call_by_reference.c
#include <stdio.h>
void increment(int *x)
{
    *x = *x + 1;          /* modifies the ORIGINAL variable via its address */
}
int main()
{
    int a = 10;
    increment(&a);        /* pass the address of a */
    printf("Value of a = %d", a); /* prints: 11 (changed) */
    return 0;
}
```

Basis	Call by Value	Call by Reference
What is passed	A copy of the actual argument's value	The memory address (location) of the actual argument
Effect on original	Original variable remains unchanged	Original variable is modified
Parameter type	Ordinary variable (int x)	Pointer variable (int *x)
Memory usage	Extra memory for the copy	No extra memory for a copy — more efficient for large data
Typical use	When the function should not alter caller's data	When the function must modify caller's data, or return multiple values

Note: Arrays and strings are always effectively passed by reference in C (since only their base address is passed), even though the syntax used to call the function looks like a normal call-by-value call.

8.10 Macros

A macro is a fragment of code that is given a name using the `#define` preprocessor directive. Whenever that name is used in the program, the preprocessor replaces it with the defined code before compilation begins — this is called macro expansion. Macros are not real functions; they involve simple text substitution, with no function-call overhead at runtime.

Object-like Macros (Symbolic Constants)

Example 8.12 — Object-like Macro

```
macro_const.c
#include <stdio.h>
#define PI 3.14159
#define MAX_STUDENTS 60
int main()
{
    printf("PI = %f\n", PI);
    printf("Max students allowed = %d", MAX_STUDENTS);
    return 0;
}
```

Function-like Macros

A macro can also accept arguments, making it behave somewhat like a function — but the substitution still happens purely as text replacement.

Example 8.13 — Function-like Macro

```
macro_func.c
#include <stdio.h>
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int main()
{
    printf("Square of 5 = %d\n", SQUARE(5));
    printf("Max of 7, 12 = %d", MAX(7, 12));
    return 0;
}
```

Note: Always enclose macro parameters (and the whole macro expression) in parentheses, e.g. `((x) * (x))` instead of `x * x` — otherwise expressions like `SQUARE(2 + 3)` expand incorrectly to `2 + 3 * 2 + 3` instead of `(2+3)*(2+3)`.

Macro vs. Function

Basis	Macro	Function
Processed by	Preprocessor (before compilation)	Compiler (during compilation) and called at runtime
Speed	Faster — no function-call overhead (pure text substitution)	Slightly slower due to call/return overhead
Type checking	None — purely textual substitution	Full type checking of arguments and return value
Code size	Increases code size if used repeatedly (code duplicated at each use)	Code size stays compact — one copy of code reused via calls
Debugging	Harder to debug since it does not appear as a distinct step	Easier to debug and trace step by step

bcalink.com

8.11 Storage Class

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

Storage Class	Keyword	Storage Location	Default Value	Scope	Lifetime
Automatic	auto	RAM (stack)	Garbage value	Local to the block	Exists only within the block/function
Register	register	CPU register (if available)	Garbage value	Local to the block	Exists only within the block/function
Static	static	RAM	Zero	Local to the block (but retains value)	Persists for the entire program run
External	extern	RAM	Zero	Global — accessible across multiple files	Persists for the entire program run

auto — Automatic Storage Class

This is the default storage class for all local variables declared inside a function or block. The keyword `auto` is rarely written explicitly since it is applied automatically.

```
auto_demo.c
void demo()
{
    auto int x = 5;    /* same as: int x = 5; */
}
```

register — Register Storage Class

Requests that the compiler store the variable in a CPU register instead of RAM, for faster access. This is typically used for variables accessed very frequently, such as loop counters. Modern compilers often optimize this automatically and may ignore the request if no register is available.

```

register_demo.c
void loopDemo()
{
    register int i;
    for (i = 0; i < 1000; i++)
    {
        /* fast-access loop counter */
    }
}

```

static — Static Storage Class

A static variable retains its value between successive function calls instead of being re-initialized every time. It is initialized only once, and its lifetime extends across the entire program run, even though its scope remains local to the block in which it is declared.

Example 8.14 — static vs. Ordinary Local Variable

```

static_demo.c
#include <stdio.h>
void counter()
{
    static int count = 0; /* initialized only ONCE */
    count++;
    printf("Count = %d\n", count);
}
int main()
{
    counter(); /* Count = 1 */
    counter(); /* Count = 2 */
    counter(); /* Count = 3 */
    return 0;
}

```

Explanation

- Because count is static, it is created and initialized to 0 only the first time counter() is called.
- On every later call, its previous value is remembered and simply incremented — producing 1, 2, 3 instead of always printing 1.

extern — External Storage Class

The `extern` keyword declares a variable or function that is defined in another file or elsewhere in the same file, allowing it to be shared across multiple source files in a large, multi-file program.

```
extern_demo.c
/* file1.c */
int total = 100;      /* actual definition */
/* file2.c */
extern int total;      /* refers to 'total' defined in file1.c */
void show()
{
    printf("Total = %d", total);
}
```

Note: Only one file should contain the actual definition of a global variable; all other files that need to use it should simply declare it using `extern`, without allocating new storage for it.

Key Takeaways

- ✓ A function is a named, reusable block of code consisting of a declaration, a call, a definition, and (optionally) a return statement.
- ✓ Functions may take no/some arguments and may or may not return a value, giving four possible forms.
- ✓ Recursion solves a problem by having a function call itself, always requiring a base case to terminate.
- ✓ Arrays and strings are passed to functions by their base address, so the called function operates on the original data.
- ✓ Call by value passes a copy of data (original unchanged); call by reference passes an address via pointers (original can be modified).
- ✓ Macros are expanded by the preprocessor as pure text substitution before compilation, unlike functions which are compiled and called at runtime.
- ✓ C has four storage classes — `auto`, `register`, `static`, and `extern` — which together determine a variable's scope, lifetime, and default value.

— End of Unit 8 —