#computer #college-studies IT #Sem2

By: Yashraj Maher

Syllabus

> ## Unit 1: Functions
>
> ### Types of Functions
>
> - Defining Function
> - Arguments
>   - Actual Parameter
>   - Formal Parameter
> - Function Prototype
> - Calling Function
> - Re-enter Function
> - Parameter Passing Mechanism
>   - Call by Value
>   - Call by Reference
> - Recursion
>
> ## Unit 2: Structures, Unions, and Pointers
>
> ### Structures
>
> - Introduction
> - Declaration and Initialization
> - Structure Members
> - Nested Structures
> - Array of Structures
> - Typedef Statement
> - Enumerated Datatypes
>
> ### Unions
>
> - Declaration
> - Differences Between Structures and Unions
>
> ### Pointers
>
> - Introduction
> - Address (`&`) and Indirection (`*`) Operators
> - Declaration and Initialization of Pointers
> - Pointer Expressions and Pointer Arithmetic
> - Pointer to Pointer
> - Dynamic Memory Allocation in C
>   - `malloc()`
>   - `calloc()`
>   - `free()`
>   - `realloc()`
>
> ## Unit 3: Storage Classes, Preprocessor, and File Handling
>
> ### Storage Classes
>
> - Scope, Visibility, and Lifetime of Variables

- Blocks and Files

Preprocessor and Directives

- Preprocessor
- Preprocessor Directives
- Conditional Compiler Directives
- Macros

File Handling

- Introduction to File Handling
- Opening and Closing Files
- Types of Files
  - Text Files
  - Binary Files
- File Operations
  - `getch()`
  - `put()`
  - `printf()`
  - `fscanf()`
  - `fread()`
  - `fwrite()`
- Writing and Reading Records
  - From Text Files
  - From Binary Files
- Advanced File Handling
  - Appending, Modifying, and Deleting Records from Random Access Files
  - Functions: `rewind()` , `flushall()` , `remove()` , `rename()`

Unit 1: Functions - The Building Blocks of C Programs

Introduction:

A function is a self-contained block of code designed to perform a specific task. Think of them as mini-programs within your larger program. Using functions makes your code more organized, easier to understand, and significantly reduces redundancy.

Why Functions?

Before diving into the details, let's understand *why* we use functions:

- **Modularity:** Breaking down a complex problem into smaller, manageable functions makes the overall program easier to design, debug, and maintain.
- **Reusability:** Once a function is defined, it can be called multiple times from different parts of the program, avoiding code duplication.
- **Readability:** Well-named functions make the code more self-documenting and easier to understand.
- **Abstraction:** Functions hide the implementation details of a task, allowing you to focus on *what* the function does rather than *how* it does it.

1.1 Defining a Function

A function definition specifies what the function does. It consists of two main parts: the *function declaration* (also known as the function header) and the *function body*.

Syntax:

```
return_type function_name(parameter_list) {
  // Function body - statements to be executed
  return value; // Optional, depending on return_type
}
```

- `return_type` : Specifies the data type of the value the function will return. If the function doesn't return a value, use `void` . Examples: `int` , `float` , `char` , `void` .
- `function_name` : A unique identifier for the function. Follows the same naming rules as variables (letters, numbers, underscore, starting with a letter or underscore).
- `parameter_list` : A comma-separated list of variables that the function accepts as input. Each parameter has a data type and a name. If the function doesn't accept any input, the list is empty.
- `function body` : The block of code enclosed in curly braces `{}` that contains the statements to be executed when the function is called.
- `return value;` : Used to return a value from the function to the calling code. The data type of the `return value` must match the `return_type` specified in the function declaration. If the `return_type` is `void` , the `return` statement is optional (and if present, should not have a value).

Example:

```
int add(int a, int b) {
  int sum = a + b;
  return sum;
}
```

This function, named `add` , takes two integer arguments (`a` and `b`), calculates their sum, and returns the sum as an integer.

1.2 Arguments

Arguments are the values passed to a function when it is called. They provide the function with the data it needs to perform its task. There are two key concepts related to arguments: actual parameters and formal parameters.

1.2.1 Actual Parameters

- **Definition:** Actual parameters (also called *arguments*) are the values that are passed to the function when it is *called* in the main program or another function.
- **Example:** In the following code:

```
int result = add(5, 3); // 5 and 3 are actual parameters
```

`5` and `3` are the actual parameters. These values are *copied* to the function's formal parameters.

1.2.2 Formal Parameters

- **Definition:** Formal parameters (also called *parameters*) are the variables declared in the function definition that receive the values of the actual parameters.
- **Example:** Referring back to the `add` function:

```
int add(int a, int b) { // a and b are formal parameters
  int sum = a + b;
  return sum;
}
```

`a` and `b` are the formal parameters. They act as placeholders for the values that will be passed in when the function is called.

1.3 Function Prototype

- **Definition:** A function prototype declares the function's name, return type, and parameter list *without* providing the function body.
- **Purpose:** The prototype informs the compiler about the function's existence and signature before the function is actually defined. This is crucial for allowing functions to call each other in any order. Without a prototype, the compiler might not know how to handle the function call.
- **Syntax:**

```
return_type function_name(parameter_list);
```

- **Example:**

```
int add(int a, int b); // Function prototype for add
```

Important Note for Turbo C++: Turbo C++ is less strict about requiring prototypes than more modern compilers. However, *always* use prototypes for good programming practice. It improves code readability and helps prevent errors.

Where to place prototypes: Function prototypes are typically placed at the beginning of the source file, before the `main()` function.

1.4 Calling a Function

- **Definition:** Calling a function means executing the code within the function body.
- **Syntax:**

```
function_name(argument_list);
```

- **Example:**

```
int result = add(10, 20); // Calling the add function
printf("The sum is: %d\n", result);
```

When the `add` function is called, the values `10` and `20` (actual parameters) are passed to the formal parameters `a` and `b` respectively. The function executes, calculates the sum, and returns the result, which is then stored in the `result` variable.

1.5 Re-enter Function

A re-enter function is a function that can be called recursively (we'll discuss recursion in section 1.7) or from multiple threads without causing issues. This means the function doesn't rely on static variables or global variables that might be modified by other calls to the function.

Key Characteristics:

- **No reliance on static variables:** Static variables retain their value between function calls. If a function relies on a static variable, it's not re-enterable because different calls might interfere with each other.
- **No reliance on global variables (ideally):** While not strictly forbidden, minimizing the use of global variables makes a function more re-enterable.
- **Local variables only:** The function should primarily use local variables, which are created and destroyed each time the function is called.

Why is this important? Re-enterable functions are essential for multi-threaded programming and for writing robust, reliable code.

1.6 Parameter Passing Mechanism

There are two primary ways to pass parameters to a function in C: call by value and call by reference.

1.6.1 Call by Value

- **Definition:** In call by value, a *copy* of the actual parameter's value is passed to the corresponding formal parameter.
- **Effect:** Any changes made to the formal parameter inside the function *do not* affect the original actual parameter.
- **Example:**

```
void modify(int x) {
  x = x + 10;
  printf("Inside modify: x = %d\n", x);
}

int main() {
  int num = 5;
  modify(num);
  printf("Inside main: num = %d\n", num);
  return 0;
}
```

Output:

```
Inside modify: x = 15
Inside main: num = 5
```

As you can see, the `modify` function changed the value of `x`, but the original `num` in `main` remained unchanged.

1.6.2 Call by Reference

- **Definition:** In call by reference, the function receives the *memory address* of the actual parameter.
- **Effect:** Any changes made to the formal parameter inside the function *directly affect* the original actual parameter.
- **Implementation in C:** C doesn't have a direct "call by reference" mechanism like some other languages. Instead, we use *pointers* to achieve the same effect.
- **Example:**

```c
void modify(int *x) {
  *x = *x + 10; // Dereference the pointer to access the value
  printf("Inside modify: *x = %d\n", *x);
}

int main() {
  int num = 5;
  modify(&num); // Pass the address of num
  printf("Inside main: num = %d\n", num);
  return 0;
}
```

Output:

```
Inside modify: *x = 15
Inside main: num = 15
```

In this case, `&num` passes the address of `num` to the `modify` function. The `*x` inside `modify` dereferences the pointer, allowing the function to directly modify the value stored at that memory address. Therefore, the change made to `*x` is reflected in `num` in `main`.

1.7 Recursion

- **Definition:** Recursion is a programming technique where a function calls itself within its own definition.
- **Key Components:**
  - **Base Case:** A condition that stops the recursion. Without a base case, the function would call itself infinitely, leading to a stack overflow.
  - **Recursive Step:** The part of the function where it calls itself with a modified input, moving closer to the base case.
- **Example: Calculating Factorial**

```c
int factorial(int n) {
  if (n == 0) { // Base case: factorial of 0 is 1
    return 1;
  } else { // Recursive step
    return n * factorial(n - 1);
  }
}

int main() {
  int num = 5;
  int result = factorial(num);
  printf("Factorial of %d is %d\n", num, result);
  return 0;
}
```

Explanation:

1. `factorial(5)` calls `5 * factorial(4)`
2. `factorial(4)` calls `4 * factorial(3)`
3. `factorial(3)` calls `3 * factorial(2)`
4. `factorial(2)` calls `2 * factorial(1)`
5. `factorial(1)` calls `1 * factorial(0)`
6. `factorial(0)` returns `1` (base case)
7. The values are then returned back up the chain: `1 * 1 = 1`, `2 * 1 = 2`, `3 * 2 = 6`, `4 * 6 = 24`, `5 * 24 = 120`.

When to use Recursion:

Recursion is particularly useful for problems that can be naturally broken down into smaller, self-similar subproblems, such as:

- Tree traversal
- Graph algorithms
- Mathematical functions like factorial and Fibonacci sequence.

Caution: Recursion can be less efficient than iterative solutions (using loops) due to the overhead of function calls. Be mindful of the potential for stack overflow if the recursion depth is too large.

Unit 2: Structures, Unions, and Pointers – Organizing and Manipulating Data in C

2.1 Structures

2.1.1 Introduction

A structure is a user-defined data type that allows you to group together variables of different data types under a single name. This is incredibly useful for representing real-world entities with multiple attributes. Think of a structure as a blueprint for creating custom data records.

Example: Representing a student with attributes like name, roll number, and marks.

2.1.2 Declaration and Initialization

Syntax:

```
struct structure_name {
  data_type member1;
  data_type member2;
  // ... more members
};
```

Example:

```
struct Student {
  char name[50];
  int roll_number;
  float marks;
};
```

Initialization:

There are several ways to initialize a structure:

- **Direct Initialization:**

```
struct Student s1 = {"Alice", 123, 85.5};
```

- **Member-by-Member Initialization:**

```
struct Student s2;
s2.name = "Bob";
s2.roll_number = 456;
s2.marks = 92.0;
```

- **Initialization using Designated Initializers (C99 and later - Turbo C++ may have limited support):**

```
struct Student s3 = {.name = "Charlie", .roll_number = 789, .marks = 78.0};
```

2.1.3 Structure Members

You access structure members using the dot (`.`) operator.

Example:

```c
struct Student s1 = {"Alice", 123, 85.5};
printf("Name: %s\n", s1.name);
printf("Roll Number: %d\n", s1.roll_number);
printf("Marks: %.2f\n", s1.marks);
```

2.1.4 Nested Structures

You can define structures within other structures. This allows you to represent more complex relationships.

Example:

```c
struct Address {
  char street[100];
  char city[50];
  int zip_code;
};

struct Employee {
  char name[50];
  int employee_id;
  struct Address address; // Nested structure
};
```

To access members of the nested structure:

```c
struct Employee emp;
strcpy(emp.address.city, "New York"); // Using strcpy to copy strings
```

2.1.5 Array of Structures

You can create an array of structures to store multiple instances of the same structure type.

Example:

```c
struct Student students[3];

// Initialize the array
strcpy(students[0].name, "Alice");
students[0].roll_number = 1;
students[0].marks = 90.0;

strcpy(students[1].name, "Bob");
students[1].roll_number = 2;
students[1].marks = 85.0;

// Accessing elements
printf("Name of student 1: %s\n", students[1].name);
```

2.1.6 Typedef Statement

The `typedef` statement creates an alias (a new name) for an existing data type. This can make your code more readable and maintainable.

Example:

```c
typedef struct Student {
  char name[50];
  int roll_number;
  float marks;
} StudentType; // StudentType is now an alias for the Student structure

StudentType s1; // Using the alias
strcpy(s1.name, "David");
```

2.1.7 Enumerated Datatypes

Enumerated datatypes (enums) define a set of named integer constants. They are useful for representing a fixed set of options.

Example:

```c
enum Day {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
};

int main() {
  enum Day today = MONDAY;
  printf("Today is %d\n", today); // Output: Today is 1
  return 0;
}
```

2.2 Unions

2.2.1 Declaration

A union is similar to a structure, but it allocates enough memory to hold only *one* of its members at a time. All members share the same memory location.

Syntax:

```c
union union_name {
  data_type member1;
  data_type member2;
  // ... more members
};
```

Example:

```c
union Data {
  int i;
  float f;
  char str[20];
};
```

2.2.2 Differences Between Structures and Unions

Feature	Structure	Union
Memory Allocation	Allocates memory for all members	Allocates memory for only one member at a time
Member Access	All members can be accessed independently	Only one member can be accessed at a time
Size	Size is the sum of the sizes of all members	Size is the size of the largest member
Use Cases	Representing entities with multiple attributes	Saving memory when only one of several data types is needed at a time

2.3 Pointers

2.3.1 Introduction

A pointer is a variable that stores the memory address of another variable. Pointers are fundamental to C programming and are used for dynamic memory allocation, passing arguments by reference, and manipulating data efficiently.

2.3.2 Address (`&`) and Indirection (`*`) Operators

- `&` **(Address-of Operator):** Returns the memory address of a variable.
- `*` **(Indirection Operator):** Accesses the value stored at the memory address pointed to by a pointer.

Example:

```
int num = 10;
int *ptr; // Declare a pointer to an integer

ptr = &num; // Assign the address of num to ptr

printf("Address of num: %p\n", &num); // %p is the format specifier for pointers
printf("Value of ptr: %p\n", ptr);   // ptr holds the address of num
printf("Value of num: %d\n", num);
printf("Value pointed to by ptr: %d\n", *ptr); // Dereferencing ptr
```

2.3.3 Declaration and Initialization of Pointers

Syntax:

```
data_type *pointer_name;
```

Example:

```
int *ptr; // Pointer to an integer
float *fptr; // Pointer to a float
char *cptr; // Pointer to a character
```

Initialization:

```
ptr = &num; // Assign the address of num to ptr
```

2.3.4 Pointer Expressions and Pointer Arithmetic

You can perform arithmetic operations on pointers, but the results are different from regular arithmetic. Pointer arithmetic is based on the size of the data type the pointer points to.

Example:

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr; // ptr points to the first element of arr

printf("Value of *ptr: %d\n", *ptr); // Output: 10

ptr++; // Increment ptr to point to the next element
printf("Value of *ptr: %d\n", *ptr); // Output: 20
```

In this example, `ptr++` increments the address stored in `ptr` by the size of an integer (typically 4 bytes).

2.3.5 Pointer to Pointer

A pointer to a pointer is a pointer that stores the address of another pointer.

Example:

```
int num = 10;
int *ptr1 = &num;
int **ptr2 = &ptr1; // ptr2 points to ptr1

printf("Value of num: %d\n", num);
printf("Value of *ptr1: %d\n", *ptr1);
printf("Value of **ptr2: %d\n", **ptr2);
```

2.3.6 Dynamic Memory Allocation in C

Dynamic memory allocation allows you to allocate memory during program execution. This is useful when you don't know the size of the data structure you need at compile time.

- `malloc()` : Allocates a block of memory of a specified size (in bytes). Returns a void pointer (`void *`) to the allocated memory.

```c
int *ptr = (int *)malloc(sizeof(int) * 10); // Allocate space for 10 integers
if (ptr == NULL) {
  printf("Memory allocation failed!\n");
  exit(1);
}
```

- `calloc()` : Allocates a block of memory and initializes all bytes to zero.

```c
float *ptr = (float *)calloc(5, sizeof(float)); // Allocate space for 5 floats, initialized to 0.0
if (ptr == NULL) {
  printf("Memory allocation failed!\n");
  exit(1);
}
```

- `free()` : Deallocates a block of memory that was previously allocated using `malloc()` or `calloc()` . **Important:** Always free dynamically allocated memory when you're finished with it to prevent memory leaks.

```c
free(ptr);
```

- `realloc()` : Resizes a previously allocated block of memory.

```c
int *ptr = (int *)malloc(sizeof(int) * 5);
ptr = (int *)realloc(ptr, sizeof(int) * 10); // Resize to hold 10 integers
if (ptr == NULL) {
  printf("Memory reallocation failed!\n");
  exit(1);
}
```

Unit 3: Storage Classes, Preprocessor, and File Handling – Managing Data Persistence and Code Flexibility in C

3.1 Storage Classes

3.1.1 Scope, Visibility, and Lifetime of Variables

Understanding how variables are stored and accessed is fundamental to writing correct and efficient C programs. Three key concepts define this:

- **Scope:** The region of the program where a variable is accessible.
- **Visibility:** Whether a variable can be seen or accessed from a particular part of the program. Closely related to scope.
- **Lifetime:** The duration for which a variable exists in memory.

3.1.2 Blocks and Files

- **Block:** A section of code enclosed in curly braces `{}` . Blocks define a local scope. Variables declared within a block are only visible within that block.
- **File:** The entire source code file. Variables declared outside of any function have file scope.

Storage Class Specifiers: C provides storage class specifiers to control the scope, visibility, and lifetime of variables. The main storage classes are:

- `auto` : (Default) Variables declared inside a function are automatically `auto` . They have local scope and their lifetime is limited to the block in which they are declared. They are automatically created when the block is entered and destroyed when the block is exited.
- `register` : Suggests to the compiler that the variable should be stored in a CPU register for faster access. However, the compiler is not obligated to honor this request. `register` variables have local scope and lifetime. You cannot use the address-of operator (`&`) with `register` variables.
- `static` : Has different meanings depending on where it's used:
  - **Inside a function:** A `static` variable retains its value between function calls. Its lifetime is the entire program execution, but its scope is still local to the function.

    - **Outside a function (file scope):** A `static` variable is visible only within the same file. It prevents other files from accessing the variable directly.
- `extern`: Declares a variable that is defined in another file. It tells the compiler that the variable exists elsewhere and allows you to access it. `extern` variables must be defined in another file.

3.2 Preprocessor and Directives

3.2.1 Preprocessor

The preprocessor is a program that runs *before* the compiler. It modifies the source code based on preprocessor directives. It performs tasks like including header files, defining macros, and conditional compilation.

3.2.2 Preprocessor Directives

Preprocessor directives are commands that start with a hash symbol (`#`).

- `#include`: Includes the contents of a header file into the current source file.

```
#include <stdio.h> // Standard input/output library
#include "myheader.h" // User-defined header file
```

- `#define`: Defines a macro. Macros are text substitutions performed by the preprocessor.

```
#define PI 3.14159
#define SQUARE(x) ((x) * (x)) // Macro for squaring a number
```

- `#undef`: Undefines a macro.

```
#undef PI
```

3.2.3 Conditional Compiler Directives

These directives allow you to compile different parts of your code based on certain conditions.

- `#if`: Starts a conditional block.
- `#else`: Provides an alternative block if the `#if` condition is false.
- `#elif`: Provides another condition to check if the previous `#if` or `#elif` conditions are false.
- `#endif`: Ends a conditional block.

```
#define DEBUG

#if DEBUG
  printf("Debugging information...\n");
#else
  // Code to be compiled only when DEBUG is not defined
#endif
```

3.2.4 Macros

Macros are powerful tools for code reuse and abstraction. However, be careful when using macros, as they can sometimes lead to unexpected behavior due to their simple text substitution nature. Parenthesize macro arguments to avoid precedence issues (as shown in the `SQUARE` macro example above).

3.3 File Handling

3.3.1 Introduction to File Handling

File handling allows your programs to interact with files on the disk. This is essential for reading data from files, writing data to files, and storing data persistently.

3.3.2 Opening and Closing Files

The `fopen()` function is used to open a file. The `fclose()` function is used to close a file.

Syntax:

```
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *fp);
```

- `filename`: The name of the file to open.
- `mode`: Specifies the mode in which to open the file:
  - `"r"`: Read mode (file must exist).
  - `"w"`: Write mode (creates a new file or overwrites an existing one).
  - `"a"`: Append mode (opens the file for appending data to the end).
  - `"rb"`: Read binary mode.
  - `"wb"`: Write binary mode.
  - `"ab"`: Append binary mode.
  - `"r+"`: Read and write mode.
  - `"w+"`: Read and write mode (creates or overwrites).
  - `"a+"`: Read and append mode.
- `FILE *fp`: A pointer to a `FILE` structure, which represents the opened file.

Example:

```
FILE *fp;
fp = fopen("mydata.txt", "w"); // Open mydata.txt in write mode

if (fp == NULL) {
  printf("Error opening file!\n");
  exit(1);
}

fclose(fp); // Close the file
```

3.3.3 Types of Files

- **Text Files:** Contain human-readable characters. Data is stored as ASCII or Unicode.
- **Binary Files:** Contain data in a raw, machine-readable format. They are more efficient for storing large amounts of data.

3.3.4 File Operations

Here's a breakdown of common file operations in C, with notes relevant to Turbo C++:

- `getch()`: Reads a character from the keyboard without echoing it to the screen. (Often used for pausing the program). This isn't directly a file operation, but often used in conjunction with file input/output.
- `put()`: Writes a character to the console. (Also not a direct file operation).
- `printf()`: Writes formatted output to the console. Can also be used with `fprintf()` to write to a file.
- `fscanf()`: Reads formatted input from a file.

```
FILE *fp;
int num;
fp = fopen("data.txt", "r");
fscanf(fp, "%d", &num); // Read an integer from the file
fclose(fp);
```

- `fread()`: Reads a block of data from a file.

```
FILE *fp;
int arr[10];
fp = fopen("data.bin", "rb");
fread(arr, sizeof(int), 10, fp); // Read 10 integers from the file
fclose(fp);
```

- `fwrite()`: Writes a block of data to a file.

```
FILE *fp;
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
fp = fopen("data.bin", "wb");
fwrite(arr, sizeof(int), 10, fp); // Write 10 integers to the file
fclose(fp);
```

3.3.5 Writing and Reading Records

- **From Text Files:** Use `fprintf()` to write formatted data and `fscanf()` to read formatted data. This is suitable for human-readable data.

- **From Binary Files:** Use `fwrite()` to write blocks of data and `fread()` to read blocks of data. This is more efficient for large amounts of data.

3.3.6 Advanced File Handling

- **Appending, Modifying, and Deleting Records from Random Access Files:** Random access files allow you to access specific records directly without reading through the entire file. This requires using `fseek()` to position the file pointer. Modifying and deleting records typically involve reading the file, making changes in memory, and then rewriting the entire file.
- **`rewind()`:** Resets the file pointer to the beginning of the file.

```
rewind(fp);
```

- **`flushall()`:** Flushes all output buffers to the disk. (Useful for ensuring that data is written to the file immediately).
- **`remove()`:** Deletes a file.

```
remove("mydata.txt");
```

- **`rename()`:** Renames a file.

```
rename("oldname.txt", "newname.txt");
```