

UNIT 2

Programming Technique

(5 Hours) — C Programming (CACS151)

2.1 Introduction to Programming Technique

A programming technique is a systematic method or strategy used by a programmer to design, organize, and write a solution to a problem in the form of a computer program. Rather than writing code in an ad-hoc manner, a good programming technique gives the developer a disciplined path from understanding a problem to producing a working, maintainable program.

Choosing the right technique affects how easy a program is to read, test, debug, modify, and extend. Over the decades several complementary techniques have emerged — top-down and bottom-up design, structured programming, modular programming, and the choice between iterative and recursive logic — each addressing a different aspect of how a solution is organized.

Why Programming Technique Matters

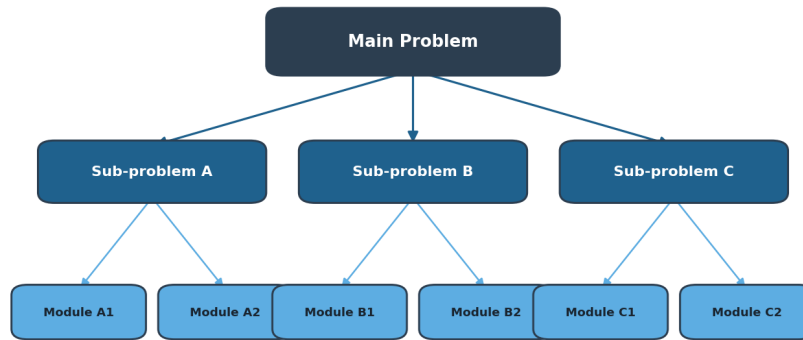
- ✓ Produces programs that are easier to understand and maintain
- ✓ Reduces the chance of logical errors during development
- ✓ Makes large problems manageable by breaking them into smaller parts
- ✓ Improves reusability of code across projects
- ✓ Supports teamwork by allowing different modules to be developed independently

2.2 Top-Down and Bottom-Up Approach

These are two contrasting strategies for decomposing a problem into smaller, solvable parts. Both aim to make a complex problem manageable, but they start from opposite ends of the abstraction hierarchy.

2.2.1 Top-Down Approach

In the top-down approach, the overall problem is first stated in its most general form. It is then progressively broken down into smaller sub-problems, and each sub-problem is further divided until every part is simple enough to be solved directly with executable statements. This successive refinement is often called “stepwise refinement.”



Executable statements / smallest solvable units

Decomposition proceeds from a broad, abstract problem down to concrete, executable modules.

Figure 2.1 — Top-down decomposition of a problem into modules

Example — Top-Down Design of a Payroll System

Main Problem: Generate employee payroll

Sub-problem A: Read employee data — Sub-problem B: Compute salary — Sub-problem C: Print payslip

Each sub-problem is refined further, e.g. Compute salary splits into Calculate basic pay and Calculate deductions, until each part is a single line of C code.

2.2.2 Bottom-Up Approach

The bottom-up approach works in the opposite direction. Programmers begin by identifying and building the most basic, low-level modules or routines first. These small, independent modules are then tested and combined into larger units, which are in turn combined until the complete system is assembled.

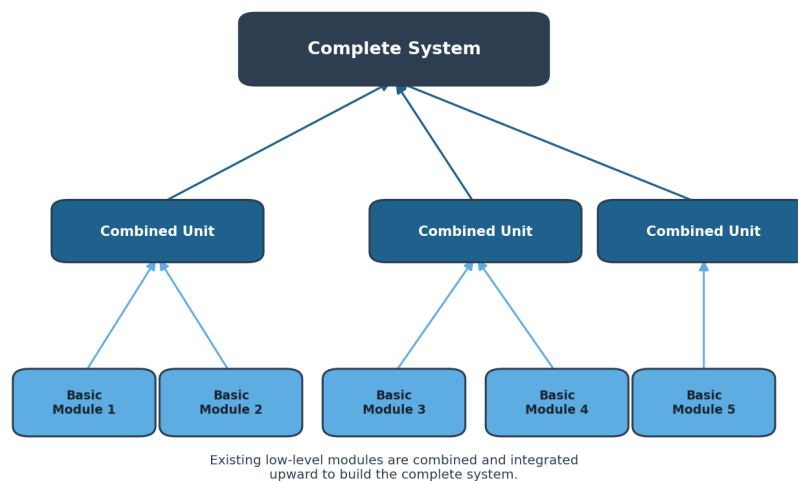


Figure 2.2 — Bottom-up integration of basic modules into a complete system

Note: Bottom-up design is common when reusable library routines already exist and can be assembled into a larger application, such as building a program from existing standard C library functions.

Aspect	Top-Down Approach	Bottom-Up Approach
Starting point	Overall problem (general)	Basic modules (specific)
Direction	General → specific	Specific → general
Design focus	Problem decomposition first	Module reuse and integration first
Best suited for	New systems designed from scratch	Systems built from existing components
Testing style	Tested after full decomposition	Modules tested individually, then combined

2.3 Cohesion and Coupling

Cohesion and coupling are two measures used to judge the quality of a modular design. They describe, respectively, how well the internal elements of a single module work together and how much one module depends on other modules.

2.3.1 Cohesion

Cohesion measures the degree to which the elements inside a single module belong together, all contributing to one well-defined task. A module with high cohesion performs one function and performs it well; a module with low cohesion performs several unrelated tasks bundled together for no strong reason.

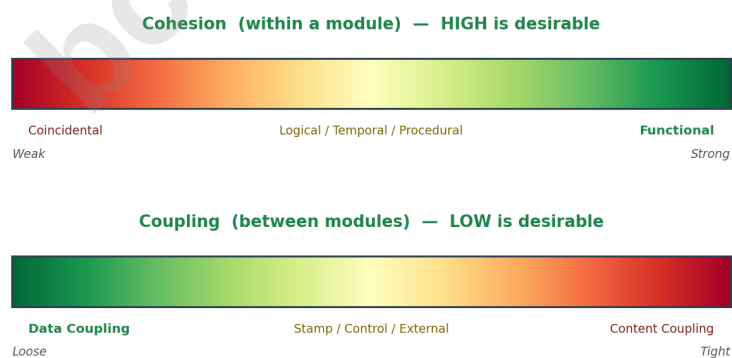
- Functional cohesion (highest) — every statement in the module contributes to a single, well-defined task.

- Procedural / Communicational cohesion — elements grouped because they follow a sequence or work on the same data.
- Temporal cohesion — elements grouped simply because they execute at the same time (e.g. an initialization routine).
- Coincidental cohesion (lowest) — elements are grouped with no meaningful relationship at all.

2.3.2 Coupling

Coupling measures the degree of interdependence between two separate modules — how much one module relies on the internal details of another. Unlike cohesion, low coupling is the desirable property, because loosely coupled modules can be changed, tested, and reused independently.

- Data coupling (lowest, best) — modules communicate only by passing simple data through parameters.
- Stamp coupling — a whole data structure is passed, even though the receiving module needs only part of it.
- Control coupling — one module passes a flag or control value that directs the internal logic of another.
- Content coupling (highest, worst) — one module directly accesses or modifies the internal data of another module.



Good design maximizes cohesion within modules while minimizing coupling between them.

Figure 2.3 — Desirable direction for cohesion (high) and coupling (low)

Cohesion vs Coupling — Key Rule

- ✓ Aim for HIGH cohesion within each module
- ✓ Aim for LOW coupling between modules
- ✓ This combination produces modules that are easy to understand, test, reuse, and modify

2.4 Structured Programming

Structured programming is a programming paradigm that improves the clarity, quality, and development time of a program by using only a small set of well-defined control structures, avoiding unrestricted jumps such as the goto statement. It was popularized following Edsger Dijkstra's argument that arbitrary jumps make programs hard to reason about and prove correct.

Any structured program can be built using just three fundamental control structures, all of which have a single entry point and a single exit point:

- Sequence — statements are executed one after another in the order they are written.
- Selection — a condition determines which of several alternative blocks of statements is executed (if-else, switch).
- Iteration (Repetition) — a block of statements is executed repeatedly while a condition holds (for, while, do-while).

Example — Structured C Code (No goto)

```
if (marks >= 40) {  
    printf("Pass");  
} else {  
    printf("Fail");  
}
```

Benefits of Structured Programming

- ✓ Programs are easier to read from top to bottom
- ✓ Debugging is simpler because control flow is predictable
- ✓ Reduces the chances of introducing hidden logical errors
- ✓ Encourages the use of modular, reusable blocks of code

2.5 Deterministic and Nondeterministic Technique

2.5.1 Deterministic Technique

A deterministic technique is one in which, for a given input, the algorithm always follows the same sequence of steps and always produces exactly the same, predictable output. Nearly all conventional C programs — sorting algorithms, arithmetic calculations, searching routines — are deterministic, since each step is completely determined by the previous state and the input.

2.5.2 Nondeterministic Technique

A nondeterministic technique is one in which, at some step, the algorithm may have more than one possible action available, and the choice of which action to take is not fixed by the algorithm itself. Conceptually, a nondeterministic algorithm is imagined to “guess” the correct choice at each branching point, or to explore multiple possibilities, often used as a theoretical tool for analyzing problem difficulty (e.g. NP problems) rather than for direct execution on real hardware.

Aspect	Deterministic Technique	Nondeterministic Technique
Output for given input	Always the same	May vary or is theoretical/guessed
Execution path	Fixed and predictable	Multiple possible paths
Real-world execution	Directly executable on real machines	Mostly a theoretical/conceptual model
Typical use	Everyday C programs	Complexity theory, problem analysis

Note: In practice, almost every C program you write in this course is deterministic; nondeterminism is primarily a concept from theoretical computer science used to classify how hard a problem is to solve.

2.6 Iterative and Recursive Logic

Both iteration and recursion are techniques for repeating a set of operations, but they achieve repetition in fundamentally different ways.

2.6.1 Iterative Logic

Iterative logic repeats a block of statements using a loop construct (for, while, or do-while) until a termination condition is met. The state of the computation — such as counters or accumulated results — is stored explicitly in variables and updated on each pass through the loop.

2.6.2 Recursive Logic

Recursive logic solves a problem by having a function call itself with a smaller version of the same problem, until it reaches a “base case” that can be solved directly without further calls. Each call waits for the result of the call it made, and results are combined as the calls return.

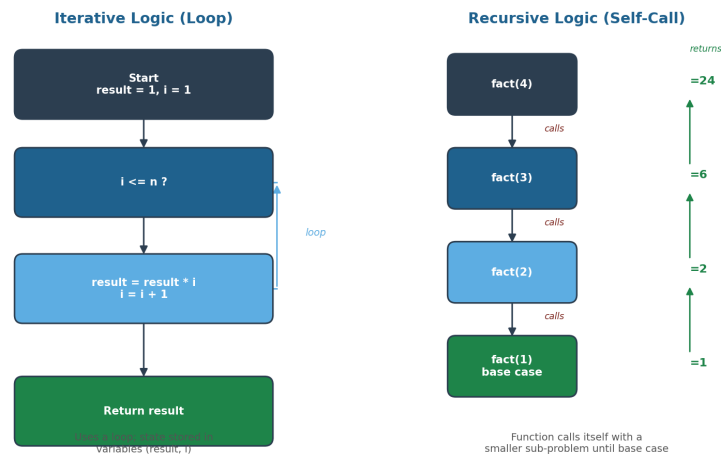


Figure 2.4 — Computing factorial(4) iteratively versus recursively

Example — Factorial: Iterative C Function

```

int factIter(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
    return result;
}

```

Example — Factorial: Recursive C Function

```

int factRec(int n) {
    if (n == 0) /* base case */
        return 1;
    return n * factRec(n - 1); /* recursive call */
}

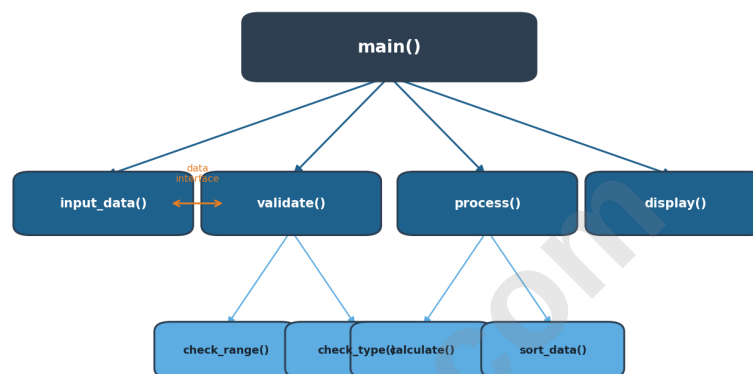
```

Aspect	Iterative Logic	Recursive Logic
Repetition mechanism	Loop (for / while / do-while)	Function calling itself
Memory usage	Generally lower (few variables)	Higher (each call uses stack space)
Termination	Loop condition becomes false	Base case is reached
Code style	Often longer, explicit state	Often shorter, more elegant for some problems
Typical use cases	Simple counting, straightforward repetition	Problems naturally defined recursively — trees, divide-and-conquer

Note: Every recursive solution can, in principle, be rewritten iteratively using an explicit stack, and vice versa — the choice is a matter of clarity, memory constraints, and how naturally the problem fits the

2.7 Modular Designing and Programming

Modular programming is the practice of dividing a program into separate, independent, interchangeable modules (in C, typically functions, and sets of functions organized into separate source and header files), each of which contains everything necessary to execute one well-defined aspect of the program's functionality.



Each module has a single well-defined purpose and communicates through clear parameter interfaces.

Figure 2.5 — Module hierarchy with clear interfaces between functions

A well-designed module should ideally satisfy the following properties, closely tied to the cohesion and coupling concepts discussed earlier:

- Single responsibility — the module performs one clearly defined task.
- Well-defined interface — the module communicates with others only through clearly specified parameters and return values.
- Information hiding — internal implementation details of a module are hidden from other modules.
- Reusability — a well-written module can be reused in other programs without modification.

Example — Modular Structure in C

input.c → function `getEmployeeData()`

process.c → function `calculateSalary()`

output.c → function `printPayslip()`

main.c → calls the above functions in sequence

Key Takeaways — Unit 2

- ✓ Programming technique is a disciplined strategy for turning a problem into working code
- ✓ Top-down decomposes a problem from general to specific; bottom-up assembles from existing basic modules upward
- ✓ High cohesion within a module and low coupling between modules together indicate good design
- ✓ Structured programming relies only on sequence, selection, and iteration — avoiding goto
- ✓ Deterministic techniques always give the same output for the same input; nondeterministic techniques allow multiple possible paths
- ✓ Iteration uses loops with explicit state; recursion uses self-calling functions with a base case
- ✓ Modular programming breaks a program into independent, reusable, well-interfaced modules

— *End of Unit 2* —

bcalink.com