

UNIT 3

Basic Concept of C

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

3.1 Introduction to C

C is a general-purpose, procedural, mid-level programming language developed for writing system software as well as application software. It combines the efficiency of assembly language with the readability of a high-level language, which is why it is often called a middle-level language. C gives the programmer direct control over memory through pointers, while still providing structured constructs such as loops, functions, and conditional statements.

C forms the foundation for many modern programming languages including C++, Java, C#, and Python, all of which borrow heavily from its syntax and core concepts. It remains the language of choice for operating systems, embedded systems, compilers, and performance-critical applications.

Note: Because C sits between low-level (assembly) and high-level languages, it is frequently described as a middle-level language.

3.2 History of C

The evolution of C can be traced through a sequence of earlier languages, each contributing ideas that shaped its final design.

Year	Language	Developed By	Remarks
1960	ALGOL 60	International Committee	Introduced the concept of structured programming
1963	CPL (Combined Programming Language)	Cambridge University	Complex; hard to implement on early hardware
1967	BCPL (Basic CPL)	Martin Richards	Simplified CPL; typeless language
1970	B Language	Ken Thompson (Bell Labs)	Simplified BCPL; used to write early Unix
1972	C Language	Dennis Ritchie (Bell Labs)	Added data types; became a full structured language
1978	K&R C	Kernighan & Ritchie	“The C Programming Language” book published; became the de facto standard

Year	Language	Developed By	Remarks
1989/90	ANSI C / ISO C (C89/C90)	ANSI Committee	First standardized version of C
1999	C99	ISO	Added inline functions, // comments, new data types
2011	C11	ISO	Added multithreading support, improved Unicode support

Note: C was originally developed by Dennis Ritchie at Bell Laboratories in 1972 to re-implement the Unix operating system, which had earlier been written in assembly language and B.

3.3 Features of C

C's enduring popularity stems from a set of features that make it both powerful and portable:

- **Simple and Efficient:** Uses a small set of keywords and a concise syntax, producing fast, compact executable code.
- **Middle-Level Language:** Supports low-level operations (bitwise manipulation, pointers) as well as high-level constructs (functions, structures).
- **Portability:** A C program written on one platform can be compiled and run on another with little or no modification.
- **Structured Language:** Programs are divided into functions/blocks, making code modular, readable, and easy to debug.
- **Rich Library Support:** A large collection of built-in library functions handles input/output, string handling, math operations, and memory management.
- **Pointer Support:** Direct access to memory addresses allows efficient handling of arrays, strings, and dynamic memory.
- **Extensibility:** Programmers can add their own functions and libraries to extend the language's capabilities.
- **Speed of Compilation and Execution:** C compiles quickly and produces highly efficient machine code, close to assembly-level performance.
- **Case-Sensitive:** Uppercase and lowercase letters are treated as distinct identifiers.

3.4 Advantages and Disadvantages of C

Advantages

- Highly portable across different machines and operating systems.
- Produces fast and efficient executable programs due to its low-level access.

- Rich set of built-in operators and library functions.
- Forms the base for learning other high-level languages such as C++ and Java.
- Supports dynamic memory allocation, giving flexible control over memory usage.
- Widely used to build operating systems, compilers, and embedded systems.

Disadvantages

- Lacks built-in support for Object-Oriented Programming (no classes, inheritance, etc.).
- No automatic garbage collection — the programmer must manage memory manually.
- No strict runtime type checking, which can lead to hard-to-find bugs.
- Absence of a built-in exception-handling mechanism (unlike try/catch in other languages).
- Pointer misuse can easily cause memory leaks or program crashes.

bcalink.com

3.5 Structure of a C Program

Every C program follows a well-defined layout consisting of several sections. Not all sections are compulsory in every program, but the `main()` function is mandatory since program execution always begins there.

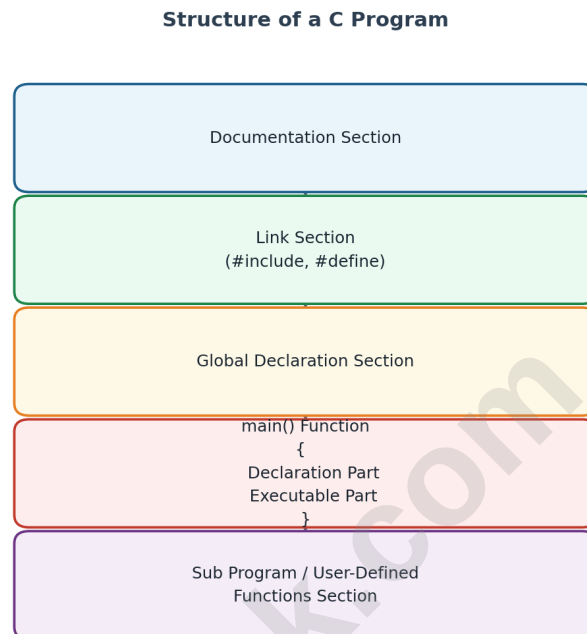


Fig 3.1 — General structure of a C program

- **Documentation Section:** Comments describing the program's purpose, author, and date.
- **Link Section:** Contains preprocessor directives such as `#include` and `#define` that link header files and define constants.
- **Global Declaration Section:** Variables and functions declared outside `main()` that are accessible to all functions in the program.
- **`main()` Function:** The entry point of execution, containing a declaration part (variable declarations) and an executable part (statements).
- **Sub-Program Section:** User-defined functions that are called from `main()` or from other functions.

Example 3.1 — A Simple C Program

```
/* Program to display a welcome message */
```

```
welcome.c
#include <stdio.h>
int main()
{
    printf("Welcome to C Programming!");
    return 0;
}
```

Explanation

- `#include <stdio.h>` links the Standard Input/Output header file.
- `int main()` is the mandatory entry point of the program.
- `printf()` is a library function that displays output on the screen.
- `return 0;` indicates successful termination of the program.

3.6 Compiling Process

Converting C source code into an executable program involves several distinct stages, each handled by a separate tool within the compiler toolchain.

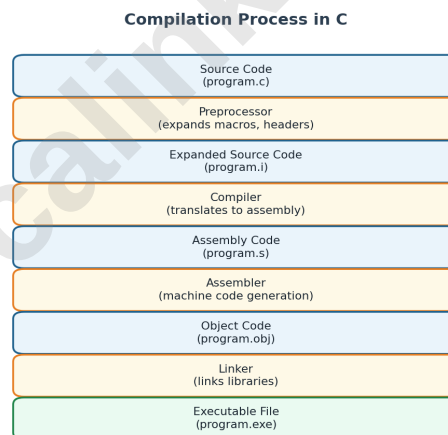


Fig 3.2 — Stages in the compilation of a C program

Stage	Description
Preprocessing	The preprocessor scans the source file, expanding macros and including header files. Output: expanded source (.i).
Compilation	The compiler translates the preprocessed code into assembly language specific to the target processor. Output: assembly file (.s).
Assembling	The assembler converts the assembly code into machine-readable object code. Output: object file (.obj / .o).
Linking	The linker combines the object code with library functions and other object files to produce a single executable. Output: executable file (.exe).

Note: If a program uses multiple source files, each file is preprocessed, compiled, and assembled separately, and the linker combines all resulting object files into one final executable.

3.7 C Preprocessor and Header Files

The C Preprocessor is a program that runs automatically before actual compilation begins. It processes all lines that start with the # symbol, known as preprocessor directives, and modifies the source code accordingly before it reaches the compiler.

Header Files

A header file is a file with a .h extension that contains declarations of library functions, macros, and constants. Including a header file makes those declarations available to the program via the #include directive.

Header File	Purpose
stdio.h	Standard input/output functions (printf, scanf, etc.)
stdlib.h	General utilities: memory allocation, conversions, exit()
string.h	String manipulation functions (strlen, strcpy, strcmp)
math.h	Mathematical functions (sqrt, pow, sin, cos)
conio.h	Console input/output (getch, clrscr) — compiler-specific
ctype.h	Character-testing and conversion functions (isalpha, toupper)

Example 3.2 — Using a Header File

```
header_demo.c
#include <stdio.h>
#include <math.h>
int main()
{
    double result = sqrt(25.0);
    printf("Square root: %.2f", result);
    return 0;
}
```

3.8 Library Functions

A library function is a pre-written, pre-compiled piece of code that performs a specific, commonly needed task. C provides a rich standard library so that programmers do not need to

write routine operations — such as printing output or computing a square root — from scratch.

- Library functions are declared in header files and defined in pre-compiled library files.
- To use a library function, its corresponding header file must be included with `#include`.
- Examples: `printf()`, `scanf()`, `strlen()`, `strcpy()`, `sqrt()`, `malloc()`, `rand()`.

Note: Library functions save development time, reduce errors, and are typically highly optimized compared to code a beginner might write manually.

bcalink.com

3.9 Character Set

The character set is the collection of valid characters that C recognizes for writing source code — including letters, digits, special symbols, and white-space characters.

Category	Characters
Letters	A–Z, a–z
Digits	0–9
Special Symbols	~ ! @ # % ^ & * () _ - + = { } [] : ; " ' < > , . ? / \
White Spaces	Blank space, horizontal tab, newline, carriage return, form feed

3.10 Comments

Comments are non-executable lines used to explain code. The compiler ignores them entirely; they exist purely to improve readability and documentation for programmers.

Type	Syntax	Example
Single-line comment	// comment text	// This calculates the area
Multi-line comment	/* comment text */	/* This program calculates area */

Note: Multi-line comments (/ ... */) cannot be nested inside one another in standard C; attempting to do so causes a compilation error.*

3.11 Tokens and Their Types

A token is the smallest individual unit in a C program that is meaningful to the compiler. Every C program is essentially a sequence of tokens separated by white space or punctuation.

Classification of Tokens in C

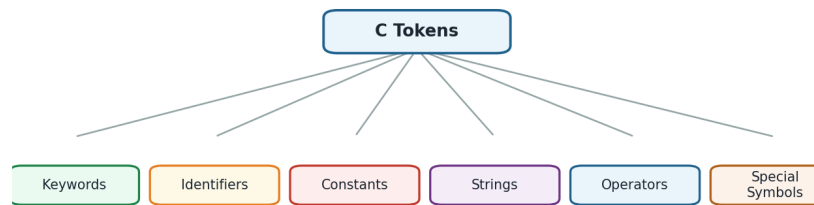


Fig 3.3 — Classification of tokens in C

Keywords

Keywords are reserved words that have a predefined meaning in C and cannot be used as identifiers (variable names, function names, etc.). C has 32 keywords in its original ANSI standard, including:

int, float, char, double, if, else, for, while, do, switch, case, break, continue, return, void, struct, union, typedef, static, const, sizeof, goto.

Identifiers

Identifiers are user-defined names given to variables, functions, arrays, and other program elements. Rules for naming identifiers:

- Must begin with a letter (A–Z, a–z) or an underscore (_).
- Can be followed by letters, digits, or underscores.
- Cannot contain spaces or special symbols such as @, #, %.
- Cannot be the same as a reserved keyword.
- Are case-sensitive (Total and total are different identifiers).

Constants

Constants are fixed values that do not change during program execution. They are classified as:

- Integer constants: 10, -25, 0, 0x1A (hexadecimal), 017 (octal)
- Real (floating-point) constants: 3.14, -0.001, 2.5e3
- Character constants: 'A', '5', '\$' — a single character enclosed in single quotes
- Enumeration constants: named integer constants defined using the enum keyword

Strings

A string is a sequence of characters enclosed within double quotes, terminated internally by a null character ('\0'). Example: "Hello, World!"

Operators

Operators are symbols that instruct the compiler to perform specific mathematical, relational, or logical operations on operands.

Category	Operators	Example
Arithmetic	+ - * / %	a + b, a % b
Relational	== != > < >= <=	a > b
Logical	&& !	(a > 0) && (b > 0)
Assignment	= += -= *= /=	a += 5
Increment/Decrement	++ --	a++, --b
Bitwise	& ^ ~ << >>	a & b, a << 2

Special Symbols

Special symbols such as brackets [], parentheses (), braces { }, comma (,), semicolon (;), colon (:), and asterisk (*) serve structural or syntactic purposes — for example, semicolons terminate statements and braces define blocks of code.

3.12 Data Types

A data type specifies the kind of value a variable can hold and how much memory is allocated for it. C provides four broad categories of data types.

Category	Examples	Description
Primary (Basic)	int, char, float, double, void	Fundamental built-in types provided directly by C
Derived	array, pointer, function	Built from primary types to represent more complex data
User-Defined	struct, union, enum, typedef	Created by the programmer to model custom data structures
Void	void	Represents “no value” — used for functions that return nothing or generic pointers

Primary Data Types — Size and Range (typical, 32-bit systems)

Data Type	Size (bytes)	Format Specifier	Typical Range
char	1	%c	-128 to 127
int	2 or 4	%d	-32,768 to 32,767 (2 bytes)
float	4	%f	3.4E-38 to 3.4E+38 (6 digit precision)
double	8	%lf	1.7E-308 to 1.7E+308 (15 digit precision)
void	0	—	No value / empty type

Note: Data type sizes can vary depending on the compiler and the underlying hardware architecture (16-bit, 32-bit, or 64-bit systems). The sizeof() operator can be used to check exact sizes on a given system.

Example 3.3 — Declaring Variables of Different Data Types

datatypes.c

```
#include <stdio.h>
int main()
{
    int age = 20;
    float price = 99.50;
    char grade = 'A';
    double pi = 3.141592;
    printf("Age: %d\n", age);
    printf("Price: %.2f\n", price);
    printf("Grade: %c\n", grade);
    printf("Pi: %lf\n", pi);
    return 0;
}
```

3.13 Escape Sequences

An escape sequence is a combination of a backslash (\) followed by a character, used to represent characters that cannot be typed directly or that have special meaning — such as a newline or a tab.

Escape Sequence	Meaning
\n	New line
\t	Horizontal tab
\b	Backspace
\r	Carriage return
\\	Backslash character
\'	Single quote character
\"	Double quote character
\0	Null character (string terminator)
\a	Alert / bell sound
\f	Form feed

Example 3.4 — Using Escape Sequences

```

escape.c
#include <stdio.h>
int main()
{
    printf("Name:\tJohn\n");
    printf("Score:\t95\n");
    printf("He said, \"Well done!\");
    return 0;
}

```

Output

Name: John

Score: 95

He said, "Well done!"

3.14 Preprocessor Directives

Preprocessor directives are instructions that begin with the # symbol and are processed by the preprocessor before compilation. They do not end with a semicolon.

Directive	Purpose	Example
#include	Includes a header file into the program	#include <stdio.h>
#define	Defines a macro or symbolic constant	#define PI 3.14159
#undef	Removes a previously defined macro	#undef PI
#ifdef	Compiles code only if a macro is defined	#ifdef DEBUG
#ifndef	Compiles code only if a macro is NOT defined	#ifndef MAX
#if / #else / #endif	Conditional compilation based on an expression	#if VERSION > 2
#pragma	Gives special instructions to the compiler	#pragma warning(disable:4996)

Example 3.5 — Using #define for a Symbolic Constant

macro.c

```
#include <stdio.h>
#define PI 3.14159
#define AREA(r) (PI * r * r)
int main()
{
    float radius = 5.0;
    printf("Area = %.2f", AREA(radius));
    return 0;
}
```

Explanation

- PI is replaced by 3.14159 wherever it appears, before compilation.
- AREA(r) is a macro with an argument — the preprocessor substitutes the expression (PI * r * r) directly into the code.

Key Takeaways

- ✓ C is a middle-level, structured, and highly portable programming language developed by Dennis Ritchie in 1972.
- ✓ A C program follows a fixed structure: documentation, link, global declarations, main(), and sub-programs.
- ✓ The compilation process moves through preprocessing, compilation, assembling, and linking to produce an executable.
- ✓ The preprocessor handles directives like #include and #define before actual compilation begins.
- ✓ Tokens — keywords, identifiers, constants, strings, operators, and special symbols — are the basic building blocks of a C program.
- ✓ C supports four data type categories: primary, derived, user-defined, and void.
- ✓ Escape sequences represent non-printable or special characters within strings and character constants.

— *End of Unit 3* —