UE23CS151B: Problem Solving with C
Lecture Slides - Slot #7, #8

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

UE23CS151B: Problem Solving with C - Syllabus

Unit 1 : Problem Solving Fundamentals - 14 Hours - 18 Slots

Introduction to Programming, Salient Features of 'C', Program Structure, Variables, Data Types & range of values, Operators and Expressions, Control Structures, Input/ Output Functions, Language Specifications - Behaviors, Single character input and output

Unit 2 : Counting, Sorting and Searching - 14 Hours - 18 Slots

Arrays – 1D and 2D, Pointers, Arrays with Pointers, Functions, Storage classes, Recursion, Enums, Bit Fields

Unit 3 : Text Processing and User-Defined Types 14 Hours - 18 Slots

Strings, String Manipulation Functions & Errors, Dynamic Memory Management functions & errors, Structures, #pragma, Unions, Lists, Priority Queue

Unit 4 : File Handling and Portable Programming - 14 Hours - 18 Slots

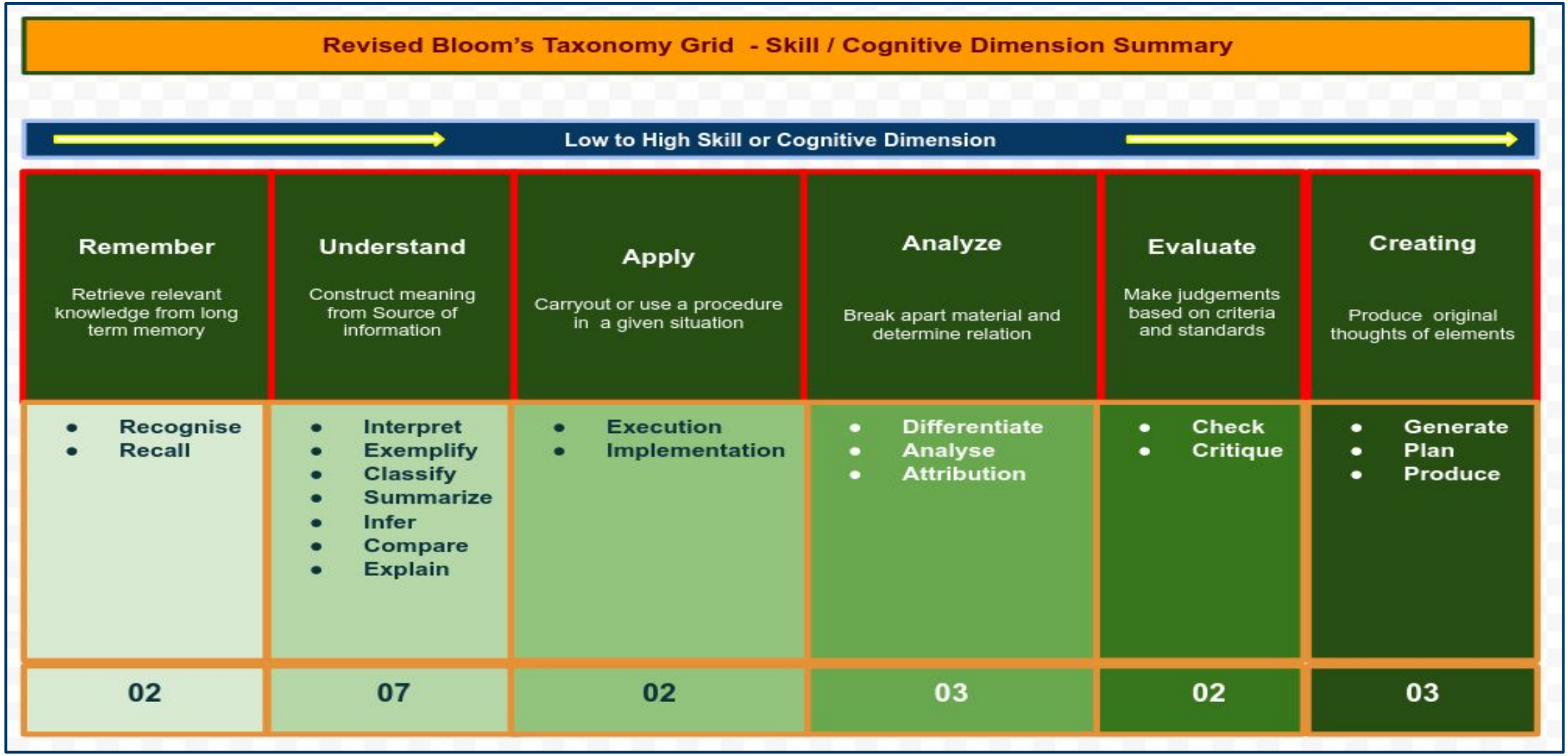
File IO using redirection, File Handling functions, Callback, Searching, Sorting, Preprocessor Directives, Conditional Compilation and Code Review, Debugging with GDB

UE23CS151B: Problem Solving with C - Course Objectives

The objective(s) of this course is to make students

- **CObj 1: Acquire knowledge on how to solve relevant and logical problems using computing machine**
- **CObj 2: Map algorithmic solutions to relevant features of C programming language constructs**
- **CObj 3: Gain knowledge about C constructs and its associated ecosystem**
- **CObj 4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviours**

Bloom's Taxonomy



Revised Bloom's Taxonomy Grid - Skill / Cognitive Dimension Summary

Low to High Skill or Cognitive Dimension

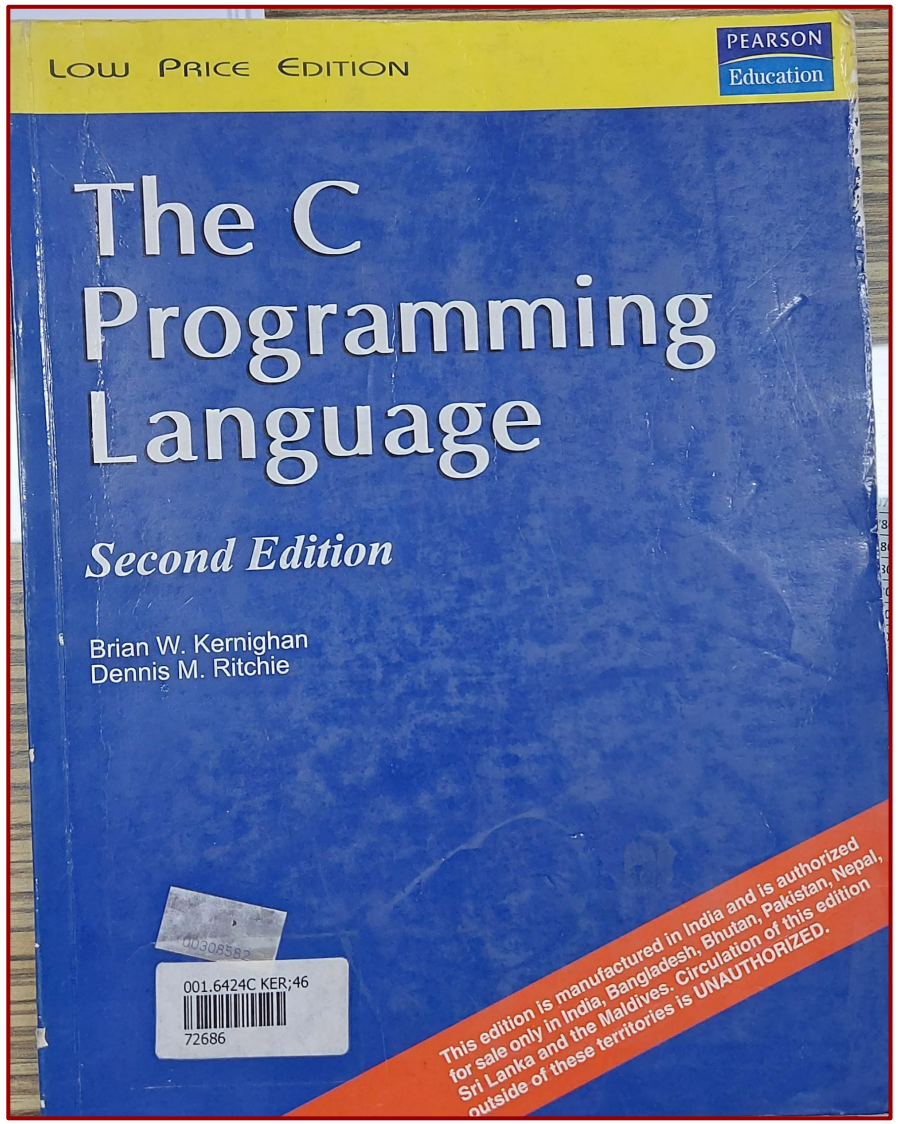
Remember	Understand	Apply	Analyze	Evaluate	Creating
Retrieve relevant knowledge from long term memory	Construct meaning from Source of information	Carryout or use a procedure in a given situation	Break apart material and determine relation	Make judgements based on criteria and standards	Produce original thoughts of elements
• Recognise • Recall	• Interpret • Exemplify • Classify • Summarize • Infer • Compare • Explain	• Execution • Implementation	• Differentiate • Analyse • Attribution	• Check • Critique	• Generate • Plan • Produce
02	07	02	03	02	03

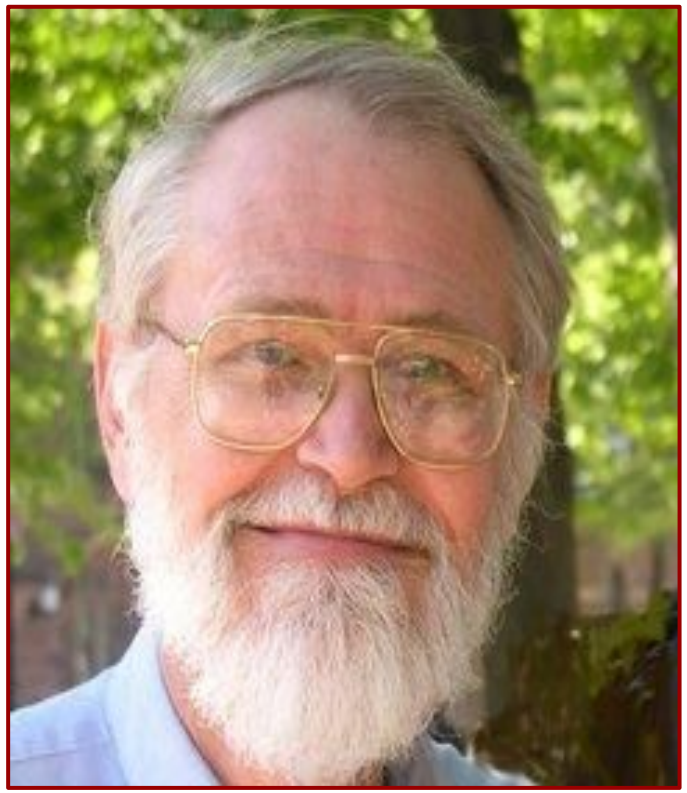
UE23CS151B: Problem Solving with C - Course Outcomes

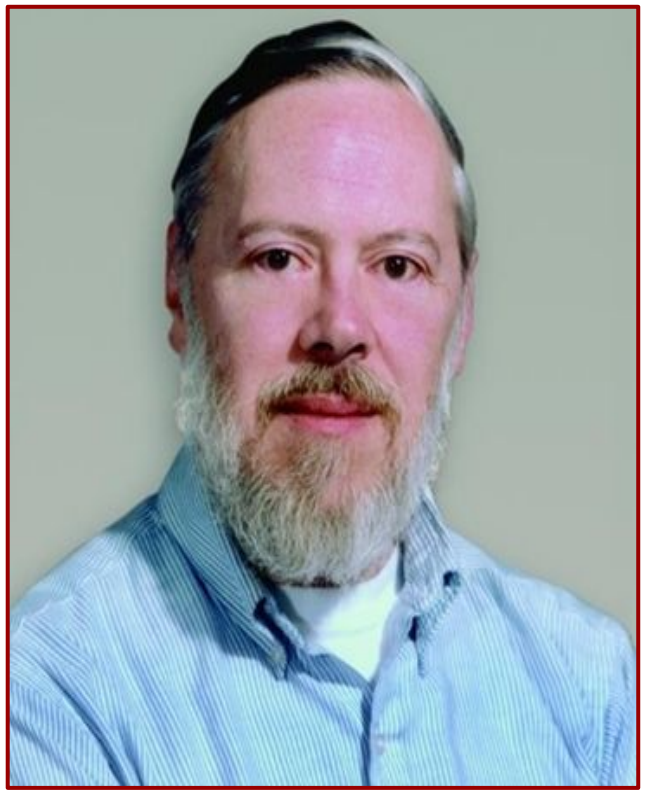
At the end of the course, the student will be able to

- CO1: Understand and Apply algorithmic solutions to basic problems using appropriate C constructs
- CO1: Understand and Apply Sorting and Searching techniques
- CO3: Understand, Analyse and Apply text processing and string manipulation methods using Arrays, Pointers and functions
- CO4: Understand prioritized scheduling and Implement the same using C structures using advanced C constructs, preprocessor directives and conditional compilation

UE22CS151B: Problem Solving with C Text Book







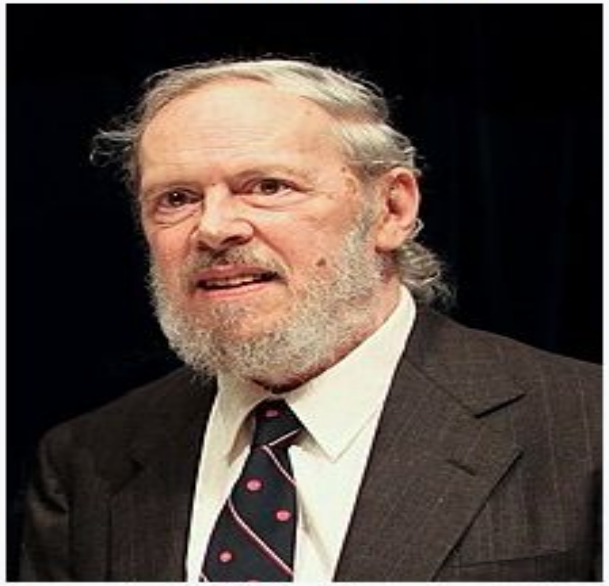
UE22CS151B: About Text Book Authors

Brian Kernighan

Brian Kernighan at Bell Labs in 2012

Born	Brian Wilson Kernighan 1942 (age 79–80)[1] Toronto, Ontario, Canada
Nationality	Canadian
Citizenship	Canada
Alma mater	University of Toronto (BASc) Princeton University (PhD)

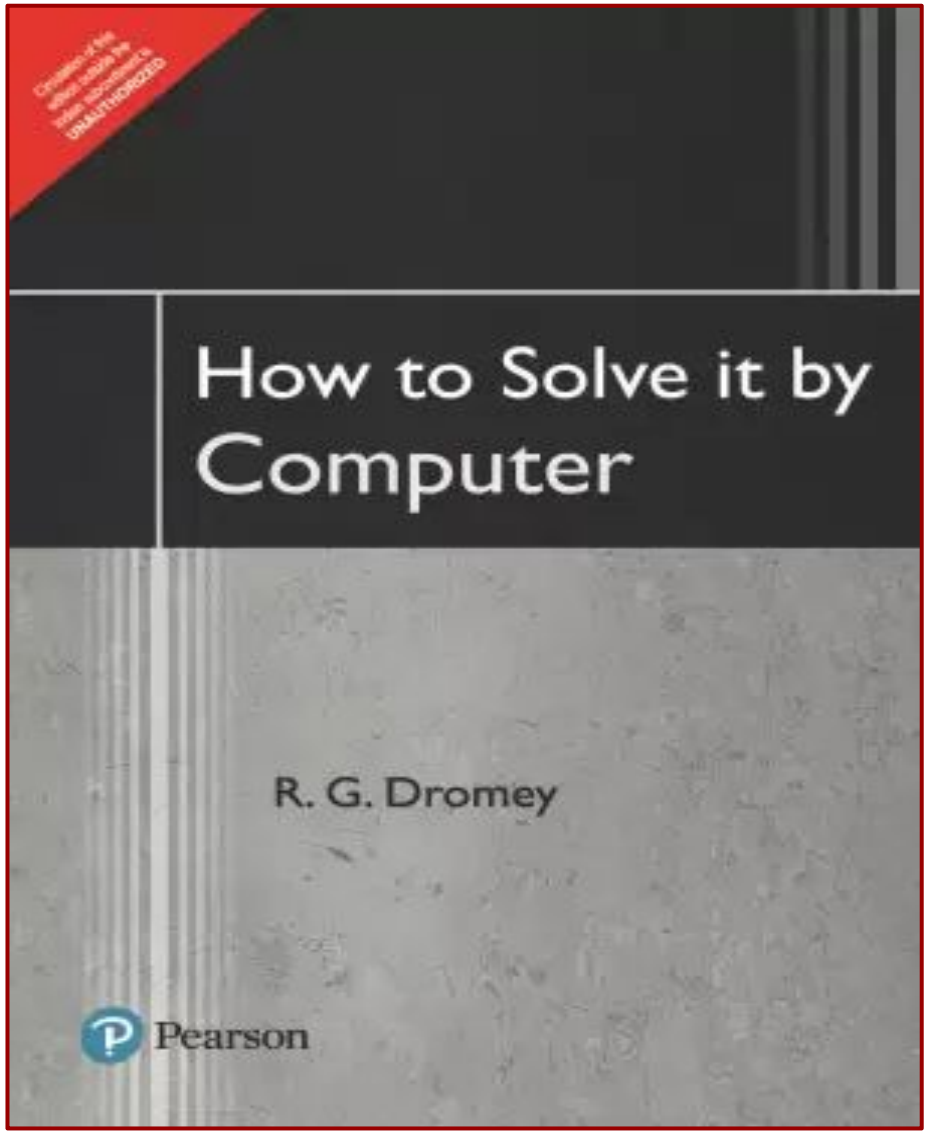
Dennis Ritchie

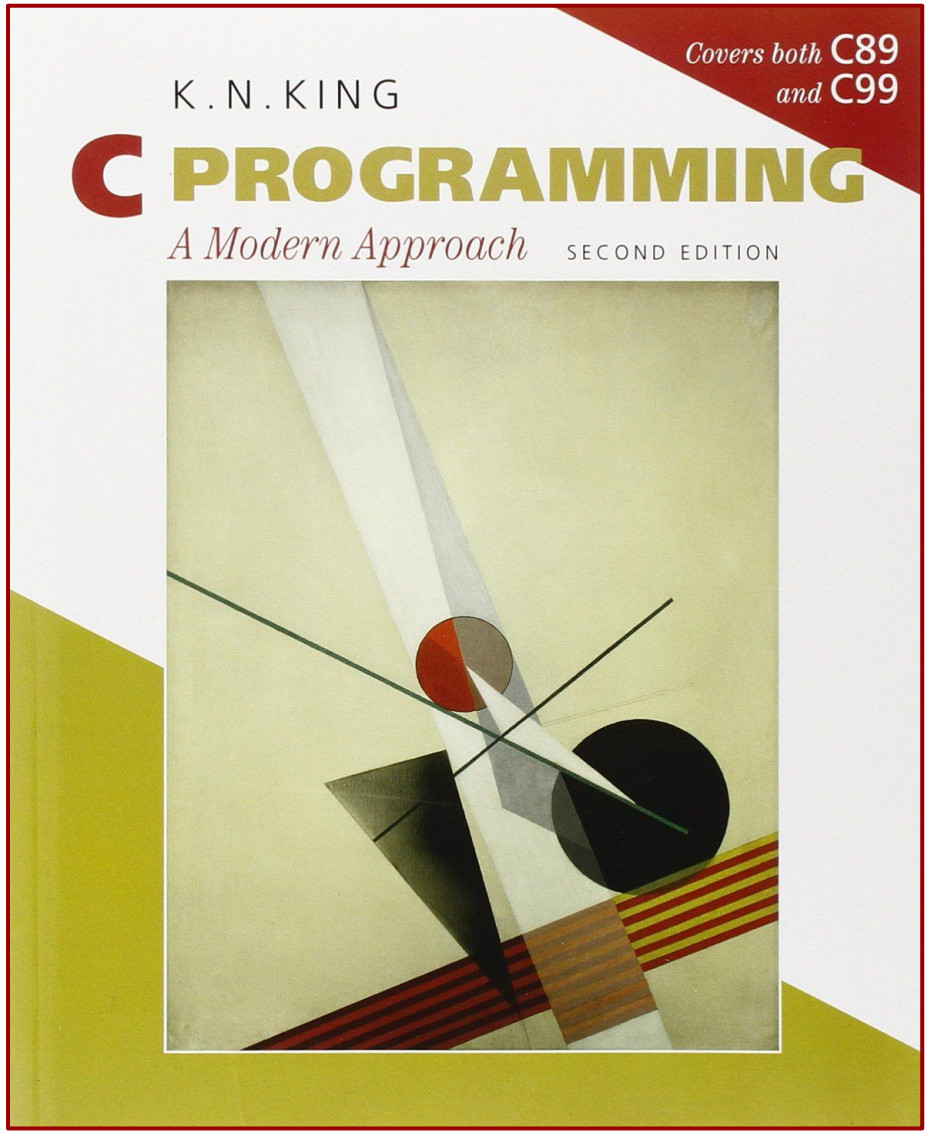


Dennis Ritchie at the Japan Prize Foundation in May 2011

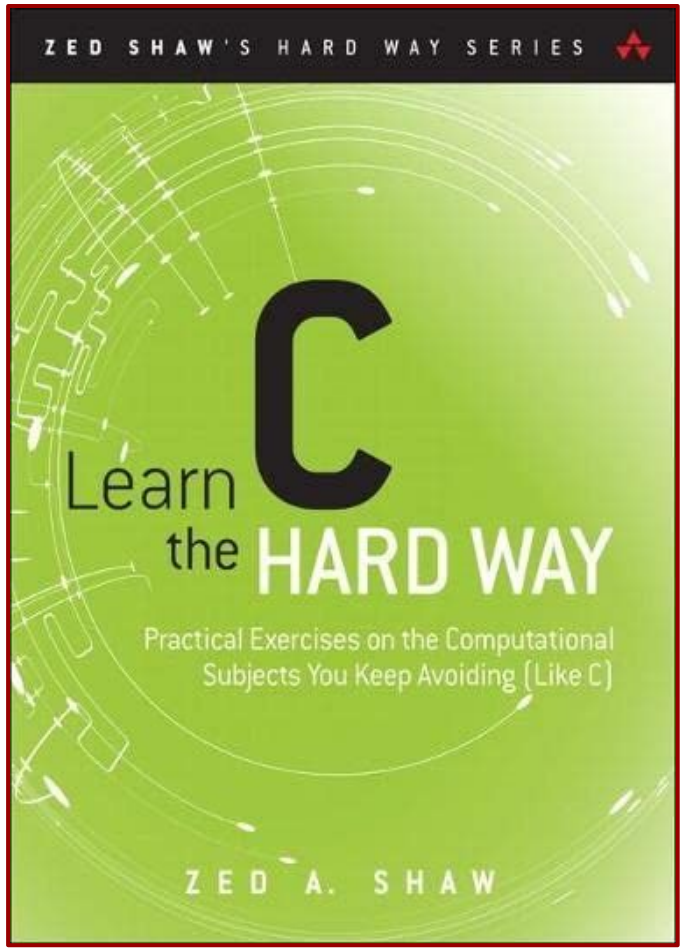
Born	September 9, 1941[1][2][3][4] Bronxville, New York, U.S.
Died	c. October 12, 2011 (aged 70) Berkeley Heights, New Jersey, U.S.
Nationality	American
Alma mater	Harvard University (Ph.D., 1968)

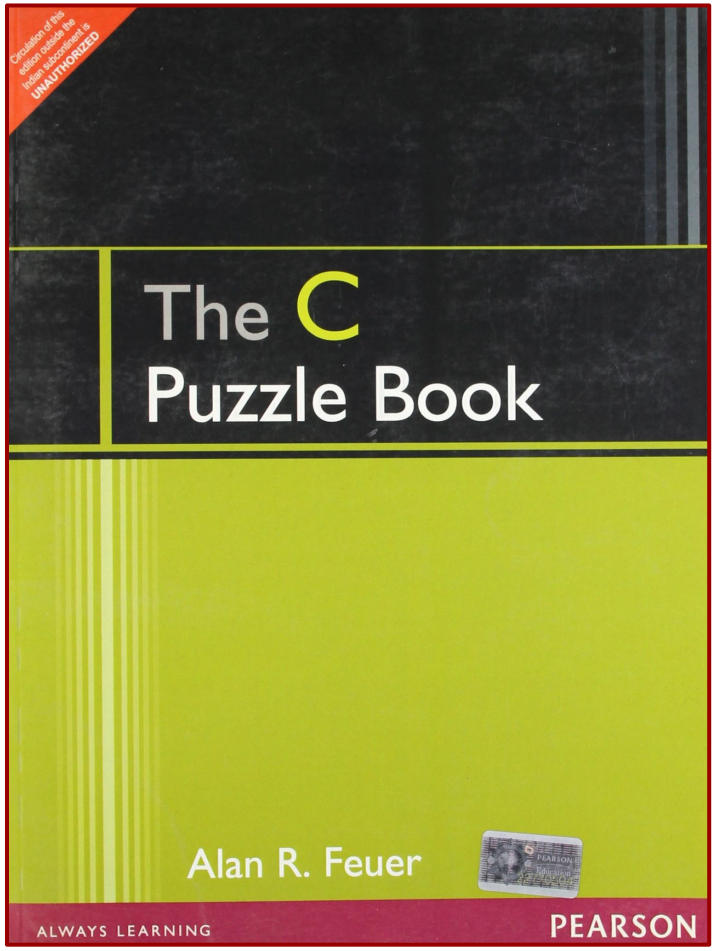
UE22CS151B: Problem Solving with C Reference Books

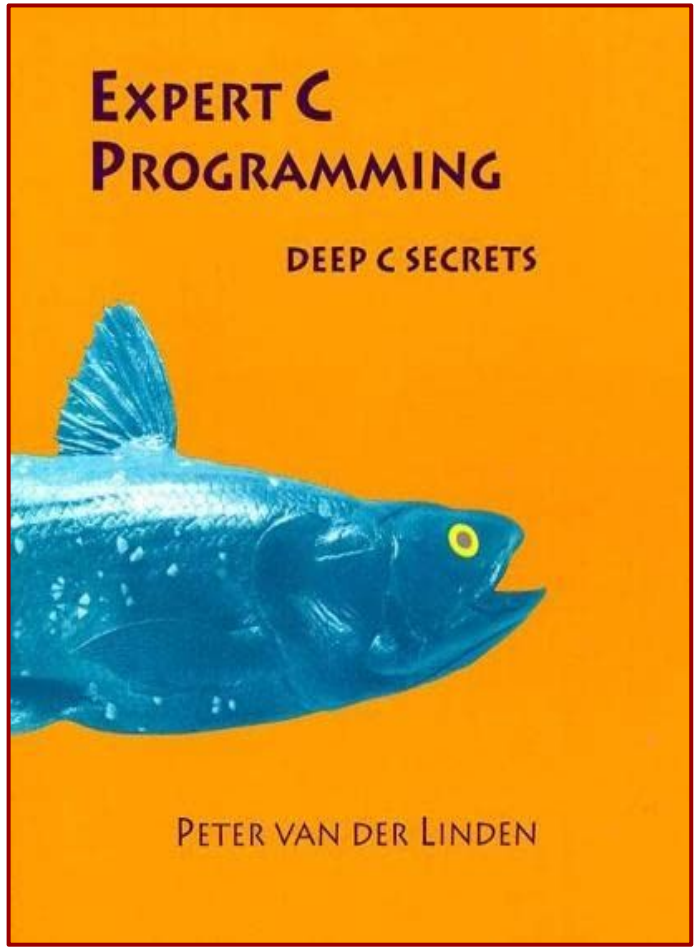




UE22CS151B: Problem Solving with C Reference Books







Week No.	Month	Day						No. of working days	Activities/Events
		Mon	Tue	Wed	Thu	Fri	Sat		
1.	Feb				8	9	10	5	Class commencement and Course registration
2.	Feb		13	**14**	15	16	17	5	
3.	Feb	19	20	21	22	**23**	24	5	
4.	Feb/Mar	26	**27**	28 **FAM I**	29	**1**	2	5	
5.	Mar	4	5	6	7	**8 H**	9	4	8th-Maha Shivaratri
6.	Mar	11	12	13 **CCM I**	14	15	16	5	
7.	Mar	18 **ISA 1**	19 **ISA 1**	20 **ISA 1**	21 **ISA 1**	22 **ISA 1**	23	5	ISA 1 Week(Units I & II)
8.	Mar	25	26	27	28	29 **AT**	30 **AT**	4	AT- Aatmatisha,The Annual Techno cultural Fest
9.	Apr	1	2	3	4	5	6 **PTM I**	5	
10.	Apr	8	**9 H**	10	**11 H**	12	13	3	9th – Chandramana Ugadi 11th - Ramzan
11.	Apr	15	16	17	18	19	20	5	
12.	Apr	22	23	24 **FAM II**	25	26	27	5	
13.	Apr/May	29	30	**1 H**	2	3	4	4	1st- May Day
14.	May	6	7	8	9	10	11	5	
15.	May	13	14	15 **CCM II**	16	17	18	5	
16.	May	20	21	22	23	24	25	5	
17.	May/Jun	27 **ISA 2**	28 **ISA 2**	29 **ISA 2**	30 **ISA 2**	31 **ISA 2 LWD**	1 **PTM II**	5	ISA 2 Week(Units III & IV)
18.	Jun	3	4 **FASD**	5	6	7	8		END SEMESTER ASSESSMENT (6th Jun-21st Jun) 18th-Bakrid
19.	Jun	10	11	12	13	14	15		
20.	Jun	17	**18 H**	19	20	21	22		

FAM: Faculty Advisor Meeting
CCM: Class Committee Meeting

FASD: Final Attendance Status Display
PTM: Parent Teachers Meeting

LWD: Last working Day
ISA : In Semester Assessment

Announcement of Results: July 02, 2024
Commencement of next Semester: August 05 2024

PES UNIVERSITY

Slot #7, #8

12.2.2024

UE23CS151B : PSWC: Unit 1

- **Literals** are the constant values assigned to the **constant** variables.
- There are **four** types of literals that exist in c programming:
  - **Integer literal**
    - It is a numeric literal that represents only integer type values.
    - It represents the value neither in fractional nor exponential part.
    - It can be specified in the following three ways
      - **Decimal number (base 10)**
        - It is defined by representing the digits between 0 to 9. Example: 1,3,65 etc
      - **Octal number (base 8)**
        - It is defined as a number in which 0 is followed by digits such as 0,1,2,3,4,5,6,7. Example: 032, 044, 065, etc
      - **Hexadecimal number (base 16)**
        - It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-f) or (A-F)) Example 0XFE oxfe

- **Literals** are the constant values assigned to the constant variables.
    - **Float literal**
        - It is a literal that contains only **floating-point** values or **real** numbers.
        - These real numbers contain the number of parts such as **integer part**, **real part**, **exponential part**, and **fractional part**.
        - The **floating-point literal** must be specified either in **decimal** or in **exponential** form.
            - **Decimal form**
                - The **decimal form** must contain either **decimal point**, **real part**, **or both**.
                - If it does not contain either of these, then the compiler will throw an error.
                - The decimal notation can be prefixed either by '+' or '-' symbol that specifies the positive and negative numbers.
                - Example: +9.5, -18.738

- **Literals** are the constant values assigned to the constant variables.
    - **Float literal**
        - The floating-point literal must be specified either in **decimal** or in **exponential** form.
            - **Exponential form**
                - The **exponential form** is useful when we want to represent the number, which is having a big magnitude.
                - It contains two parts, i.e., **mantissa** and **exponent**.
                - For example, the number is **3450000000000**, and it can be expressed as **3.45e12** in an exponential form

UE23CS151B : PSWC: Unit 1

- **Literals** are the constant values assigned to the constant variables.
    - The floating-point literal must be specified either in decimal or in exponential form.
        - **Exponential form**
            - Syntax of float literal in **exponential form**
                - [+/-] <Mantissa> <e/E> [+/-] <Exponent>
            - Examples of real literal in exponential notation are
                - +3e24, -7e3, +3e-15
            - Rules for creating an **exponential notation**
                - In **exponential notation**, the **mantissa** can be specified either in **decimal** or **fractional** form
                - An **exponent** can be written in both **uppercase** and **lowercase**, i.e., **e** and **E**.
                - We can use both the signs, i.e., **positive** and **negative**, before the **mantissa** and **exponent**. Spaces are not allowed

UE23CS151B : PSWC: Unit 1

- **Literals** are the constant values assigned to the constant variables.
    - **Character Literal**
        - A **character literal** contains a single character enclosed within single quotes.
        - If we try to store more than one character in a character literal, then the warning of a multi-character character constant will be generated.
        - Representation of **character literal**
            - A **character literal** can be represented in the following ways:
                - It can be represented by specifying a single character within single quotes. For example, 'x', 'y', etc.
                - We can specify the escape sequence character within single quotes to represent a character literal. For example, '\n', '\t', '\b'.
                - We can also use the **ASCII** in integer to represent a character literal. For example, the ascii value of 65 is 'A'.
                - The octal and hexadecimal notation can be used as an escape sequence to represent a character literal. For example, '\043', '\0x22'.

UE23CS151B : PSWC: Unit 1

- **Literals** are the constant values assigned to the constant variables.
  - **String** Literal
    - A **string** literal represents multiple characters enclosed within double-quotes
    - It contains an additional character, i.e., '\0' (null character), which gets automatically inserted.
    - This **null** character specifies the **termination** of the **string**.

UE23CS151B : PSWC: Unit 1

- The **sizeof** operator is the most common operator in C.
- It is a **compile-time unary operator** and is used to compute the size of its operand.
- It **doesn't execute** (run) the **code** inside ()
- It returns the size of a variable.
- It can be applied to any data type, float type, pointer type variables
- When **sizeof()** is used with the data types, it simply returns the amount of memory allocated to that data type.
- The output can be different on **different machines** like a **32-bit system** can show different output while a **64-bit system** can show different of same data types

UE23CS151B : PSWC: Unit 1

- Input and output functions are available in the **c language** to perform the most common tasks.
- In every **c program**, three basic functions take place namely **accepting of data as input**, **the processing of data**, and **the generation of output**
- When a **programmer** says **input**, it would mean that they are **feeding** some **data** in the program.
- Programmer can give this **input** from the **command line** or **in the form of any file**.
- The **c programming language** comes with a set of various **built-in functions** for **reading** the **input** and then **feeding** it to the available program as per our requirements.
- When a **programmer** says **output**, they mean **displaying** some **data** and **information** on the **printer**, the **screen**, or any other **file**.
- The **c programming language** comes with various **built-in functions** for **generating** the **output** of the **data** on any **screen** or **printer**, and also redirecting the output in the form of binary files or text file.

UE23CS151B : PSWC: Unit 1

- The **unformatted functions** are **not capable** of **controlling** the **format** that is involved in **writing** and **reading** the available **data**.
- Hence **these functions** constitute the most **basic** forms of **input** and **output**.
- The supply of input or the display of output **isn't allowed** in the **user format**, hence we call these functions as **unformatted functions** for **input** and **output**.
- The unformatted input-output functions further have two categories:
    - The **character** functions
        - We use the character input functions for reading only a single character from the input device by default the keyboard
            - **getchar(), getche(),** and the **getch()** refer to the **input functions** of **unformatted type**
        - We use the character output functions for writing just a single character on the output source by default the screen
            - the **putchar()** and **putch()** refer to the output functions of unformatted type

UE23CS151B : PSWC: Unit 1

- The **unformatted functions** are **not capable** of **controlling** the **format** that is involved in **writing** and **reading** the available **data**.
- Hence **these functions** constitute the most **basic** forms of **input** and **output**.
- The supply of input or the display of output **isn't allowed** in the **user format**, hence we call these functions as **unformatted functions** for **input** and **output**.
- The unformatted input-output functions further have two categories:
    - The **string** functions
        - In any programming language including c, the **character array** or **string** refers to the **collection** of various **characters**
        - Various types of **input and output** functions are present in **c programming** that can easily read and write these strings.
            - The **puts()** and **gets()** are the most commonly used ones for **unformatted** forms
            - **gets()** refers to the **input** function used for **reading** the **string** characters
            - **puts()** refers to the **output** function used for **writing** the **string** characters

- **printf()**
  - This function is used to display one or multiple values in the output to the user at the console.
    - int printf(const char *format, ...)
    - Predefined function in stdio.h
    - Sends formatted output to stdout by default
    - Output is controlled by the first argument
    - Has the capability to evaluate an expression
    - On success, it returns the number of characters successfully written on the output.
    - On failure, a negative number is returned.
    - Arguments to printf can be expressions
  - While calling any of the formatted console input/output functions, we must use a specific format specifiers in them, which allow us to read or display any value of a specific primitive data type.
  - % [flags] [field_width] [.precision] conversion_character where components in brackets [] are optional.
  - The minimum requirement is % and a conversion character (e.g. %d)
    - %d, %x, %o, %f, %c, %p, %lf, %s

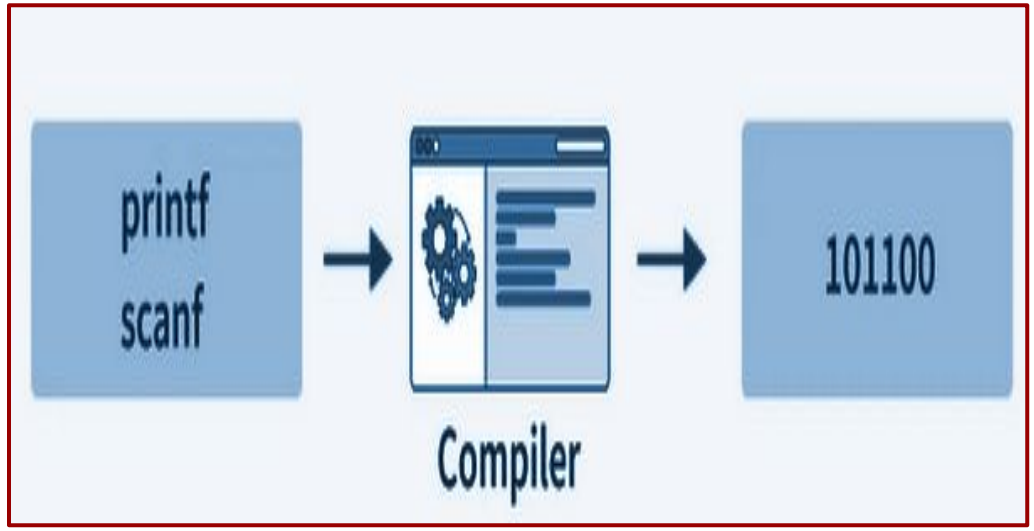
UE23CS151B : PSWC: Unit 1

- **Error**
  - a mistake
  - the state or condition of being wrong in conduct or judgement
  - a measure of the estimated difference between the observed or calculated value of a quantity and its true value
- **Preprocessor Error**
  - #error is a preprocessor directive in c which is used to raise an error during compilation and terminate the process

UE23CS151B : PSWC: Unit 1

- **Compile time** is the period when the programming code is converted to the machine code.

- **Runtime** is the period of time when a program is running and generally occurs after compile time



UE23CS151B : PSWC: Unit 1

Compile-time error	Run-time error
These errors are detected during the compile-time	These errors are detected at the run-time
Compile-time errors do not let the program be compiled	Programs with run-time errors are compiled successfully but an error is encountered when the program is executed
Errors are detected during the compilation of the program	Errors are detected only after the execution of the program
Compile-time errors can occur because of wrong syntax or wrong semantics	Run-time errors occur because of absurd operations

- **Compile time Error**
  - When the programmer does not follow the syntax of any programming language, then the compiler will throw the **Syntax Error**, such errors are also called **Compile Time Error**
  - **Syntax Errors** are easy to figure out because the compiler highlights the line of code that caused the error.

UE23CS151B : PSWC: Unit 1

- **Linker Error**
  - **Linker** is a program that takes the object files generated by the compiler and combines them into a single executable file.
  - **Linker Errors** are the errors encountered when the executable file of the code can not be generated even though the code gets compiled successfully.
  - This **Error** is generated when a different object file is unable to link with the main object file.
  - We can run into a linked error if we have imported an incorrect header file in the code

- **Runtime Error**
  - Errors that occur during the execution (or running) of a program are called **Runtime Errors.** These errors occur after the program has been compiled and linked successfully.
  - When a program is running, and it is not able to perform any particular operation, it means that we have encountered a runtime error.

- **Logical Error**
  - Sometimes, we do not get the output we expected after the compilation and execution of a program.
  - Even though the code seems error free, the output generated is different from the expected one.
  - These types of errors are called **Logical Errors**.
- **Semantic Errors**
  - Errors that occur because the compiler is unable to understand the written code are called Semantic Errors.
  - A semantic error will be generated if the code makes no sense to the compiler, even though it is syntactically correct.
  - It is like using the wrong word in the wrong place in the English language

UE23CS151B : PSWC: Unit 1

Keywords in C Programming			
auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

There are **32** keywords in c Language

Keyword Coverage
7 of 32
21.88%

PES UNIVERSITY

CELEBRATING 50 YEARS

UE23CS151B
THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University
nitin.pujari@pes.edu

For Course Digital Deliverables visit www.pesuacademy.com