Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #1
Prelude

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Prelude

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

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Prelude

Let us answer few questions before we start with C.

Q1. What is a Computer Programming Language (CPL)?

- Any set of rules that converts strings, or graphical program elements in the case of visual programming languages, to various kinds of **machine code output.**
- A **CPL** is an **artificial language** that can be used to control the behaviour of a machine, particularly a computer.
- **CPLs**, like human languages, are defined through the use of syntactic and semantic rules, to determine structure and meaning respectively.
- **CPLs** are used to implement **algorithms**.
- **CPLs allow us to give instructions to a computer** in a language the computer understands.
- **Formal computer language or constructed language** designed to communicate instructionsto a machine, particularly a computer (wiki definition).
- Can be used to create programs **to control the behaviour of a machine.**

Q2. Why CPL?

- Advance our ability to develop real algorithms.
- Majority of CPLs come with a lot of features for the Computer Programmers - CP.
- CPLs can be used in a proper way to get the best results.
- Improve Customization of our Current Coding.
- By using basic features of the existing CPL we can simplify things to program a better option to write resourceful codes.
- There is no compulsion of writing code in a specific way, but rather is the usage of features used and clarity of the concept.

Q3. Why so many Programming Languages?

To choose the right language for a given problem. Example Domains are listed – Web Browsers, Social Networks, Image Viewer, Facebook.com, Bing, Google.com, Games, Various Operating Systems.

Q4. What are the different Levels of Programming Language?

- **Low Level**
  - Binary codes which CPU executes
  - Programmer's responsibility is more
  - Machine Language
- **Middle Level**
  - Offers basic data structure and array definition, but the programmers shouldtake care of the operations
  - C and C++
- **High Level**
  - Programmer concentrates on the algorithm and programming itself
  - Java , Python, Pascal

High Level	*Middle Level*	*Low Level*
High level languages provide almost everything that the programmer might need to do as already built into the language	Middle level languages don't provide all the built-in functions found in high level languages, but provides all building blocks that we need to produce the result we want	Low level languages provides nothing other than access to the machines basic instruction set
Examples: Java, Python	C, C++	Assembler

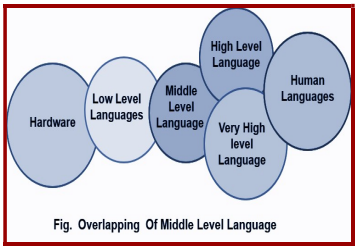


Fig. Overlapping Of Middle Level Language

Q5. What is the meaning of Paradigm?

A programming paradigm is a **style, or "way," of programming**. Some languages make it easy to write in some paradigms but not all of them.

Imperative: Programming with an explicit sequence of commands that update state –
Example: python

Declarative: Programming by specifying the result you want, not how to get it
Example: LISP, SQL

Structured: Programming with clean, goto-free, nested control structures

Example: C Language

Procedural: Imperative programming with procedure calls

Example: C Language.

Functional (Applicative): Programming with function calls that avoid any global state

Examples: Scheme, Haskell, Miranda and JavaScript.

Function-Level (Combinator): Programming with no variables at all

Examples: Scheme, Haskell, Miranda and JavaScript.

Object-Oriented: Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces.

Class-based: Objects get state and behavior based on membership in a class.

Prototype-based: Objects get behavior from a prototype object.

Examples: Java, C++, Python

Event-Driven: Programming with emitters and listeners of asynchronous actions.

Flow-Driven: Programming processes communicating with each other over predefined channels.

Logic (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.

Constraint: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.

Aspect-Oriented: Programming cross-cutting concerns applied transparently.

Reflective: Programming by manipulating the program elements themselves.

Array: Programming with powerful array operators that usually make loops unnecessary.

Q6: Which Languages are used while Developing Whatsapp? Think. !

Erlang, JqGrid, Libphonenum, LightOpenId, PHP5, Yaws and many more…

Q7. Why should one learn C? What are the advantages of 'C'? Is not 'C' an outdated language?

We have to fill our stomach every day 3 or 4 times so that our brain and body get enough energy to function. How about eating Vidyarthi Bhavan Dosa

every day? What about Fridays when the eatery is closed? Why not buy Dosa batter from some nearby shop? Or do you prefer to make the batter yourself? Would you have time to do that? Would that depend on how deep your pockets are? Would you like to decrease your medical bills?

Every language has a philosophy. The language used by poets may not be suitable for conversation. Poets use ambiguity in meaning to their advantage, and some verses in Sanskrit have more than one meaning. But that will not be suitable for writing a technical report. The goal of 'C' is efficiency. The safety is in the hands of the programmer. 'C' does very little apart from what the programmer has asked for.

Example: When we index outside the bounds of a list in Python, we get an "index error" at runtime. To support this feature, Python runtime should know the current size of a list and should also check whether the index is valid each time we index on a list. You are all very good programmers, and I am sure you never get an index error. You get what you deserve. If you are lucky, the program crashes. Otherwise, something subtle may happen, which later may lead to catastrophic failures.

- C gives importance to efficiency
- C is not very safe; you can make your program safe
- C is not very strongly typed; mixing of types may not result in errors
- C is the language of choice for all hardware related softwares
- C is the language of choice for software's like operating system, compilers, linkers, loaders, device drivers.

Q8. Is 'C' not an old language?

Yes and No.

It was designed by Dennis Ritchie –We use 'C' like languages and Unix like operating systems both have his contribution – in 70s. But the language has evolved over a period. The latest 'C' was revised in 2011.

There is one more reason to learn 'C'. 'C' is the second most popular language as of now according to TIOBE index ratings. https://www.tiobe.com/tiobe-index/.

Q9. What is TIOBE Index?

- **The Importance Of Being Earnest" - TIOBE.**
- TIOBE is an indicator of the popularity of programming languages.
- The TIOBE index is updated once a month.
- The ratings are based on the number of skilled engineers world-wide, courses and third party vendors.
- Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Bing are used
- It is not about the best programming language or the language in which most lines of code have been written.

Feb 2025	Feb 2024	Change	Programming Language	Ratings	Change
1	1		Python	23.88%	+8.72%
2	3	↑	C++	11.37%	+0.84%
3	4	↑	Java	10.66%	+1.79%
4	2	↓	C	9.84%	-1.14%
5	5		C#	4.12%	-3.41%
6	6		JavaScript	3.78%	+0.61%
7	7		SQL	2.87%	+1.04%
8	8		Go	2.26%	+0.53%

Q10: What is the history of PLs?

- 1820-1850 England, Charles Babbage invented two mechanical Computational device i.e., Analytical Engine and Difference Engine
- In 1942, United States, ENIAC used electrical signals instead of physical motion
- In 1945, Von Newman developed two concepts: Shared program technique and Conditional control transfer
- In 1949, Short code appeared

- In 1951, Grace Hopper wrote first compiler, A-0
- Fortran: 1957, John Backus designed.
- Lisp, Algol- 1958
- Cobol: 1959
- Pascal: 1968, Niklaus Wirth
- **C: 1972, D Ritchie**
- C++: 1983, Bjarne Stroustrup, Compile time type checking, templates are used.
- Java: 1995, J. Gosling, Rich set of APIs and portable across platform through the use of JVM
- **Development of C**
  - **Martin Richards, around 60's developed BCPL [Basic Combined Programming Language]**
  - **Enhanced by Ken Thompson and Introduced B language.**
  - **C is originally developed between 1969 and 1973 at Bell Labs by Dennis Ritchie and Kernighan. Closely tied to the development of the Unix operating system**
- **Standardized by the ANSI [American National Standards Institute] since 1989 and subsequently by ISO [International Organization for Standardization].**
  - **ANSI C, C89, C99, C11, C17, or C23**

Happy Coding!

Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #3
GCC, PDLC

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: GCC, PDLC

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

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction to GCC

The **original author of the GNU C Compiler is Richard Stallman,** the founder of the GNU Project. **GNU** stands for **GNU's not Unix.** The GNU Project was started in 1984 to create a **complete Unix-like operating system as free software**, in order to promote freedom and cooperation among computer users and programmers. The first release of gcc was made in 1987, as the first portable ANSI C optimizing compiler released as free software. A **major revision of the compiler came with the 2.0 series in 1992**, which added the ability to compile C++. The acronym gcc is now used to refer to the "**GNU Compiler Collection**"

GCC has been extended to support many additional languages, including **Fortran, ADA, Java and Objective-C.** Its development is guided by the gcc Steering Committee, a group composed of representatives from gcc user communities in industry, research and academia

Features of GCC:

- A portable compiler, it runs on most platforms available today, and can produce **output** for many types of **processors**
- Supports **microcontrollers**, **DSPs** and **64-bit** CPUs
- It is not only a native compiler, it can also cross-compile any program, producing executable files for a different system from the one used by **gcc** itself.
- Allows software to be compiled for embedded systems
- **Written in C with a strong focus on portability**, and can compile itself, so it can be adapted to new systems easily
- It has multiple language frontends, for parsing different languages
- It can compile or cross-compile programs in each language, for any architecture Example: Can compile an ADA program for a **microcontroller** or a C program for a **supercomputer**
- It has a **modular design**, allowing support for new languages and architectures to be added.
- It is **free software,** distributed under the GNU General Public License (GNU GPL), which means we have the freedom to use and to modify gcc, as with all GNU

software.

- gcc users have the freedom to share any enhancements and also make use of enhancements to gcc developed by others.

Installation of gcc on different Operating systems:

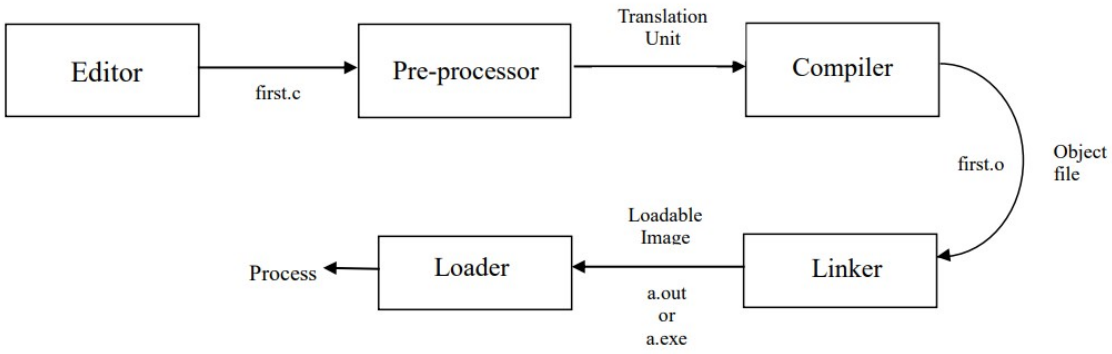
- **Windows OS – Installation of gcc using Mingw.**

  https://www.youtube.com/watch?v=sXW2VLrQ3Bs

- **Linux /MAC OS – gcc available by default**

Program Development Life Cycle [PDLC]

Phases involved in PDLC are **Editing, Pre-processing, Compilation, Linking, Loading and execution.**



The stages for a C program to become an executable are the following:

1. Editing:

We enter the program into the computer. This is called editing. We save the **sourceprogram**. Let us create first.c

$ gedit first.c // press

enter key#include

<stdio.h>

```
int main()
{       // This is a comment
        //Helps to understand the code Used for readability
        printf("Hello Friends");
        return 0;
}// Save this code in first.c
```

2. Pre-Processing:

In this stage, the following tasks are done:

a. Macro substitution–To be discussed in detail in Unit-5

b. Comments are stripped off

c. Expansion of the included files

The output is called a **translation unit or translation.**

To understand Pre-processing better, compile the above 'first.c' program using flag -E, which willprint the pre-processed output to stdout.

$ gcc -E first.c// No file is created. Output of pre–processing on the standard output-terminal

```
.....
# 846 "/usr/include/stdio.h" 3 4
# 886 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute_____((_
nothrow_)); extern int ftrylockfile (FILE *__stream) __attribute
_____((_
nothrow_)) ; extern void funlockfile (FILE *__stream) __
attribute_____((_
nothrow_));
.....
# 916 "/usr/include/stdio.h" 3 4
```

```
# 2

"first.c"

int

main()

{

printf("HelloFriends");

return0;

}
```

In the above output, you can see that the source file is now filled with lots of information, but still at the end of it we can see the lines of code written by us. Some of the observations are as below.

- Comment that we wrote in our original code is not there. This proves that all the comments are stripped off.
- The '#include' is missing and instead of that we see whole lot of code in its place. So it is safe to conclude that stdio.h has been expanded and literally included in our source file.

3. Compiling

Compilation refers to the process of converting a program from the textual source code into machine code, the sequence of 1's and 0's used to control the central processing unit (CPU) of the computer. Machine code is then stored in a file known as an executable file. C programs can be compiled from a single source file or from multiple source files, and may use system libraries and header files. Compiler processes statements written in a particular programming language and converts them into machine language or "code" that a computer's processor uses. The translation is **compiled**. The output is called an **object file**. There may be more than one translation. So, we may get multiple object files. When compiling with '-c', the compiler automatically creates an object file whose name is the same as the source file, but with '.o' instead of the original extension

gcc -c first.c

This command does pre-processing and compiling. Output of this command is first.o -> **Object file.**

4. Linking

It is a computer program that takes one or more object files generated by a compiler and combine them into one, executable program. Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program. This is the stage at which all the linking of function calls with their definitions are done. Till stage, gcc doesn't know about the definition of functions like printf(). Until the compiler knows exactly where all of these functions are implemented, it simply uses a place-holder for the function call. The definition of printf() is resolved and the actual address of the function printf() is plugged in. We put together all these object files along with the predefined library routines by **linking**. The output is called as **image or a loadable image** with the name a.out [linux based] or a.exe[windows]

gcc first.o // Linking to itself

// If more than one object files are created after compilation, all must be mentioned with gcc and linked here to get one executable.

If a file with the same name[a.out/a,exe] as the executable file already exists in the current directory it will be overwritten. This can be avoided by using the gcc option -o. This is usually given as the last argument on the command line.

gcc first.o -o PESU // loadable image - Output file name is PESU

5. Executing the loadable image

Loader loads the image into the memory of the computer. This creates a **process** when command is issued to execute the code. We **execute or run** the process. We get some results. In 'C', we get what we deserve!

In windows,

a.exe and press enter

OR

PESU and press enter

In Linux based systems,

./aout and press enter

OR

./PESU and press enter

For interested students:

The following options of gcc are a good choice for finding problems in C and C++ programs.

gcc **-ansi -pedantic -Wall -W -Wconversion -Wshadow -Wcast-qual -Wwrite-strings**

Happy Coding!

UE24CS151B: Problem Solving with C - Introduction for B.Tech Second Semester A & N Section - EC Campus Lecture Slides - Slot #4, #5, #6

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

UE24CS151B: Problem Solving with C - Syllabus

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

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

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

Arrays–1D and 2D, Pointers, Pointer to an array, Array of pointers, Functions, Call back, Storage classes, Recursion, Searching, Sorting

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

Strings, String Manipulation Functions & Error handling, Command line arguments, Dynamic Memory Management functions & Error handling, Structures, #pragma, Array of Structures, Pointer to structures, Passing Structure and Array of structure to a function, Bit fields, Unions, Enums, Lists, Stack, Queue, Priority Queue.

Unit IV: File Handling and Portable Programming – 14 Hours – 18 Slots

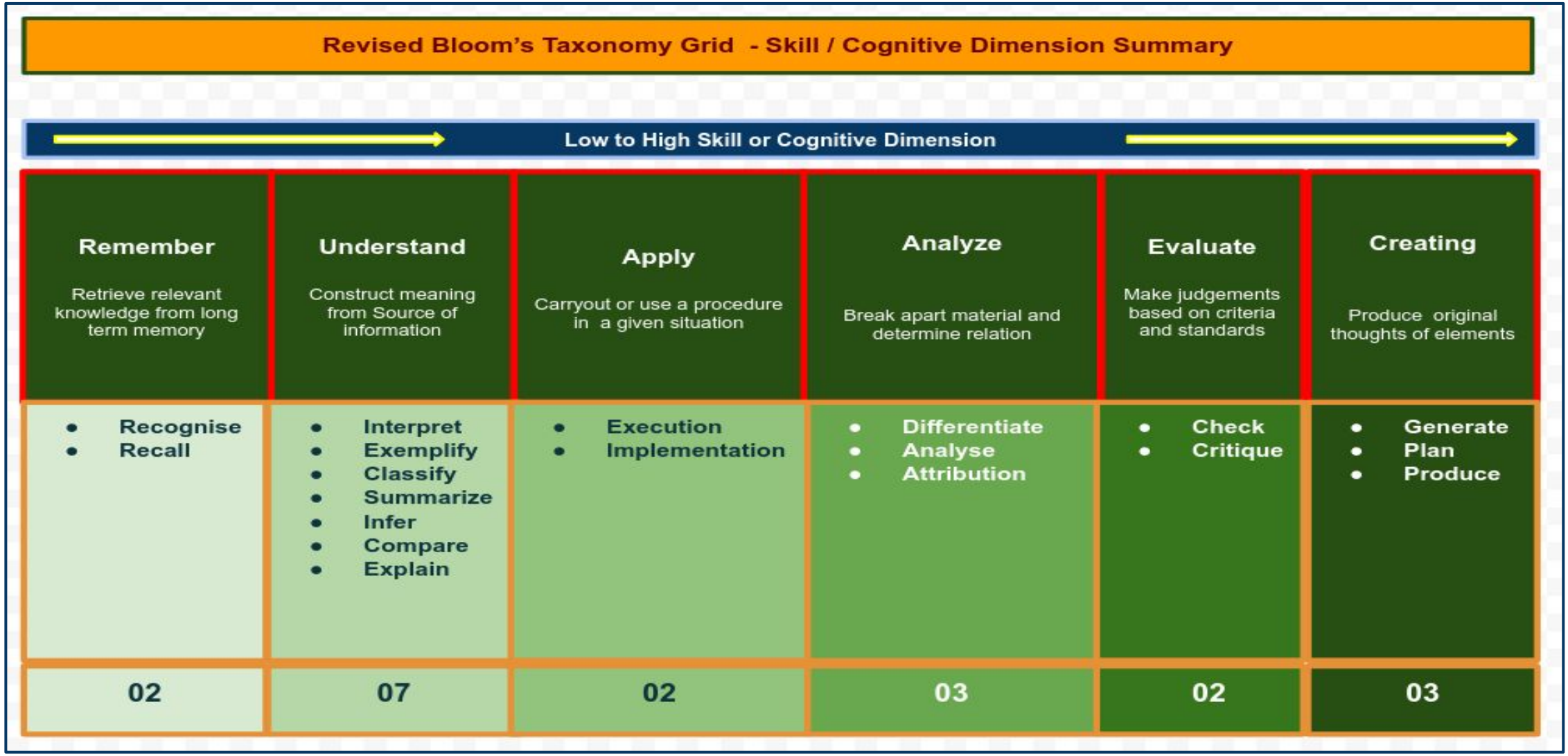
File IO using redirection, File Handling functions of C, Searching, Sorting, Header files, Comparison of relevant User defined and Built-in functions, Variable Length Arguments, Environment variables, Preprocessor Directives, Conditional Compilation.

UE24CS151B: Problem Solving with C - Course Objectives

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

- **CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine**
- **CObj2: Map algorithmic solutions to relevant features of C programming language constructs**
- **CObj3: Gain knowledge about C constructs and its associated ecosystem**
- **CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviours**

Bloom's Taxonomy



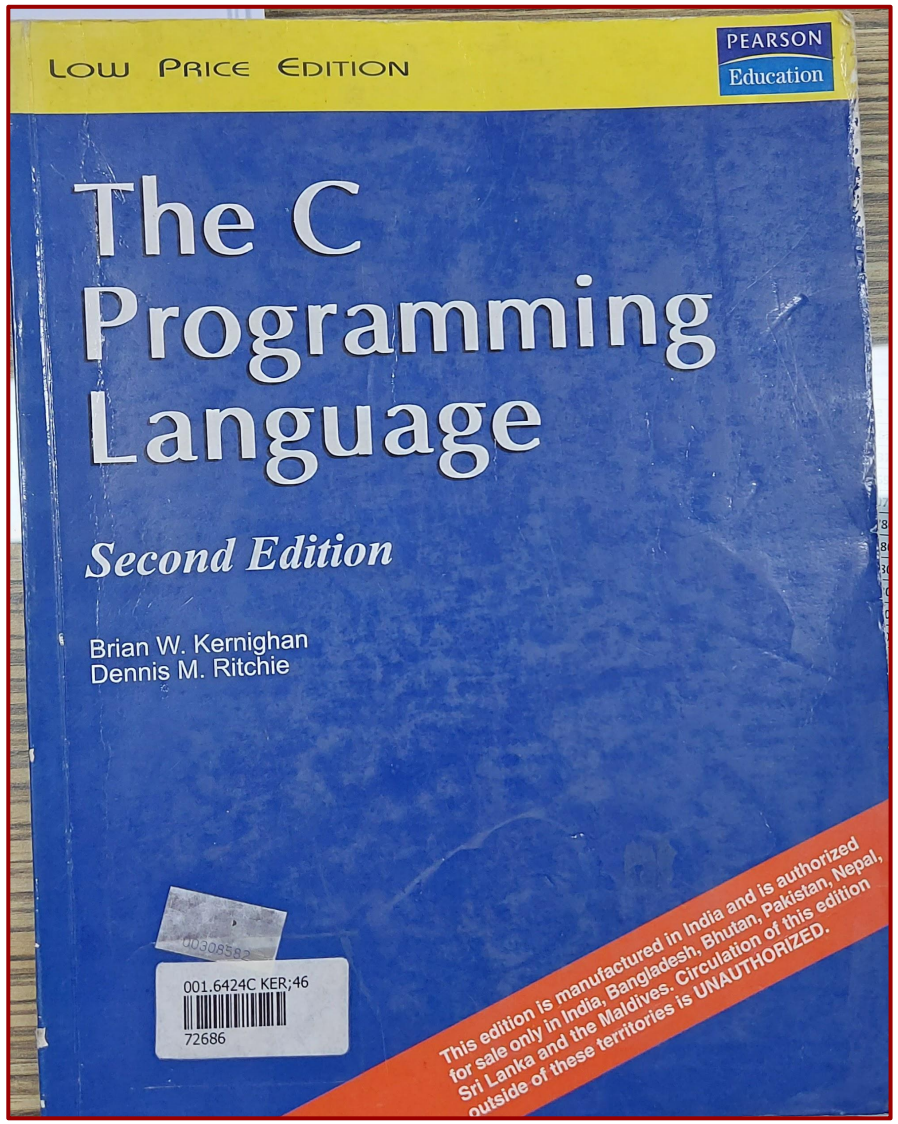
Remember	Understand	Apply	Analyze	Evaluate	Creating
Retrieve relevant knowledge from long term memory	Construct meaning from Source of information	Carryout or use a procedure in a given situation	Break apart material and determine relation	Make judgements based on criteria and standards	Produce original 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

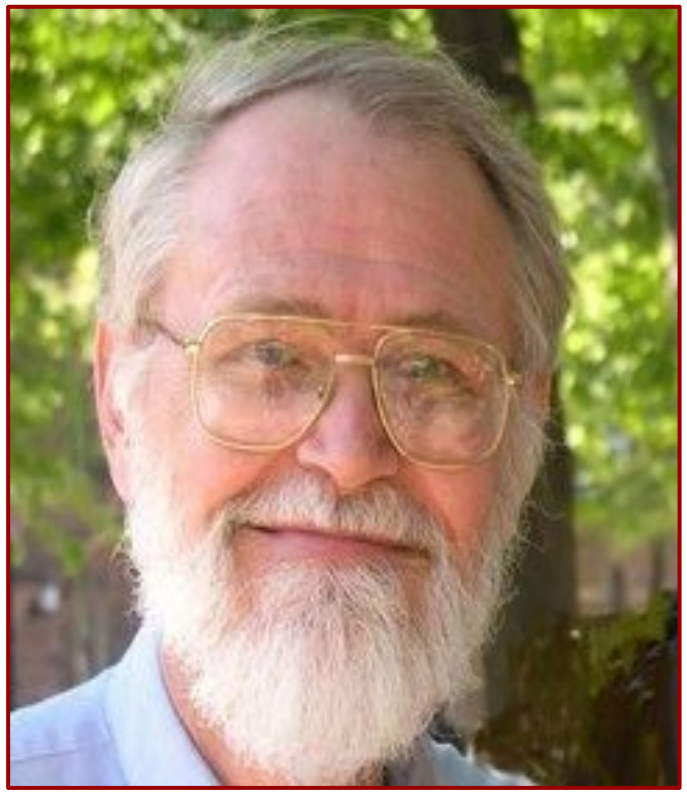
UE24CS151B: Problem Solving with C - Course Outcomes

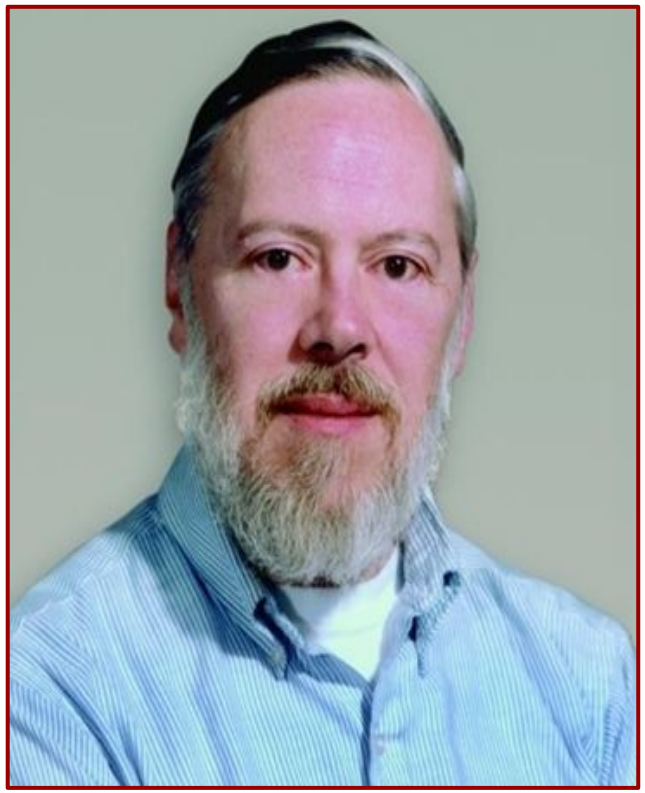
At the end of the course, the student will be able to

- CO1: Understand and apply algorithmic solutions to counting problems using appropriate C constructs
- CO2: Understand, analyse and apply Sorting and Searching techniques
- CO3: Understand, analyse and apply text processing and string manipulation methods using Arrays, Pointers and functions
- CO4: Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

UE24CS151B: Problem Solving with C Text Book







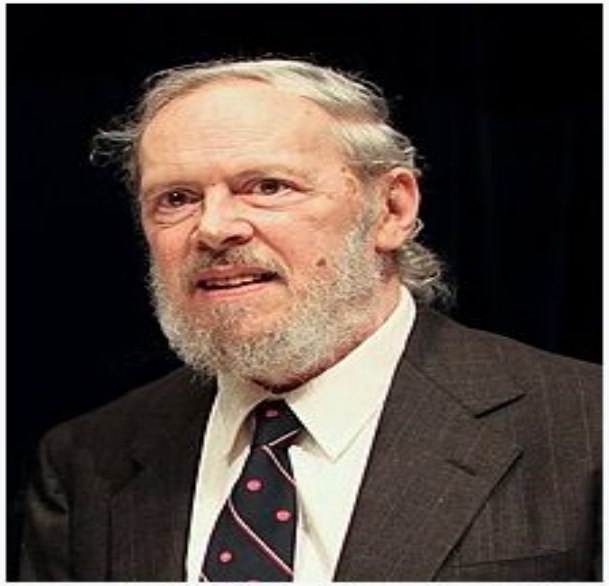
UE24CS151B: About Text Book Authors

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)

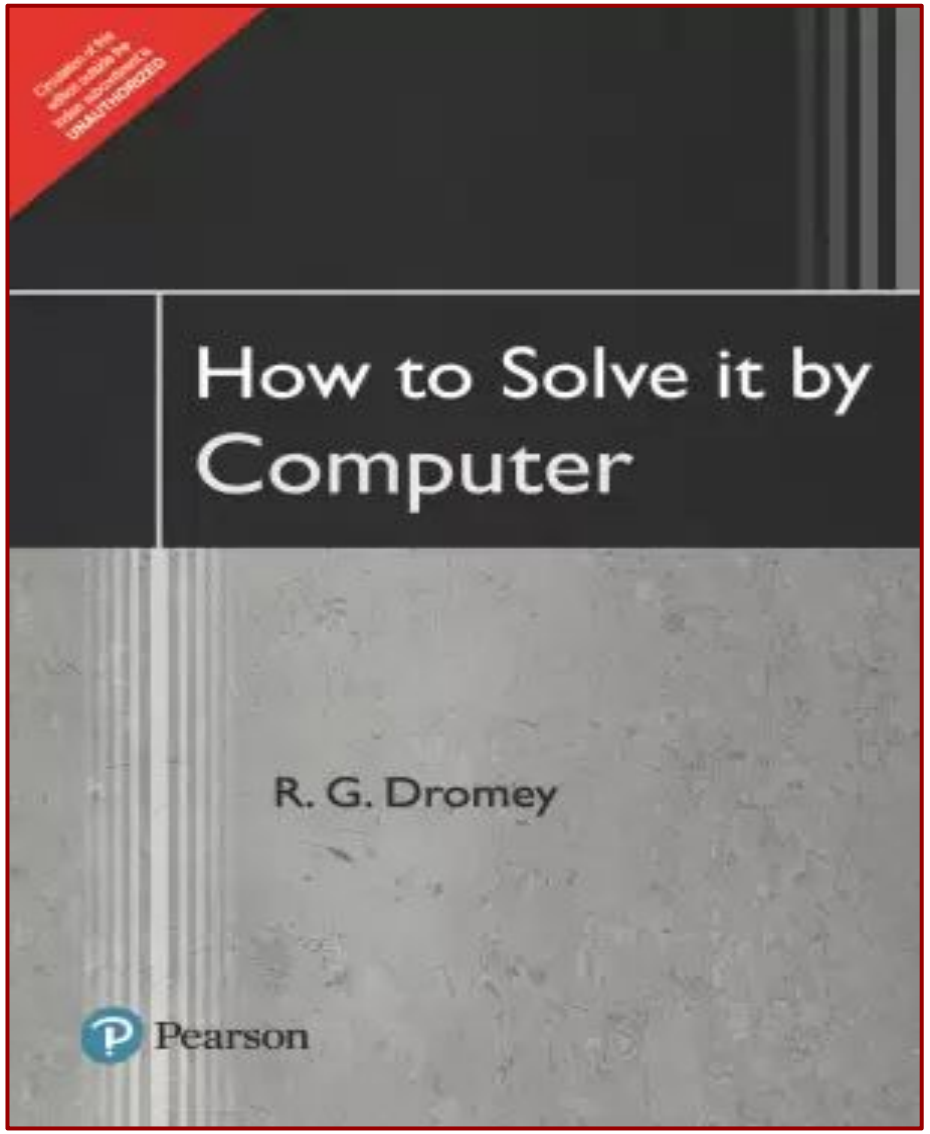
Dennis Ritchie

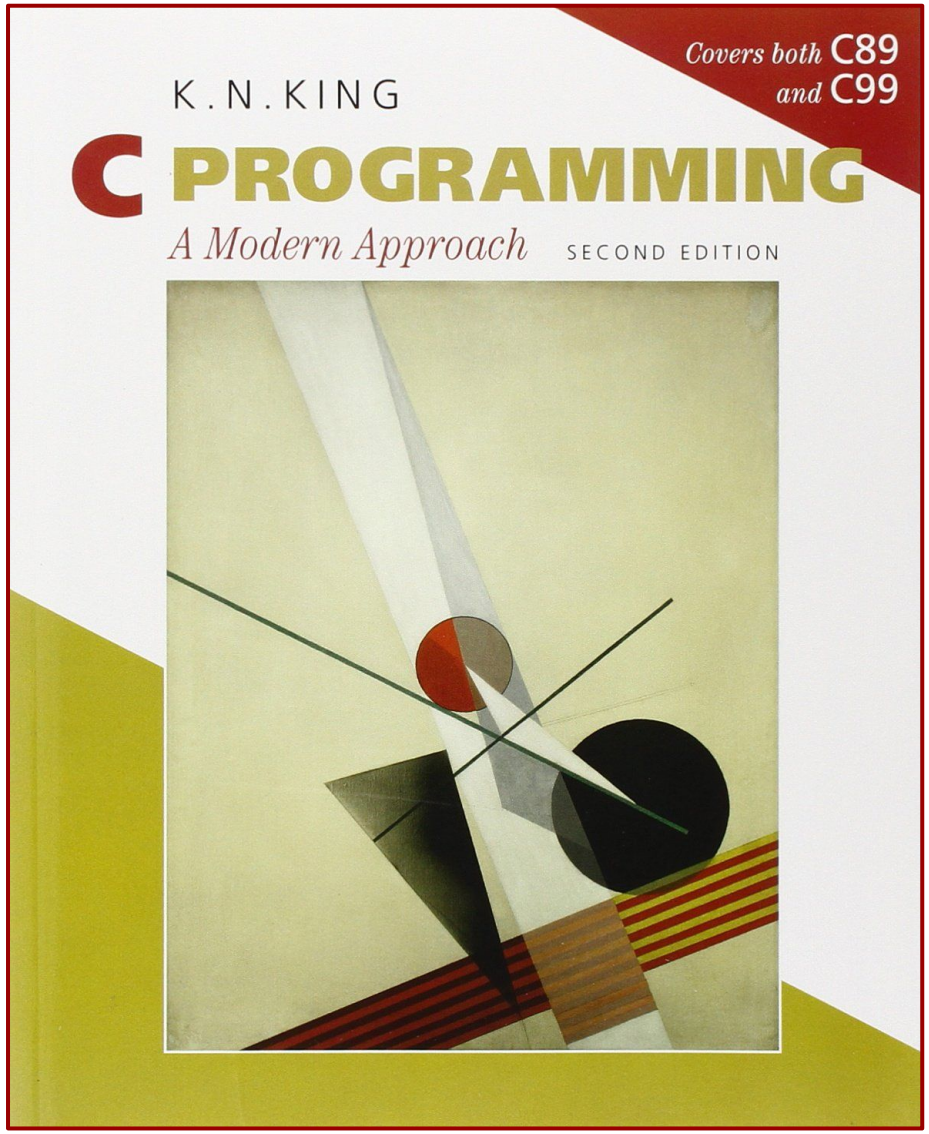


Dennis Ritchie at the Japan Prize Foundation in May 2011

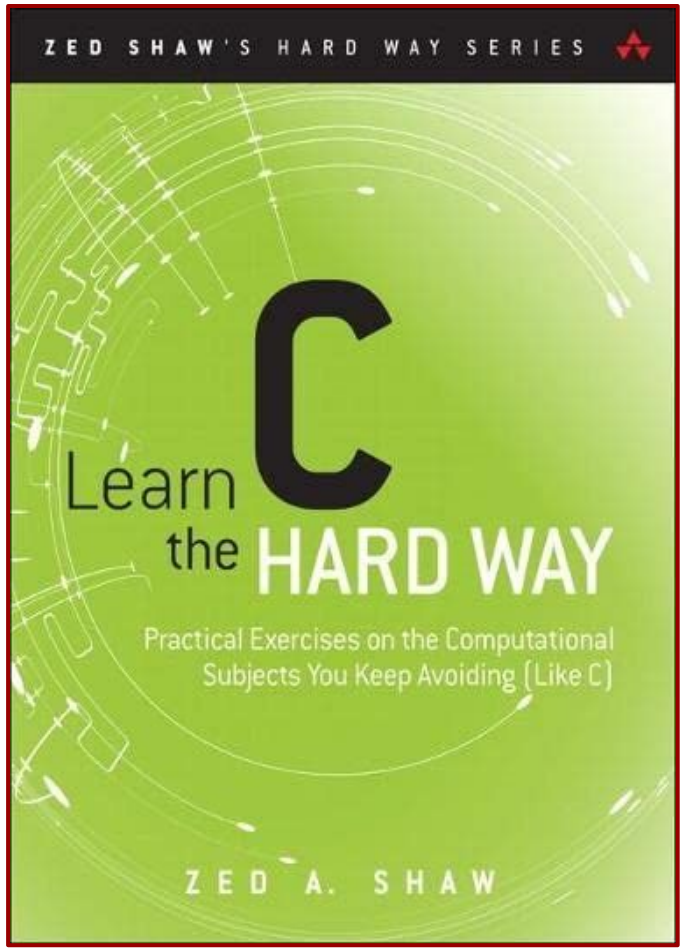
Born	September 9, 1941[1][2][3][4] Bronxville, New York, U.S.
Died	c. October 12, 2011 (aged 70) Berkeley Heights, New Jersey, U.S.
Nationality	American
Alma mater	Harvard University (Ph.D., 1968)

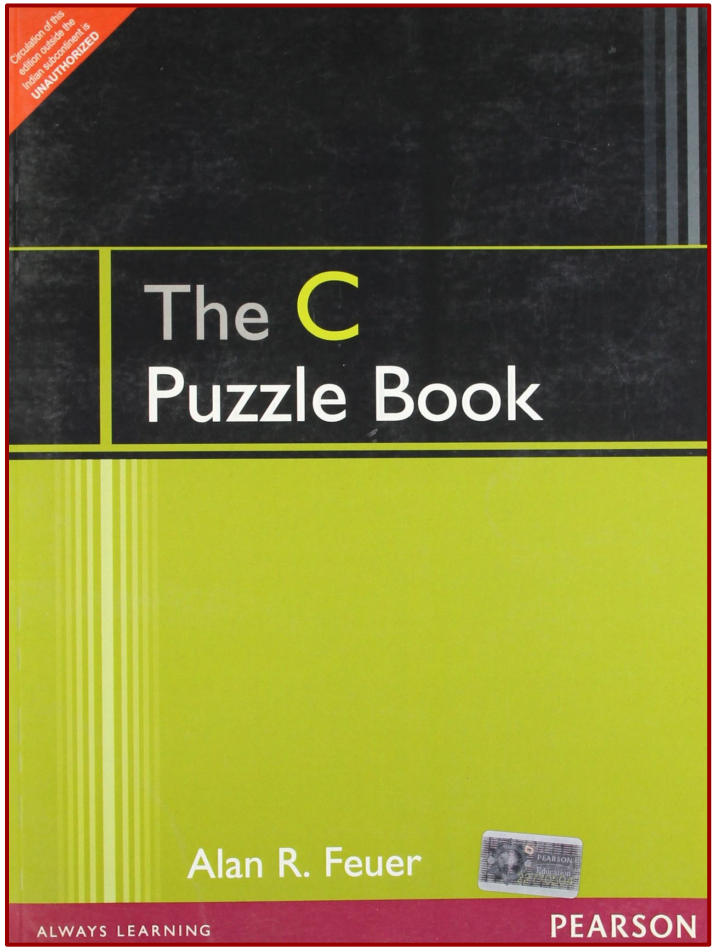
UE24CS151B: Problem Solving with C Reference Books

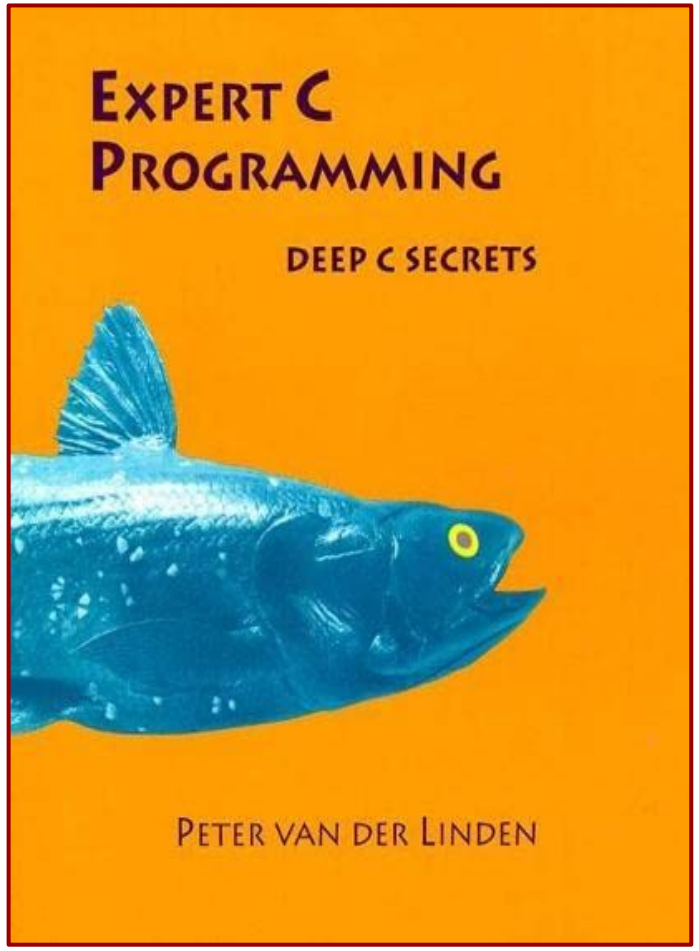




UE24CS151B: Problem Solving with C Reference Books







UE24CS151B : PSWC: Unit 1 - gcc Insights

- The original author of the GNU C Compiler (**gcc**) is **Richard Stallman**, the founder of the **GNU** Project
- **GNU** stands for **GNU's not Unix**
- The GNU Project was started in **1984** to create a complete Unix-like operating system as **free software**, in order to promote freedom and cooperation among computer users and programmers
- The **first** release of **gcc** was made in **1987,** as the first portable ANSI C optimizing compiler released as free software
- A major revision of the compiler came with the **2.0** series in **1992**, which added the ability to compile **C++**
- The acronym **gcc** is now used to refer to the "**GNU Compiler Collection**"
- **gcc** has been extended to support many additional languages, including Fortran, ADA, Java and Objective-C
- Its development is guided by the **gcc** Steering Committee, a group composed of representatives from **gcc** user communities in industry, research and academia

UE24CS151B : PSWC: Unit 1 - gcc Features

- **gcc** is a portable compiler, it runs on most platforms available today, and can produce **output** for many types of **processors**
- **gcc** also supports **microcontrollers**, **DSPs** and **64-bit** CPUs
- **gcc** is not only a native compiler, it can also cross-compile any program, producing executable files for a different system from the one used by **gcc** itself.
- **gcc** allows software to be compiled for embedded systems which are not capable of running a compile
- **gcc** written in C with a strong focus on portability, and can compile itself, so it can be adapted to new systems easily
- **gcc** has multiple language frontends, for parsing different languages
- **gcc** can compile or cross-compile programs in each language, for any architecture
- **gcc,** for example,can compile an ADA program for a **microcontroller**, or a C program for a **supercomputer**

UE24CS151B : PSWC: Unit 1 - gcc Features

- **gcc** has a modular design, allowing support for new languages and architectures to be added.

- **gcc** is free software, distributed under the GNU General Public License (GNU GPL), which means we have the freedom to use and to modify **gcc**, as with all GNU software.

- **gcc** users have the freedom to share any enhancements and also make use of enhancements to **gcc** developed by others.

- If we need support for a new type of **CPU**, a new language, or a new feature we can add it ourselves, or hire someone to enhance **gcc** for us, in addition we can hire someone to fix a bug if it is important for our work

UE24CS151B : PSWC: Unit 1 - gcc for c programming

- c is one those languages that allow **direct** access to the computer's **memory**.
- Historically, **c** has been used for writing low-level systems software, and applications where **high performance** or control over resource usage are **critical**
- Great care is required to ensure that memory is accessed correctly, to avoid corrupting other data-structures whenever one uses **c** language for programming
- In addition to C , the GNU Project also provides other high-level languages, such as C++, GNU Common Lisp (gcl), GNU Smalltalk (gst), the GNU Scheme extension language (guile) and the GNU Compiler for Java (gcj).
- These languages with exception to **c** and **c++**, do not allow the user to access memory directly, **eliminating** the possibility of **memory access errors**.
- They are a safer alternative to c and c++ for many applications

UE24CS151B : PSWC: Unit 1 - Compiling a c program using gcc

- **c** programs can be compiled from a single source file or from multiple source files, and may use system libraries and header files

- Compilation refers to the process of converting a program from the textual source code, in a programming language such as **c** into machine code, the sequence of 1's and 0's used to control the central processing unit (CPU) of the computer.

- **Machine code** is then **stored** in a file known as an **executable** file, sometimes referred to as a **binary** file

UE24CS151B : PSWC: Unit 1 - Compiling a c program using gcc

- `gcc -Wall PESU.c -o PESU`
- This compiles the source code in **PESU.c** to machine code and stores it in an executable file **PESU**'.
- The output file for the machine code is specified using the **'-o'** option.
- **-o** option is **usually** given as the **last** argument on the **command line**, If it is omitted, the output is written to a default file called **'a.out'**.
- If a file with the same name as the executable file already exists in the current directory it will be overwritten
- The option **'-Wall' turns on** all the most commonly-used **compiler warnings**, it is **recommended** that we always **use** this **option**!
- **gcc** will not produce any warnings **unless** they are **enabled**.
- Compiler **warnings** are an essential aid in **detecting** problems when programming in **c**
- Source code which does not produce any warnings by **gcc**, is said to be **compile cleanly**.

UE24CS151B : PSWC: Unit 1 - Finding errors in a simple program using gcc

- Compiler **warnings** are an essential aid when programming in c
- **Error** is not obvious at first sight, but can be detected by the compiler if the warning option '-Wall' has been enabled for **safety**
- The compiler distinguishes between **error** messages, which prevent successful compilation, and **warning** messages which indicate possible problems but **do not stop** the program from compiling
- It is very **dangerous** to develop a program **without checking** for compiler **warnings**.
- If there are any functions which are not used correctly they can cause the program to crash or produce incorrect results.
- Turning on the compiler warning option **'-Wall'** for safety will catch many of the commonest errors which occur in c programming

UE24CS151B : PSWC: Unit 1 - Creating object files from source files using gcc

- The command-line option '**-c**' is used to compile a source file to an **object** file.
- '-c' produces an object file '**<filename>.o**' containing the machine code for the main function.
- '**<filename>.o**' contains a reference to the **external** function <filename>, but the corresponding memory address is left undefined in the object file at this stage, which will be filled in later by linking
- There is **no need** to use the option '-o' to specify the name of the output file in this case.
- When compiling with '-c' the compiler **automatically** creates an object file whose name is the same as the source file, but with **'.o'** instead of the original extension

- The list of directories for **header files** is often referred to as the include path, and the list of directories for libraries as the library search path or link path
- For example, a header file found in '/usr/local/include' takes precedence over a file with the same name in '/usr/include'.
- Similarly, a library found in '/usr/local/lib' takes precedence over a library with the same name in '/usr/lib'
- When additional libraries are installed in other directories it is necessary to extend the search paths, in order for the libraries to be found.
- The compiler options '-I' and '-L' add new directories to the beginning of the include path and library search path respectively

UE24CS151B : PSWC: Unit 1 - c Language Standards using gcc

- By default, **gcc** compiles programs using the **GNU** dialect of the C language, referred to as **GNU C**
- This dialect incorporates the official ANSI/ISO standard for the C language with several useful GNU extensions, such as nested functions and variable-size arrays.
- Most ANSI/ISO programs will compile under GNU C without changes
  - `gcc -Wall -ansi <filename.c>`
  - `gcc -Wall <filename.c>`
- The command-line option '-pedantic' in combination with '-ansi' will cause gcc to reject all GNU C extensions, not just those that are incompatible with the ANSI/ISO standard
  - `gcc -Wall -ansi -pedantic <filename.c>`
- The following options are a good choice for finding problems in C programs
  - `gcc -ansi -pedantic -Wall -W -Wconversion -Wshadow -Wcast-qual -Wwrite-strings`

Note: **While this list is not exhaustive, regular use of these options will catch many common errors.**

- **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

- **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.

- **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.

UE24CS151B : PSWC: Unit 1 - Errors

- **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

UE24CS151B : PSWC: Unit 1 - Simple Input / Output Function

- 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.

UE24CS151B : PSWC: Unit 1 - Unformatted I/O

- 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

UE24CS151B : PSWC: Unit 1 - Unformatted I/O

- 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

UE24CS151B : PSWC: Unit 1 - Formatted Output in C - printf

- **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

UE24CS151B : PSWC: Unit 1 - keywords in c language

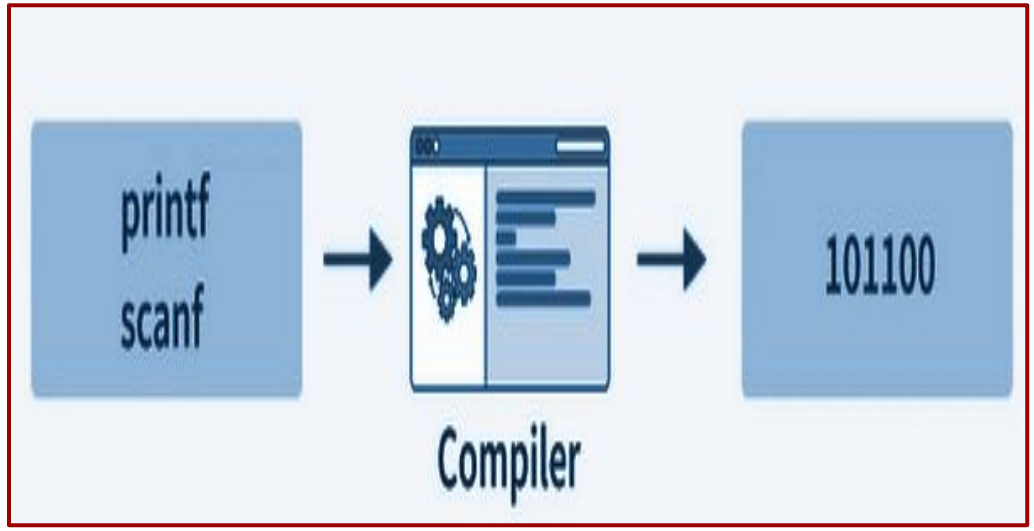
Keywords in C Programming			
auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

There are **32** keywords in c Language

There are **33** keywords in python

- **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



UE24CS151B : PSWC: Unit 1 - Compile time and Runtime in c

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

UE24CS151B : PSWC: Unit 1 - sizeof operator in c Language

- 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 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

UE24CS151B : PSWC: Unit 1 - Literals and Constants in c Language

- **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

UE24CS151B : PSWC: Unit 1 - Literals and Constants in c Language

- **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

UE24CS151B : PSWC: Unit 1 - Literals and Constants in c Language

- **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

UE24CS151B : PSWC: Unit 1 - Literals and Constants in c Language

- **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'.

- **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**.

PES UNIVERSITY

CELEBRATING 50 YEARS

UE24CS151B
End of Slot #4, #5, #6

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

For Course Digital Deliverables visit www.pesuacademy.com

Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #5
Variables and Data types

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Variables and Data types

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

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Basic constructs in C

C program consists of various tokens and a **token can be any identifier -> a keyword, variable, constant or any symbol.** For example, the following C statement consists of five tokens.

printf("HelloFriend
s");The individual tokens
are − printf
(
"HelloFriends"
)
;

Identifiers

An identifier is a **name used to identify a variable, function, or any other user-defined item**. An identifier **starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores and digits (0 to 9)**. C does not allow few punctuation characters such as #, @ and % within identifiers. As C is **a case-sensitive programming language**, Manpower and manpower are two different identifiers in C.

Here are some examples of acceptable identifiers

_sum stud_name a_123 myname50 temp j count123 printf

Keywords

The keywords are identifiers which have special **meaning in C** and hence cannot be used as constants or variables. There are **32 keywords in C Language**.

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

Variables

This is the most important concept in all languages. Variable is **a name given to a storage area that our programs can manipulate.** It has a **name, a value, a location and a type**. It also has something **called life, scope, qualifiers** etc. Variables are used to store information to be referenced and manipulated in a computer program. It provides a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of Variables as **containers that hold data or information**. **Purpose of Variables is to label and store data in memory, which then can be used throughout the program.**

Sometimes, the runtime makes variables with no names. These are called temporary variables. We cannot access them by name as we have no name in our vocabulary**. A variable has a type.** In 'C'**, type should be specified before a variable is ever used**. We define the variable with respect to type before using it. We cannot define the variable more than once.

Examples: int a; double b;

Rules for naming a Variable aka User defined Identifier:

A Variable name **cannot be same as the keyword** in c Language. A Variable name **should not be same as the standard identifiers** in c Language. A Variable name can **only have letters (both uppercase and lowercase letters), digits and underscore**. The **first letter** of a Variable should be either a **letter or an underscore**. There is no rule on how long a variable name aka user defined identifier can be. Variable name may run into problems in some compilers, if the variable name is longer than 31 characters. C is a **strongly typed language**, which means that the variable type cannot be changed once it is declared.

The type of a variable can never change during the program execution. The type decides what sort of values this variable can take and what operations we can perform. Type is a compile time mechanism. **The size of a value of a type is implementation dependent and is fixed for a particular implementation.**

Variable is associated with three important terms: **Declaration, Definition and Initialization**.

- **Declaration of a variable** is for informing to the compiler about the **name of the variable and the type of value it holds.** Declaration gives details about the properties of a variable.
- **Definition of a variable**: The variable gets stored. i.e., memory for the variable is allocated during the definition of the variable.

  If we want to only declare variables and not to define it i.e. we do not want to allocate memory, then the following declaration can be used.

  extern int a; // We will discuss this in Unit – 2
- **Initialization of a variable:** We can initialize a variable at the point of definition. **An uninitialized variable within a block has some undefined value**. 'C' does not initialize any default value.

```
int c = 100;
int d; // undefined value !!  variable can be assigned a value later in the code
int c = 200;
int c;  // Declaration again throws an error
d = 300;
```

Note: Assignment is considered as an expression in 'C' .

Data Types

In programming, data type is a classification that specifies type of value and what type of mathematical, relational or logical operations can be applied to it without causing an error. It deals with the **amount of storage to be reserved for the specified variable**. **Significance** of data types are given below:

1) Memory allocation
2) Range of values allowed
3) Operations that are bound to this type
4) Type of data to be stored

Size of a type:

The size required to represent a value of that type. Can be found using an operator called sizeof. **The size of a type depends on the implementation.** We should never conclude that the size of an int is 4 bytes. Can you answer this question –How many pages a book has? What is the radius of Dosa we get in different eateries?

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 returns the size of a variable/value/type. It can be applied to any data type, float type, pointer type variables When sizeof() is used with the data types, **it simply gives 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.

The size of a type is decided on an implementation based on efficiency. C standards follow the below Rules.

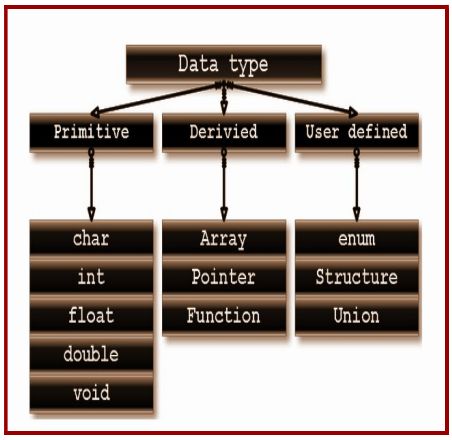
sizeof(short int)<=sizeof(int)<=sizeof(long int)<=sizeof(long long int)<= sizeof(float)<=sizeof(double)<=sizeof(long double)

Classification of Data types

Data types are categorized into **primary and non-primary (secondary) types**.

Primary ---> int, float, double, char, void

Secondary---> Derived and User-defined types



Data types can be extended using qualifiers:

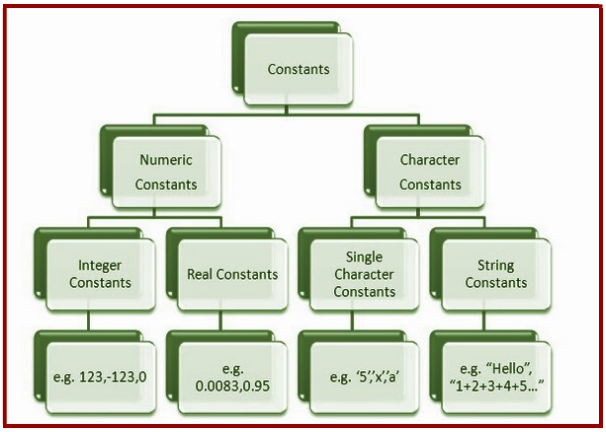
Size and Sign Qualifiers. - This will be discussed in Lecture-6.

Literals

Literals are the constant values assigned to the constant variables. There are four types of literals that exist in c programming:

1. **Integer literal**
2. **Float literal**
3. **Character literal**
4. **String literal –** Will be discussed in Unit – 2

Constant is basically a named memory location in a program that holds a single value throughout the execution of that program



Integer Literal: It is a numeric literal that represents only integer type values. Represents the value neither in fractional nor exponential part. Can be specified in the following three way.s

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 etc..

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: Must contain 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

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**. Example: the number is 3450000000000, and it can be expressed as 3.45e12 in an exponential form

Rules for creating an 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.

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. A character literal can be represented in the following ways:

It can be represented by specifying a single character within single quotes.

Example: 'x', 'y', etc.

We can specify the escape sequence character within single quotes to represent a character literal.

Example, '\n', '\t', '\b'.

We can also use the ASCII in integer to represent a character literal.

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.

Example, '\043', '\0x22'.

Think! What is the maximum and minimum integer value I can store in a variable?

The limits.h header file determines various properties of the various variable types. The macros defined in this header, limits the values of various variable types like char, int and long. These limits specify that a variable cannot store any value beyond these limits The values indicated in the table are implementation-specific and defined with the #define directive, but these values may not be any lower than what is given in the table.

Macro	Value	Description
CHAR_BIT	8	Defines the number of bits in a byte.
SCHAR_MIN	-128	Defines the minimum value for a signed char.
SCHAR_MAX	+127	Defines the maximum value for a signed char.
UCHAR_MAX	255	Defines the maximum value for an unsigned char.
CHAR_MIN	-128	Defines the minimum value for type char and its value will be equal to SCHAR_MIN if char represents negative values, otherwise zero.
CHAR_MAX	+127	Defines the value for type char and its value will be equal to SCHAR_MAX if char represents negative values, otherwise UCHAR_MAX.
MB_LEN_MAX	16	Defines the maximum number of bytes in a multi-byte character.
SHRT_MIN	-32768	Defines the minimum value for a short int.
SHRT_MAX	+32767	Defines the maximum value for a short int.
USHRT_MAX	65535	Defines the maximum value for an unsigned short int.
INT_MIN	-2147483648	Defines the minimum value for an int.
INT_MAX	+2147483647	Defines the maximum value for an int.
UINT_MAX	4294967295	Defines the maximum value for an unsigned int.
LONG_MIN	-9223372036854775808	Defines the minimum value for a long int.
LONG_MAX	+9223372036854775807	Defines the maximum value for a long int.

The **float.h** header file of the c Standard Library contains a **set of various platform-dependent constants** related to floating point values. These constants are proposed by ANSI C. They allow making more portable programs.

Coding Example_1:

```
#include<stdio.h>
#include<limits.h>
#include<float.h>
int main()
```

```
{
        int a = 8;
        char p = 'c';
        double d = 89.7;
        int e = 0113; // octal literal
        /*printf("%d\n",e);
        printf("%o\n",e);
        printf("%X\n",e);
        printf("%x\n",e);
        */
        //int h = 0xRG; // error
        int h = 0xA;
        //printf("%d",h);
        int i = 0b111;
        //printf("%d",i);
        //printf("%d %d %d %d\n",sizeof(int),sizeof(short int),sizeof(p),sizeof(long));
        //int g = 787878787888888787;
        int g = 2147483649;
        /*printf("%d\n",INT_MAX);
        printf("%d\n",INT_MIN);
        printf("%d\n",INT_MIN);
        printf("%d\n",INT_MIN);*/
        printf("%d\n",CHAR_MAX);        printf("%g\n",DBL_MAX);
        printf("%g\n",FLT_MAX);         printf("%d\n",g);
        return 0;
}
```

Question for you to think!

- What is header file?
- What is Library file?
- Is there any difference between the above two?

Happy Coding using Identifiers in C!!

Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #6
Simple Input function in C

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Simple Input functions in C

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

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Input and Output functions in C

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.

Simple Input functions in C

Two Types: Formatted Input function and Unformatted Input function

Formatted input function – scanf

scanf takes a format string as the first argument. The input should match this string**.**

int scanf(const char *format, ...);

where

int (integer) is the return type

format is a stringthatcontains the type specifier(s)

"..." (ellipsis) indicates that thefunction accepts a variable number of arguments; each argument must be amemory address where the converted result is written to. On success, the function writesthe result into the arguments passed.

- scanf("%d%d", &var1,&var2);// & -address operator is compulsory in scanf for all primary types

  When the user types, 23 11, 23 is written to var1 and 11 is to var2.

- scanf("%d,%d", &a, &b);

  scanf has a comma between two format specifiers, the input should have a commabetween a pair of integers.

- scanf("%d%d\n", &a, &b);

  It is not a good practice to have any kind of escape sequence in scanf. In the above code, it expects two integers. Ex: 20 30. Then if you press enter key, scanf does not terminate. You need to give some character other than the white space.

The function returns the following value:

>0 - The number of items converted and assigned successfully.

0 - No item was assigned.

<0 - Read error encountered or end-of-file (EOF) reached before any assignment was made.

- n = scanf("%d",&a);// If user enters 20, a becomes 20 and 1 is returned by the function.
- n = scanf("%d,%d",&a,&b); //If user enters 20 30, a becomes 20, value of b is undefinedand 1 is returned by the function

Unformatted input functions

A character in programming refers to a single symbol. A character in 'C' is like a very small integer having one of the 256 possible values. It occupies a single byte. We code English alphabets A as 65, B as 66, Z as 90, a as 97 and so on in a coding scheme called ASCII. To read a character from the keyboard, we could use

scanf("%c", x);We could also use **x = getchar();**

We prefer the second as it is developed only for handling a single char and therefore more efficient even though it is not generic. The unformatted input functions further have two categories: **Character functions and string functions**

Character functions: getchar(), getche(), getch()

String functions: gets() – This will be discussed in Unit - 2

Coding Example_1: To read two characters and display them.

```
int main()
{
        char ch = 'p'; // ch is a variable of type char and it can store only one character at a
        time.
        //Value for a character variable must be within single
        quote. // printf("Ch is %c\n",ch);// p
        //char ch1 = 'pqrs';// TERRIBLE
        CODEprintf("Ch is %c\n",ch);// s
        /*
        char x; char y;
        scanf("%c", &x); scanf("%c", &y);
        printf("x:%c y:%c\n", x, y);
        */
        x = getchar(); y = getchar(); putchar(x);
        putchar(y); printf("%d", y);
        return 0;
}
```

// If I enter P<newline>, x becomes P and y becomes newline

// scanf and printf : generic; not simple

// getchar and putchar : not generic; read /write a char; simple

// If I enter p<space>q, q will not be stored in y. Only space is stored. This can be avoided using fflush(stdin) function between two getchar function calls. This function clears the key board buffer.

// fflush(stdin) –windows

//_fpurge(stdin) – Linux based. Include <stdio_ext.h> in Linux based

Note: While using scanf for reading character input, care should be taken to handle White Spaces and Special Characters handling buffering problem.

Happy Coding using Input functions in C!!

Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #6
Qualifiers

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Qualifiers in C

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

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Qualifiers in C

Introduction

Qualifiers are keywords which are applied to the data types resulting in Qualified type. Applied to basic data types to alter or modify its sign or size.

Types of Qualifiers are as follows.

- **Size Qualifiers**
- **Sign Qualifiers**
- **Type qualifiers**

Size Qualifiers

Qualifiers are prefixed with data types to **modify the size of a data type** allocated to a variable. Supports two size qualifiers, **short and long. T**he Size qualifier is generally used with an integer type. In addition, double type supports long qualifier.

Rules regarding size qualifier as per ANSI C standard:

short int <= int <=long int

float <= double <= long double

Note: short int may also be abbreviated as short and long int as long. But, there is no abbreviation for long double.

Coding Example_1:

```
#include<stdio.h>
int main()
{       short int i = 100000; // cannot be stored this info with 2 bytes. So warning
        int j = 100000;  // add more zeros and check when compiler results in warning
        long int k = 100000;
        printf("%d %d %ld\n",i,j,k);
        printf("%d %d %d",sizeof(i),sizeof(j),sizeof(k));
        return 0;
}
```

Sign Qualifiers

Sign Qualifiers are used to specify the signed nature of integer types. It specifies whether a variable can hold a negative value or not. It can be used with int and char types

There are two types of Sign Qualifiers in C: s**igned and unsigned**

A signed qualifier specifies a variable which can hold both positive and negative integers

An unsigned qualifier specifies a variable with only positive integers.

Note: In a t-bit signed representation of n, the most significant (leftmost) bit is reserved for the sign, "0" means positive, "1" means negative.

Coding Example_2:

```
#include<stdio.h>
int main()
{
        unsigned int a = 10;
        unsigned int b = -10;   // observe this
        int c = 10;  // change this to -10 and check
        signed int d = -10;
        printf("%u %u %d %d\n",a,b,c,d);
        printf("%d %d %d %d",a,b,c,d);
        return 0;
}
```

Type Qualifiers

A way of expressing additional information about a value through the type system and ensuring correctness in the use of the data.

Type Qualifiers consists of two keywords i.e., **const** and **volatile.**

const

The **const** keyword is like a normal keyword but the only difference is that once they are defined, their values can't be changed. They are also called as literals and their values are fixed.

Syntax: const data_type variable_name

Coding Example_3:

```
#include <stdio.h>
int main()
{       const int height = 100; /*int constant*/
        const float number = 3.14; /*Real constant*/
        const char letter = 'A'; /*char constant*/
        const char letter_sequence[10] = "ABC"; /*string constant*/
        const char backslash_char = '\?'; /*special char cnst*/
        //height++; //error
        printf("value of height :%d \n", height );
        printf("value of number : %f \n", number );
        printf("value of letter : %c \n", letter );
        printf("value of letter_sequence : %s \n", letter_sequence);
        printf("value of backslash_char : %c \n", backslash_char); return 0;
}
```

Output:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

Note: In detail explanation of Pointers, constant Pointer and Pointer to constant will be discussed in Unit-2 wrt the Functions Topic.

volatile

It is intended to prevent the compiler from applying any optimizations. Their values can be changed by the code outside the scope of current code at any time. A type declared as volatile can't be optimized because its value can be easily changed by the code. The declaration of a variable as volatile tells the compiler that the variable can be modified at any time by another entity that is external to the implementation, for example: by the operating system or by hardware. **Syntax:** volatile data_type variable_name

Applicability of Qualifiers to Basic Types

The below table helps us to understand which Qualifier can be applied to which basic type of data.

No.	Data Type	Qualifier
1.	char	signed, unsigned
2.	int	short, long, signed, unsigned
3.	float	No qualifier
4.	double	long
5.	void	No qualifier

Happy Coding with Qualifiers!

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Problems on basic constructs in C and control structures

Course objectives:

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

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

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

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C contructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

List of problems

1. You are transporting some boxes through a tunnel whose height is only 41 feet. Given the length, width, and height of the box, calculate the volume of those boxes and check if they pass through the tunnel.

   Ex: Input = 5, 5, 5 Output = 125

   Input = 1, 2, 40 Output = 80

   Input = 10, 5, 41 Output = Can't Pass!

2. Given the number of rows, print a hollow diamond using star symbol: Ex: Input = 5

3. Check whether the given number is divisible by the sum of its digits. Display appropriate message
4. Write a C program to find the eligibility of admission for a professional course based on the following criteria: Eligibility Criteria : Marks in Maths >=65 and Marks in Physics >=50 and Marks in Chemistry>=55 and Total in all three subjects >=190 or Total in Maths and Physics >=140
5. Write a C program to print the prime numbers within the given input range. Ex. if user gives 10 and 50 as the input range, then the program must display all the prime numbers between the range 10 and 50 (i.e. 11 13 17 19 23 29 31 37 41 43 47)
6. Construct a menu-based calculator to perform arithmetic operations (Add, Subtract, Multiply,Divide) on complex numbers.
7. Given a number N, check whether it is a palindrome numbers or not. Ex. 121 , 3443 are palindromes whereas 123,4531 are not plaindromes
8. Write a program to print all odd numbers between the two limits "m" and "n" (both m and n is given as an integer inputs by the user and m<n) and print all odd numbers excluding those which are divisible by 3 and 5.

9. Implement a converter program to obtain an integer from a hexadecimal byte.
10. Given a number N, check whether the nth bit of a number, N is set to 1 or not. Input the N value from the user.

A few solutions

1. You are transporting some boxes through a tunnel whose height is only 41 feet. Given the length, width, and height of the box, calculate the volume of those boxes and check if they pass through the tunnel.

   Ex: Input = 5, 5, 5 Output = 125

   Input = 1, 2, 40 Output = 80

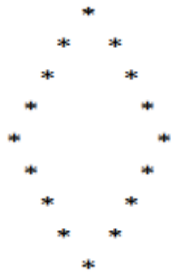
   Input = 10, 5, 41 Output = Can't Pass!

   Solution:

```
#include<stdio.h>
int main()
{
   int h, w, l;
   printf("Enter the length, width, height: ");
   scanf("%d %d %d", &l, &w, &h);
   if(h >= 41)
   {
      printf("Can't pass!");

   }
   else
   {
            int volume = l*w*h;
            printf("%d", volume);
   }
   return 0;
}
```

2. Given the number of rows, print a hollow diamond using star symbol: Ex: Input = 5



Solution:

```
#include<stdio.h>
int main()
{
  int n;
  printf("Enter number of rows: ");
  scanf("%d",&n);
  for(int i=1; i<= n; i++)
  {
    for(int j=i; j<= n; j++)
      printf(" ");
     for(int k = 1; k <= 2*i-1; k++)
    {
      if(k == 1 || k == (2*i-1))
          printf("*");
      else
          printf(" ");
    }
    printf("\n");
  }
  for(int i = n-1; i>=1; i--)
  {
```

```
    for(int j = n; j>=i; j--)
      printf(" ");

    for(int k = 1; k <= 2*i-1; k++)
    {
      if(k == 1 || k == 2*i-1)
          printf("*");
      else
          printf(" ");
    }
    printf("\n");
  }
  return 0;
}
```

3. Check whether the given number is divisible by the sum of its digits. Display appropriate message (Divisible or Not Divisible)

Solution:

```
#include<stdio.h>
 int main()
{
   int n;
   printf("Enter the number:");
   scanf("%d", &n);
   int temp = n;
   //finding the sum of digits of the number
   if(n<=0)
      printf("invalid");
   else
   {
   int sum = 0;
   while(n)
```

```
{
    int r = n % 10;
    sum = sum + r;
    n = n / 10;
}
//check if sum of digits divides the number
if(temp%sum == 0)
    printf("Divisible");
else
    printf("Not Divisible");
}
return 0;
}
```

4. Write a C program to find the eligibility of admission for a professional course based on the following criteria: Eligibility Criteria : Marks in Maths >=65 and Marks in Physics >=50 and Marks in Chemistry>=55 and Total in all three subjects >=190 or Total in Maths and Physics >=140

Solution:

```
#include <stdio.h>
int main()
{
   int p,c,m,total,math_phy;
  printf("Input the marks obtained in Physics, Chemistry, Math :");
  scanf("%d %d %d",&p, &c, &m);
  total = m + p + c;
  math_phy = m + p;
  printf("Total marks of Maths, Physics and Chemistry : %d\n",m+p+c);
  printf("Total marks of Maths and  Physics : %d\n",m+p);
  if (m>=65)
       if(p>=50)
          if(c>=55)
```

```
            if((total)>=190||(math_phy)>=140)
               printf("The  candidate is eligible for admission.\n");
            else
              printf("The candidate is not eligible.\n");
             else
          printf("The candidate is not eligible.\n");
           else
         printf("The candidate is not eligible.\n");
       else
        printf("The candidate is not eligible.\n");
        return 0;
    }
```

5. Write a C program to print the prime numbers within the given input range. Ex. if user gives 10 and 50 as the input range, then the program must display all the prime numbers between the range 10 and 50 (i.e. 11 13 17 19 23 29 31 37 41 43 47)

```
#include <stdio.h>
int main()
{
   int low, high, i, flag;
   printf("Enter two numbers(intervals): ");
   scanf("%d %d", &low, &high);

   printf("Prime numbers between %d and %d are: ", low, high);

   while (low < high)
   {
      flag = 0;

      for(i = 2; i <= low/2; ++i)
      {
         if(low % i == 0)
         {
            flag = 1;
            break;
         }
      }

      if (flag == 0)
         printf("%d ", low);
```

```
    ++low;
  }

  return 0;
}
```

6. Construct a menu-based calculator to perform arithmetic operations (Add, Subtract, Multiply,Divide) on complex numbers.

   Solution:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int choice, a, b, c, d;
 do
 {
  printf("1. Addition\n");
  printf("2. Subtraction\n");
  printf("3. Multiplication\n");
  printf("4. Divide\n");
  printf("Enter your choice\n");
  scanf("%d", &choice);
  if (choice > 4 || choice < 1)
    exit(0);          // terminates the program
  printf("Enter a and b where a + ib is the first complex number.");
  scanf("%d %d", &a, &b);
  printf("Enter c and d where c + id is the second complex number.");
  scanf("%d %d", &c, &d);
  /*
  if (choice == 1)
  {
   int real = a+c;
```

```
  int img = b+d;
  if (img >= 0)
    printf("Sum = %d + %di\n", real, img);
  else
    printf("Sum = %d%di\n", real, img);
}
else if (choice == 2)
{
  int real = a-c;
  int img = b-d;

  if (img >= 0)
    printf("Difference = %d + %di", real, img);
  else
    printf("Difference = %d %di", real, img);
}
else if (choice == 3)
{
  int real = a*c - b*d;
  int img = b*c + a*d;

  if (img >= 0)
    printf("Product = %d + %di", real, img);
  else
    printf("Product = %d %di", real, img);
}
else if (choice == 4)
{
  if (c == 0 && d == 0)
    printf("Division by 0 + 0i isn't allowed.");
  else
```

```
{
 int x = a*c + b*d;
 int y = b*c - a*d;
 int z = c*c + d*d;

 if (x%z == 0 && y%z == 0)
 {
  if (y/z >= 0)
    printf("Division of the complex numbers = %d + %di", x/z, y/z);
  else
    printf("Division of the complex numbers = %d %di", x/z, y/z);
 }
 else if (x%z == 0 && y%z != 0)
 {
  if (y/z >= 0)
    printf("Division of two complex numbers = %d + %d/%di", x/z, y, z);
  else
    printf("Division of two complex numbers = %d %d/%di", x/z, y, z);
 }
 else if (x%z != 0 && y%z == 0)
 {
  if (y/z >= 0)
    printf("Division  = %d/%d + %di", x, z, y/z);
  else
    printf("Division = %d %d/%di", x, z, y/z);
 }
 else
 {
  if (y/z >= 0)
    printf("Division = %d/%d + %d/%di",x, z, y, z);
  else
```

```
        printf("Division = %d/%d %d/%di", x, z, y, z);
    }
  }
}*/
int real, img;
switch(choice)
{
case 1:
  real = a+c;
  img = b+d;
  if (img >= 0)
    printf("Sum = %d + %di\n", real, img);
  else
    printf("Sum = %d%di\n", real, img);
  break;

case 2:
  real = a-c;
  img = b-d;
  if (img >= 0)
    printf("Difference = %d + %di\n", real, img);
  else
    printf("Difference = %d %di\n", real, img);
  break;

case 3:
  real = a*c - b*d;
  img = b*c + a*d;
  if (img >= 0)
    printf("Product = %d + %di\n", real, img);
  else
```

```
    printf("Product = %d %di\n", real, img);
  break;
case 4:
  if (c == 0 && d == 0)
    printf("Division by 0 + 0i isn't allowed.");
  else
  {
    int x = a*c + b*d;
    int y = b*c - a*d;
    int z = c*c + d*d;
    if (x%z == 0 && y%z == 0)
    {
      if (y/z >= 0)
        printf("Division of the complex numbers = %d + %di", x/z, y/z);
      else
        printf("Division of the complex numbers = %d %di", x/z, y/z);
    }
    else if (x%z == 0 && y%z != 0)
    {
      if (y/z >= 0)
        printf("Division of two complex numbers = %d + %d/%di", x/z, y, z);
      else
        printf("Division of two complex numbers = %d %d/%di", x/z, y, z);
    }
    else if (x%z != 0 && y%z == 0)
    {
      if (y/z >= 0)
        printf("Division  = %d/%d + %di", x, z, y/z);
      else
        printf("Division = %d %d/%di", x, z, y/z);
    }
```

```
        else
        {
          if (y/z >= 0)
            printf("Division = %d/%d + %d/%di",x, z, y, z);
          else
            printf("Division = %d/%d %d/%di", x, z, y, z);
        }
       }
       break; Breaks come out of current loop
      }
  }while(choice<=4);
  return 0;
}
```

HAPPY CODING!

Solve other problems. If any issues, please contact sindhurpai@pes.edu