



**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.

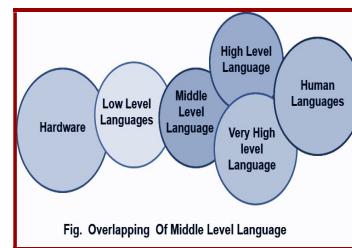
Q3. Why so many Programming Languages?

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 should take 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 machine's basic instruction set
Examples: Java, Python	C, C++	Assembler



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 #2
Intro to C in the era of AI, Features, Programming
Environment***

**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: Intro to C in the era of AI, Features, Programming Environment

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 C in the era of AI

C has been a foundational programming language for decades, known for its efficiency, portability, and close-to-hardware capabilities. Despite the rise of high-level languages like Python, C remains relevant in AI development, especially in **performance-critical areas**.

Why C still matters?

- **Performance & Efficiency** – AI applications often require high-speed computation. C provides faster execution than interpreted languages, making it crucial for optimizing AI models.
- **Embedded AI & Edge Computing** – AI models are increasingly deployed on IoT devices, robotics, and embedded systems where C is widely used due to its lightweight nature.
- **Interfacing with AI Frameworks** – Many AI libraries (e.g., TensorFlow, PyTorch) have C or C++ backends, allowing deep integration for performance enhancements.
- **Parallel Processing & GPU Acceleration** – AI applications leverage parallelism, and C interfaces with CUDA, OpenCL, and other GPU libraries for efficient computations.

Use Cases of C in AI

- **AI Accelerators & Hardware Optimization** – Writing low-level code for AI chips and FPGAs.
- **Memory Optimization in Large Models** – Managing memory more efficiently than high-level languages.
- **Embedded AI Applications** – AI-powered sensors, real-time inference on microcontrollers.
- **Algorithm Development** – Implementing AI algorithms with custom optimizations for speed and memory use.

While **Python dominates AI research**, learning C can be highly beneficial for optimizing performance and efficiency. It enables AI enhancements through C extensions in Python, such as NumPy and SciPy, allowing faster computations. C is also valuable for developing AI applications in low-resource environments, including embedded systems and IoT devices. Additionally, it provides a deeper understanding of computation and memory management, essential for building high-performance AI solutions. In conclusion, **combining C with AI-centric languages like Python enables developers to build efficient and scalable AI solutions.**

Features of C Language

Simple and Efficient:

The **basic syntax style** of implementing C language is very simple and easy to learn. This makes the language easily comprehensible and enables a programmer to redesign or create a new application. C is usually used as an introductory language to introduce **programming** to school students because of this feature.

Fast:

It is a well-known fact that **statically typed programming languages** are faster than **dynamic ones**. C is a **statically typed** programming language, which gives it an edge over other dynamic language. Also, unlike Java and Python, which are interpreter-based, C is a **compiler-based** program. This **makes the compilation and execution of codes faster**. Another factor that makes C fast is the availability of only the essential and required features. Newer programming languages come with numerous features, which increase functionality but reduce efficiency and speed. Since C offers limited but essential features, the headache of processing these features reduces, increasing speed.

Portability:

Another feature of the C language is portability. To put it simply, C programs are **machine-independent** which means that you can run the fraction of a code created in C on

various machines with none or some machine-specific changes. Hence, it provides the functionality of using a single code on multiple systems depending on the requirement.

Extensibility:

You can easily (and quickly) extend a C program. This means that if a code is already written, you can add new features to it with a few alterations. Basically, it allows adding new features, functionalities, and operations to an existing C program.

Function-Rich Libraries:

C comes with an extensive set of libraries with several built-in functions that make the life of a programmer easy. Even a beginner can easily code using these built-in functions. You can also create your user-defined [functions](#) and add them to C libraries. The availability of such a vast scope of functions and operations allows a programmer to build a vast array of programs and applications.

Dynamic Memory Management:

One of the most significant features of C language is its support for dynamic memory management (DMA). It means that you can utilize and manage the size of the [data structure in C](#) during runtime. C also provides several predefined functions to work with memory allocation. For instance, you can use the `free()` function to free up the allocated memory at any time. Similarly, there are other functions such as `malloc()`, `calloc()`, and `realloc()` to perform operations on data structure and memory allocations.

Modularity with Structured Language:

C is a general-purpose structured language. This feature of C language allows you to break a code into different parts using functions which can be stored in the form of libraries for future use and reusability. Structuring the code using functions increases the visual appeal and makes the program more organized and less prone to errors.

Mid-Level Programming Language:

Although C was initially developed to do only low-level programming, it now also supports the features and functionalities of high-level programming, making it a mid-level language. And as a mid-level programming language, it provides the best of both worlds. For

instance, C allows direct manipulation of hardware, which high-level programming languages do not offer.

Pointers:

With the use of pointers in C, you can directly interact with memory. As the name suggests, pointers point to a specific location in the memory and interact directly with it. Using the C pointers, you can operate with memory, arrays, functions, and structures.

Recursion:

C language provides the feature of recursion. Recursion means that you can create a function that can call itself multiple times until a given condition is true, just like the loops. Recursion in C programming provides the functionality of code reusability and backtracking.

C Programming Environment

The C programming environment consists of several essential components that enable writing, compiling, debugging, and executing C programs efficiently. It includes a text editor, a compiler, a linker, a debugger, and a runtime environment.

Text Editor – Used to write and edit C source code files (.c). Common editors include VS Code, Sublime Text, and Notepad++.

Compiler – Translates C source code into machine code. Popular C compilers include **GCC** (GNU Compiler Collection), **Clang**, and **MSVC** (Microsoft Visual C++).

Preprocessor – Handles directives like #include, #define, and #ifdef before compilation.

Assembler & Linker – The assembler converts assembly code to machine code, while the linker combines multiple object files and links required libraries to create an executable.

Debugger – Tools like **GDB** (GNU Debugger) and **LLDB** help identify and fix errors in the code by allowing step-by-step execution and inspecting variable states.

Standard Libraries – C relies on standard libraries such as stdio.h, stdlib.h, and math.h for essential functions like input/output operations and mathematical computations.

Integrated Development Environments (IDEs) – IDEs like Code::Blocks, Dev-C++, Turbo C++, and Eclipse provide an all-in-one environment for writing, compiling, debugging, and executing C programs.

Execution Environment – The compiled C program runs in an operating system environment such as Windows, Linux, or macOS, interacting with system resources through system calls.

We will start exploring C Constructs in the upcoming lecture notes by solving some of the interesting Problems. 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.**

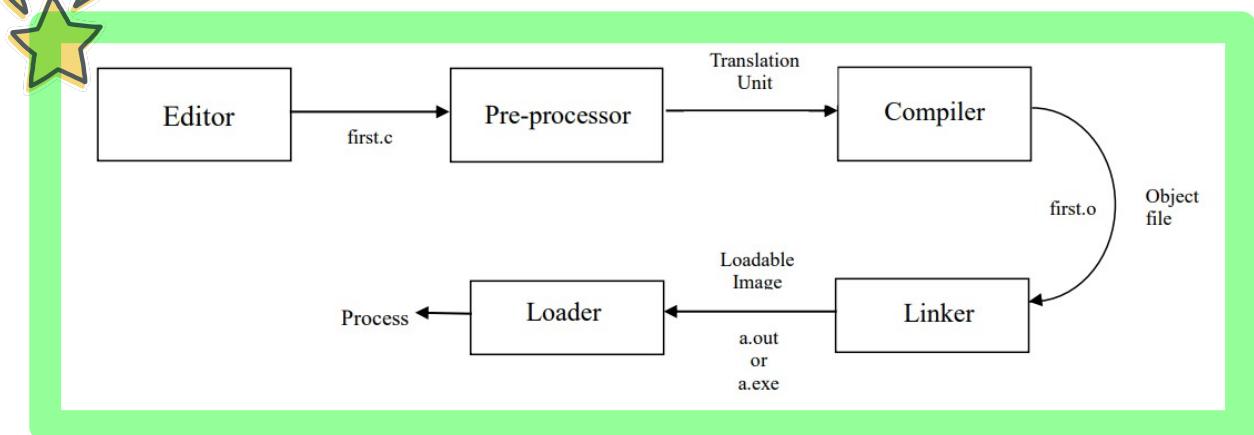
<https://www.youtube.com/watch?v=sXW2VLrQ3Bs>

- 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 will print 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

```

Comments

Single-line comments can be made with `//`

Multi-line comments can be made between `/* [text] */`

Terminal Commands

```

gcc filename.c -o filename.exe
./filename.exe (this runs the code)

```

`-o` allows us to give the executable a desired output name.

Library Functions

- `%d` → integers
- `%f` → floats
- `%c` → characters

Always give characters within single quotes.

```

// example
#include <stdio.h>
int main()
{
    int a = 2;
    printf("The given integer is %d", a);
    return 0;
}

```

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!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #4
Program Structure, Characteristics, Errors***

**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: Program Structure, Characteristics, Errors

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

Characteristics of a C Program

- The language is **case sensitive**.
- The program is **immune to indentation**. ‘C’ follows the free format source code concept. The way of presentation of the program to the compiler has no effect on the logic of the program. However, we should properly indent the program so that the program becomes readable for humans. When you prepare a new dish, I am sure you will also consider how you present it.
- The program has a **well-defined entry point**. Every program has to have an entry point called **main**, which is the starting point of execution.

```
int main()           // execution begins here
{
    printf("Hello Friends");
    return 0;
}
```

Think: Why the main function has the return type int? Why not something else ?

Can we have something else? Is it a good practice?

- A program in ‘C’ normally has one or more **pre-processor directives**. These start with a **symbol #**. One of the directives is “include”.

- `#include <stdio.h>`

The above line asks the translator to find the file whose name is stdio.h at some location in our file system and read and expand it at that point in the ‘C’ code.

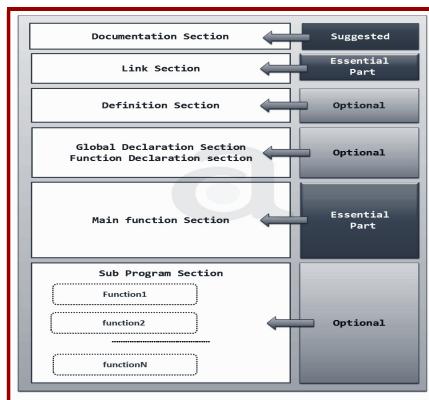
- A program in ‘C’ has to have a **function called main**. It is **invoked(called)** from some **external agency** – like **ourshell**. It is the duty of this function to tell the caller whether it succeeded or failed. **So, by convention, main returns 0 on success and non-zero on failure**.

Refer to coding example_1 for more details.

- A block contains a number of statements. **Each statement ends in a semicolon**. This symbol ; plays the **role of statement terminator**

Program structure

Consists of different sections of which few are optional and other few are mandatory sections. Refer to the below diagram.



Documentation Section: Optional but always good to have. Usually contains the information about the code in terms of comments

// Single line comment

/* multi line/ block Comments */

Link Section: Provides instructions to the compiler to link functions from the system library such as using the #include directive. Basically Header files that are required to execute a C program are included in the Link section.

Definition Section: This is the optional section

Contains the definitions of functions before calling the functions

Global Declaration Section: This is the optional section

Contains global variables declaration and function declaration(no body, only prototype/signature)

Main function Section: Every C program must have one main function section. Contains two parts: the declaration part and the executable part. Declaration part declares all the variables used in the executable part. Executable part: There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

Sub program section: Optional section and contains user defined functions. We will discuss in Unit – 2.

Coding example_1: // Importance of return value in main

```
int main() // this code is saved as main.c
{
    printf("welcome to main function in C");    return 0; }
```

Execute the above code using below steps:

```
gcc -c main.c // press enter - main.o gets created if no error
gcc main.o // press enter - executable gets created if no error
a.exe // prints welcome to main function in C
```

If someone wants to know whether the recently executed process is executed successfully or not, this return value is very helpful.

After you get the output, use below commands to know what the return value of recently executed process.

```
echo %ERRORLEVEL% // in windows
echo $? // in linux based systems
```

Must return 0 to represent the success[by convention 0 is success]. Modify the code as below.

```
int main() // this code is saved as main.c
{
    printf("welcome to main function in C");
}
```

Repeat the same steps as mentioned in the first case, there is no way to get the success or failure info about the recently executed process. Results in some random value other than 0.

Coding example_2: // Compilation using different C Standards

__STDC_VERSION__: A macro defined in stdio.h

To understand better, use codes using –E option of GCC. gcc -E

// file saved as first.c

```
int main()
{
    printf("C standard is %d\n",__STDC_VERSION__);
    return 0;
}
```

Preprocessing and Compilation using C99: **gcc -std=c99 -c first.c**

Linking remains the same: **gcc first.o**

Execute: **a.exe OR ./a.out**

Output: **C standard is 199901**

Preprocessing and Compilation using C11: **gcc -std=c11 -c first.c**

Linking remains the same: **gcc first.o**

Execute: **a.exe OR ./a.out**

Output: **C standard is 201112**

Preprocessing and Compilation using C89: **gcc -std=c89 -c first.c**

Results in Compile time Error as this macro is not available in C89 standard

Error(s) in C Code

Error is 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. Error is an illegal operation performed by the programmer which results in abnormal working of the program. Error is also known as bugs in the world of programming that may occur unwillingly which prevent the program to compile and run correctly as per the expectation of the programmer. We deal with four kinds of errors as of now.

Compile time Error

Link time Error

Run time Error

Logical Error

Compile time Error

Deviation from the rules of the Language or **violating the rules of the language** results in **compile time Error**. Whenever there is Compile Time Error, **object file(.o file)** will **not be created**. Hence linking is not possible. When the programmer does not follow the syntax of any programming language, then the compiler will throw the **Syntax Error** which **are also compile time errors**. Syntax Errors are easy to figure out because the compiler highlights the line of code that caused the error.

Coding Example_3:

```
#include<stdio.h>

int main()
{
    printf("hello friends); // " missing at the end
    printf("%d",12/2) // ; missing at the end of statement
    return 0;
}
```

Link time Error

Errors encountered when the executable file of the code cannot be generated even though the code gets compiled successfully. This usually happens due to the mismatch between Function call and function definitions. When you compile the code, object file gets created. Sometimes with a warning. During linking, definition of `Printf` is not available in the included file. Hence Link time Error. No loadable object will be created due to this error. This Error is generated when a different object file is unable to link with the main object file.

Coding Example_4:

```
#include<stdio.h>

int main()
{
    printf("hello friends"); // P is capital
    printf("%d",12/2);
    return 0;
}
```

Run time 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 the code is compiled, object file(.o file) will be created. Then during linking, executable will be created. When you run a loadable image, code terminates throwing an error. Generally occurs due to some illegal operation performed in the program. Examples of some illegal operations that may produce runtime errors are: Dividing a number by zero, trying to open a file which is not created and lack of free memory space.

Coding Example_5:

```
#include<stdio.h>

int main()
{
    printf("hello friends");
    printf("%d",12/0); // Observe 12/0
    printf("bye friends");
    return 0; }
```

Logical Error:

Even though the code seems error free, the output generated is different from the expected one. Logical error occurs when the code runs completely but the **expected output is not same as the actual output.**

Coding Example_6:

```
#include<stdio.h>
int main ()
{
    printf("%d",12/21); //By Mistake entered 21 instead of 2
    return 0;
}
```

Warning(s) in C:

Warnings are **diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error or crash in future**. 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.

Why the warning must be enabled?

C and C++ compilers are notoriously bad at reporting some common programmer mistakes by default, such as:

- forgetting to initialise a variable
- forgetting to return a value from a function
- arguments in printf and scanf families not matching the format string
- a function is used without being declared beforehand (C only)

Most compilers understand options like **-Wall, -Wpedantic and -Wextra**. -Wall is essential and all the rest are recommended (note that, despite its name, **-Wall only enables the most important warnings**, not all of them). These options can be used separately or all together.

Error vs warning

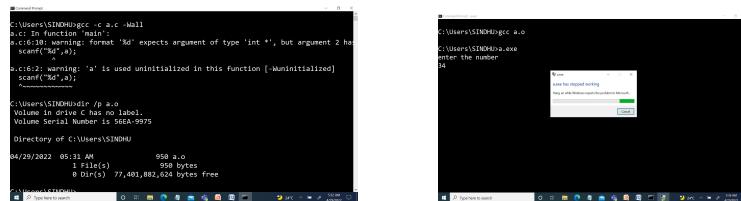
Errors report problems that make it impossible to compile your program. GCC reports errors with the source file name and line number where the problem is apparent. Warnings report **other unusual conditions in your code that may indicate a problem, although compilation can proceed**.

Coding Example_7:

```
#include<stdio.h>
int main()
{
    int a;
    printf("enter the number\n");
    scanf("%d",a);
    printf("you entered %d\n",a);
    return 0;
}
```

Without -wall option, there is no warning or error. But results in Runtime Error when the code is run using the executable.

The above code compilation using –Wall is as below. This creates the object file. But at runtime, it fails to execute.



Happy Error Finding!!



**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.

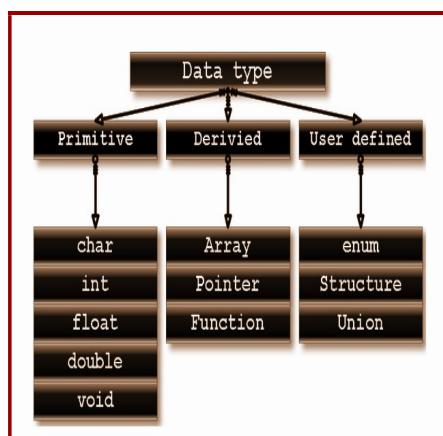
$$\text{sizeof(short int)} \leq \text{sizeof(int)} \leq \text{sizeof(long int)} \leq \text{sizeof(long long int)} \leq \text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$$

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:

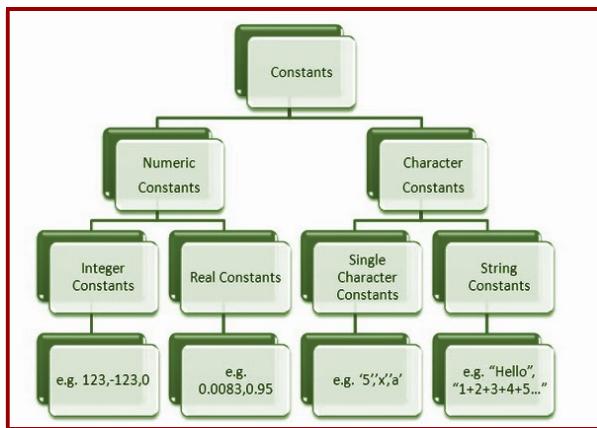
Size and Sign Qualifiers. - This will be discussed in Lecture-6.

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 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 OX** 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!!

Concept_Segment	Sample C Program
Integer Data Types	<pre data-bbox="620 210 1404 700"> /* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate integer data types in C #include <stdio.h> int main() { int a = 10; short b = 20; long c = 30000; long long d = 4000000000; printf("int: %d, short: %hd, long: %ld, long long: %lld\n", a, b, c, d); return 0; } </pre>
Floating-Point Data Types	<pre data-bbox="620 768 1404 1193"> /* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate floating-point data types in C #include <stdio.h> int main() { float a = 10.5; double b = 20.55; long double c = 30.5555L; printf("float: %f, double: %lf, long double: %Lf\n", a, b, c); return 0; } </pre>

Implicit Type Conversion	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate implicit type conversion #include <stdio.h> int main() { int a = 5; float b = a; // Implicit conversion from int to float printf("Converted value: %f\n", b); return 0; }</pre>
Explicit Type Conversion	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate explicit type conversion #include <stdio.h> int main() { float a = 5.67; int b = (int) a; // Explicit conversion from float to int printf("Original: %f, Converted: %d\n", a, b); return 0; }</pre>

Arithmetic Operations	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate arithmetic operations #include <stdio.h> int main() { int a = 10, b = 4; printf("Addition: %d\n", a + b); printf("Subtraction: %d\n", a - b); printf("Multiplication: %d\n", a * b); printf("Division: %d\n", a / b); printf("Modulus: %d\n", a % b); return 0; }</pre>
Size of Data Types	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate sizeof operator #include <stdio.h> int main() { printf("Size of int: %lu bytes\n", sizeof(int)); printf("Size of float: %lu bytes\n", sizeof(float)); printf("Size of double: %lu bytes\n", sizeof(double)); printf("Size of long long: %lu bytes\n", sizeof(long long)); return 0; }</pre>

Limits of Data Types	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate limits of numerical data types #include <stdio.h> #include <limits.h> #include <float.h> int main() { printf("Min int: %d, Max int: %d\n", INT_MIN, INT_MAX); printf("Min float: %e, Max float: %e\n", FLT_MIN, FLT_MAX); printf("Min double: %e, Max double: %e\n", DBL_MIN, DBL_MAX); return 0; }</pre>
Random Number Generation	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Generate random numbers #include <stdio.h> #include <stdlib.h> #include <time.h> int main() { srand(time(0)); printf("Random number: %d\n", rand() % 100); return 0; }</pre>

Mathematical Operations	<pre>/* PES University B.Tech 2nd Semester A and N Section - EC Campus */ // Demonstrate mathematical operations #include <stdio.h> #include <math.h> int main() { double a = 9.0; printf("Square root: %f\n", sqrt(a)); printf("Power: %f\n", pow(a, 2)); printf("Log: %f\n", log(a)); return 0; }</pre>
-------------------------	---

Swapping Two Numbers Without Third Variable

```
/* PES University B.Tech 2nd Semester A and N Section - EC Campus */

// Swap two numbers without using a third variable
#include <stdio.h>

int main()
{
    int a = 5, b = 10;

    printf("Before swap: a = %d, b = %d\n", a, b);

    a = a + b;
    b = a - b;
    a = a - b;

    printf("After swap: a = %d, b = %d\n", a, b);

    return 0;
}
```



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #5
Simple output functions 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 output 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 Output functions in C

Two Types: Formatted Output function and Unformatted output function

Formatted Output function - `printf`

The C library function `printf()` sends formatted output to `stdout`. A predefined function in "stdio.h" header file. By using this function, we can print the data or user defined message on console or monitor. **On success, it returns the number of characters successfully written on the output. On failure, a negative number is returned.**

`int printf(const char *format, ...)`

- `printf("hello Friends");`

The first argument to `printf` is a string. The output depends on this string. String literals in 'C' are enclosed in double quotes.

- `printf("hello ", "friends"); // hello`

Output depends on the first string. As there is no interpretation of the **second parameter, it would not appear in the output.**

- `printf("hello %s\n", "friends\n"); // hello friends`

The presence of `%s` in the first string makes the function `printf` look for the next parameter and interpret it as a string.

- `printf("%s %s %s %s\n", "one", "two", "three", "four");`

The function **printf takes variable number of arguments**. All arguments are interpreted as strings because the number of `%s` matches the number of strings following the first string.

- `printf("%s %s%s %s\n", "one", "two", "three");`

NO! we are in for trouble. There is no argument for the 4th `%s` in the format string. So, `printf` tries to interpret it as a string. If we are lucky, the program will crash. We have an “**undefined behaviour**”. C does no checking at runtime!

- `printf("%5d and %5d is %6d\n", 20, 30, 20 + 30); // Discussed in detail:Format String Arguments to printf can be expressions` – Then the expressions are evaluated and their values are passed as arguments.

`%d` : indicates to `printf` to interpret the next parameter as an integer.

`%5d` : tells `printf` to display the integer using 5 character width.

- `int a = 20; int n = printf("%d\n", a); // printf executes and returns # of characters successfully printed on the terminal`

`printf("n is %d", n); //2`

- `printf("what : %d\n", 2.5);`

GOD if any should help the programmer who writes such code!! **undefined behaviour**. In ‘C’, type is a compile mechanism and value is a runtime mechanism. As the goal of ‘C’ is efficiency, only values are stored at runtime. There is no way to infer the type by looking at the bit pattern. **There is no translator at runtime in ‘C’**

Note: All the code that executes at runtime should be compiled and cannot be added at runtime.

Let us look at our present example. `printf("what : %d\n", 2.5);`

The function `printf` takes varying number of arguments. In such cases, the compiler cannot match arguments and parameters for type. So, the compiler might emit some warning (if it knows what `printf` expects) –but cannot throw errors.

As the compiler does not know that 2.5 should be converted to an integer, it does not do any conversion. So, the compiler puts the value of 2.5 the way it is expected to be stored –as per the standard IEEE 754. This standard uses what is called mantissa exponent format to store fractional (floating point) values.

At runtime, `printf` tries to interpret the bit pattern as an integer when it encounters the format `%d`. At runtime, no conversion can occur –we do not even know that what is stored as a bit pattern is a floating point value –Even if we know, there is no way to effect a conversion as we do not have a compiler at that point. We end up getting some undefined value. So, in 'C', you get what you deserve.

Format string

Used for the beautification of the output. The format string is of the form

`% [flags] [field_width].[precision] conversion_character`

where components in brackets [] are optional. The minimum requirement is `%` and a conversion character (e.g. `%d`). Below conversion characters for each type of data.

<code>%d : dec int</code>
<code>%x : hex int</code>
<code>%o : oct int</code>
<code>%f : float</code>
<code>%c : char</code>
<code>%p : address</code>
<code>%lf : double</code>
<code>%s : string –[Will be discussed in detail in Unit-2]</code>
<code>float c = 2.5;</code>

```

printf("%4.2f\n",c);// width.precision , f is type

printf("%2.1f\n",c);// data will not be lost

printf("%-5d %d\n",16,b);//-for left justification

```

Escape sequences

Represented by two key strokes and represents one character.

\n	newline--move cursor to next line
\t	tab space
\r	carriage return --Move cursor to the beginning of that line
\a	alarm or bell
\"	double quote
\'	single quote
\\\	black slash
\b	back space

```

printf("rama\nkrishna\bheema\n");//ramakrishnabheema

printf("rama\rkri\rbh");// bhia \r works differently in different platforms.

//We will not discuss more about this

printf("rama\tbheema\n");// ramabheema printf("\\"mahabharatha\\\"");
//"\mahabharatha"

printf("rama\b\bbheema");// rabheema

```

Unformatted output functions

These functions are not capable of controlling the format that is involved in writing the available data. Hence these functions constitute the most basic forms of output. The display of output isn't allowed in the user format, hence we call these functions as unformatted functions. The unformatted output functions further have two categories:

Character functions and string functions

Character functions: putchar – This will be discussed in Lecture-15

String functions: puts – This will be discussed in Unit - 2

Happy Coding using Output functions!!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #7 & #8
Operators and Expressions***

**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: Operators and Expressions

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

Operators in C

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. Operator is a symbol used for calculations or evaluations

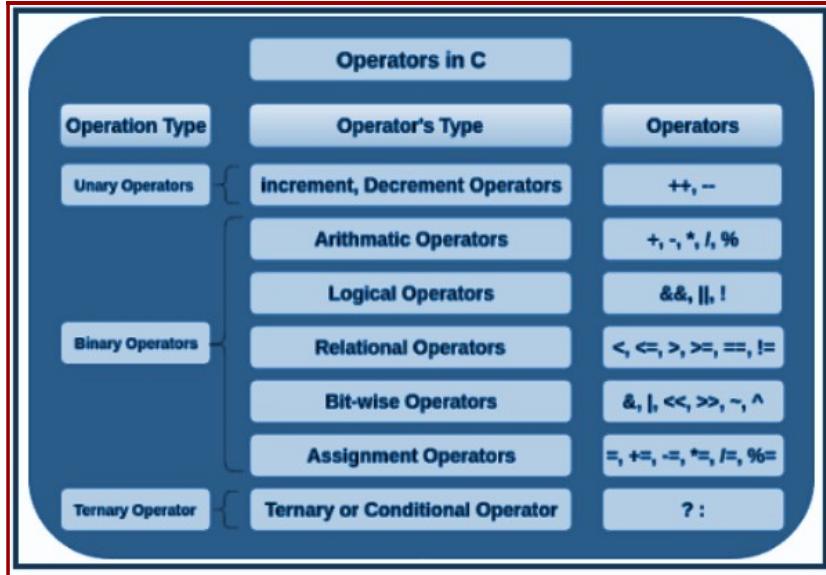
Classification of Operators:

Based on the number of operands.

- 1) Unary +, -, ++, --, sizeof
, &, *, 2) Binary +, -, *, /, %
- 3) Ternary ?:

Based on the operation on operands.

- 1) Arithmetic Operators
+, -, *, /, %
- 2) Increment and Decrement Operators
++, --
- 3) Relational
==, !=, >=, >, <=, <
- 4) Logical
&&, ||, !
- 5) Bitwise
&, |, ^, >>, <<
- 6) Address Operator and Dereferencing Operator
&, *
- 7) Assignment =
- 8) Short-hand
+=, -=, *=, /=, %=
- 9) Misc operators
sizeof



Arithmetic Operators:

These operators are used to perform arithmetic/mathematical operation and give us results accordingly.

`+` (Plus) – Give sum as a result. Example: $1+2=3$

`-` (Minus) – Give the difference as a result. Example: $-1-2=-3$

`*` (Asterisk) – Used for multiplication and given the product as a result. Example: $2*2=4$

`/` (Forward slash) – Used for division and give quotient as a result. Example: $10/2 = 5$

`%` (modulus) – Gives the remainder as a result. Example: $10 \% 11 = 10; 17 \% 3 = 2$

Note: `+` and `-` can also be used as unary operators –in such case, the operator is prefixed – appears to the left of the operand.

If the operands are int for `/`, the result is int quotient obtained truncation of the result. The operator

`%` is defined on integral types only.

If the operands are mixed types(int and double) in operations `+ - * /`, the result is of type double.

There is no exponentiation operator in C.

Coding Example_1: Arithmetic operators in an expression

```
#include <stdio.h>

int main()
{
    printf("%d%d%d%d",2+3,4-2,4*2,5/2); // 5,2,8,2
    printf("%d %f %f %f\n", 25 / 4, 25.0 / 4.0, 25.0 / 4, 25 / 4.0);
    // 6 6.25 6.25 6.25
    // mixed mode
    // 25.0 / 4 => 25.0 / (double)4.0

    return 0;
}
```

Increment and Decrement Operators

There are two types of **Increment (++) and Decrement (--) Operators.**

1. Pre -increment and Pre-decrement operators: First increment or decrement the value and then use it in the expression.
2. Post -increment and post-decrement operators: First use the value in the expression and then increment or decrement.

When `++` and `--` are used in stand-alone expression, pre and post increment and decrement doesn't make difference.

Coding Example_1: Demonstration of pre and post increment/decrement in stand-alone expressions

```
#include<stdio.h>

int main()
{
    int i=5;           int j=5;
    print("Beginning i value is %d\n",i); //5
    print("Beginning j value is %d\n",j);
    ++i;// i++;
    printf("Later i value is %d\n",i);
    j--;// --j;
    printf("Later j value is %d\n",j);
    return 0;
}
```

Coding Example_2: Code which demonstrates the usage of ++ and -- in an expression.

```
#include<stdio.h>

int main()
{
    int a=34; int b;

    printf("Beginning a value is %d\n",a);// 34

    b=a++; // b is 34 and a is incremented to 35

    //b=++a;// First increment and then use the value in the expression

    // b is 35 and a is 35

    printf("b is %d and a is %d\n",b,a);

    // same works for pre and post decrement operators

    return 0;
}
```

Relational Operators

Used to **check the relationship between the two operands**. If the relation is true, it returns 1. If the relation is false, it returns the value 0. Relational operators are used in **decision-making and loops constructs**. Programming language like C which doesn't support Boolean data type return result as 1 or 0.

Greater than (>) – Returns true (1) when the left operand value is greater than the right operand value. Example: 15 > 13 is evaluated to 1.

Less than (<): Returns true(1) when the left operand value is less than the right operand value else false(0). Example: 15 < 13 is evaluated to 0.

Greater than or equal to (>=): Return true(1) when the left operand value is greater than or equal to the right operand value. Example: 15 >= 13 is evaluated to 1.

Less than or equal to (<=): Return true(1) when the left operand value is less than or equal to the right operand. Example: 15 <= 13 is evaluated to 0.

Equal to (==): Returns true when left operand value is equal to right operand value.

Example: 5 == 5 is evaluated to 1.

Not Equal to (!=): Return true when left operand value is not equal to right operand.

Example: $2 \neq 3$ is evaluated to 0

The result of relational operation is 1 or 0. 1 stands for true and 0 for false. The relational operators should not be cascaded – not logically supported in C.

Coding Example_3: Demo of cascading of operators in C

```
#include<stdio.h>
int main()
{
    int a=23;      int b=45;
    //printf("%d",5==5 && 5==5); //True results in 1
    printf("%d", 5 == 5 == 5); // results in 0
    return 0;
}
```

Bit-wise Operators

Used for performing bitwise operations on bit patterns or binary numerals that involve the manipulation of individual bits. Bitwise operators always evaluate both operands. Bitwise operators work on bits and perform bit-by-bit operations.

& (AND): Example: $a \& b$

|(OR): Example: $a | b$

^(Exclusive OR (XOR)): Example: $a ^ b$

~(One's Complement (NOT)): Example: $\sim a$

>> (Shift Right): Example: $a >> 1$

<< (Shift Left): Example: $a << 1$

Few inputs about carrying out arithmetic operation using bit wise operators

* multiply variable n by 2

$n << 1$

* check whether n is odd

n & 1

* swap two variables a and b

a = a ^ b

b = a ^ b

a = a ^ b

* check whether ith bit in variable n is 1

n & (1 << i) : 0 implies not set and non 0 implies set

* set the ith bit in n

n = n | 1 << i

* clear the ith bit in n

n = n & ~(1 << i)

Coding Example_4: // Demo of bitwise operators

```
int main()
{
    int a=10;
    int b=8;
    printf("%d",a&b);    //8
    printf("%d",a|b);    //10
    printf("%d",a<<2); //40
    printf("%d",a>>2); //2
    printf("%d",~(10)); // -11
    return 0;
}
```

Assignment Operators

Used to assign a new value to a variable. Assignment operators can also be used for logical operations, bitwise logical operations or operations on integral operands and Boolean operands.

= **Simple assignment operator:** Assigns values from right side operands to left side operands. Example: $C = A + B$ will assign the value of $A + B$ to C

+= **Add AND assignment operator:** It adds the right operand to the left operand and assigns the result to the left operand.

Example: $C += A$ is equivalent to $C = C + A$

-= **Subtract AND assignment operator:** It subtracts the right operand from the left operand and assigns the result to the left operand.

Example: $C -= A$ is equivalent to $C = C - A$

*= **Multiply AND assignment operator:** It multiplies the right operand with the left operand and assigns the result to the left operand.

Example: $C *= A$ is equivalent to $C = C * A$

/= **Divide AND assignment operator:** It divides the left operand with the right operand and assigns the result to the left operand.

Example: $C /= A$ is equivalent to $C = C / A$

%=: **Modulus AND assignment operator:** It takes modulus using two operands and assigns the result to the left operand.

Example: $C \%= A$ is equivalent to $C = C \% A$

Ternary/Conditional operator

? : Conditional Operator and it requires three operands. $E1 ? E2 : E3$ where $E1$, $E2$ and $E3$ are expressions. The expression $E1$ is first evaluated. If it is true, then the value of the expression is $E2$ else it is $E3$.

Coding Example_5: Demo of ?:

```
int main()
{
    int a= 10; int b = 20;
    printf("%d\n", (a>b)?a:b);           //20
    (a>b)? printf("%d",a):printf("%d",b); //20
    printf("%d", (a>b)?a:b);           //20
    return 0;
}
```

Logical Operators

Logical Operators are operators that determine the **relation between 2 or more operands and return a specific output as a result of that relation.**

NOT (!) – Used to negate a Logical statement.

AND (&&) and **OR (||)** – Used to combine simple relational statements into more complex Expressions. In ‘C’, 0 is false and any non-zero value is true. C follows **short circuit evaluation**. Evaluation takes place left to right and evaluation stops as soon as the truth or falsehood of the expression is determined.

Note: Some of the operators support Sequence Point Operation.

Sequence Point Operation

How do we write safe and proper code in ‘C’ if operators have side effects and the variable values are not defined in such cases? The language ‘C’ also **specifies points in the code beyond which all these effects will definitely be complete. These are called sequence points.**

It defines **any point in the execution of a program which guarantees or ensures that all the side effects the previous evaluation of the program’s code are done or successfully performed.** It ensures that **none of the alterations or side effects of the subsequent evaluations is yet to be performed at all.** The term “**side effects**” is nothing but the **changes done by any sort of function or expression where the state of something gets changed.**

There are some of the basic sequence points available in C Language.

- **Logical AND - &&**
- **Logical OR - | |**
- **Conditional Operator - ?:**
- **Comma Operator - ,**
- **Expression of while loop**
-

```
// int a = 10;
// to support short ckt evaluation, && becomes a sequence point
// expression before &&, || will be completely evaluated.
//a++ == 10 && a == 11 // ok; will be true 1
```

a + a++ is undefined as the value of left a is not defined.

a++ == 10 && a == 11 will be true.

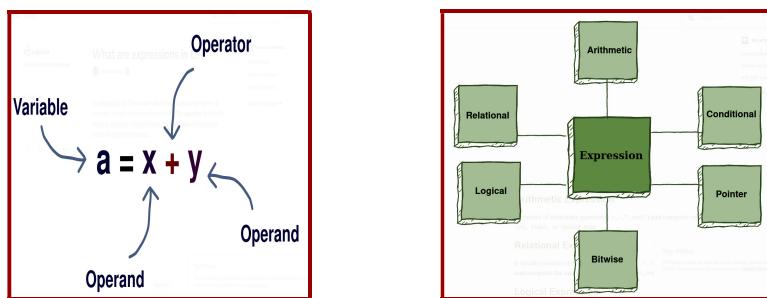
The value a++ is 10 and a becomes 11 before the evaluation moves across the sequence point &&.

Coding Example_6:

```
#include<stdio.h>
int main()
{
    int a = 11;
    printf("%d",a++==12 || a==12); // 1
    printf("\n%d",a); // 12
    return 0;
}
```

Expression in C

An expression in C is a **combination of operands and operators**. Expressions compute a single value stored in a variable. The **operator denotes the action or operation** to be performed. The **operands are the items to which we apply the operation**.



An expression can be defined depending on **the position and number of its operator and operands**.

Infix Expression: Operator is used **between** the operands

$a = x + y$

Postfix Expression: Operator is used **after** the operands

$xy+$

Prefix Expression: Operator is used **before** the operands

$+xy$

Unary Expression: **One operator and one operand**

$++x$

Binary Expression: **One operator and two operands**

$x+y$

Arithmetic Expression: It consists of arithmetic operators (+ , - , * , and /) and computes values of int, float, or double type.

Relational Expression: It consists of comparison operators (> , < , \geq , \leq , == , and !=) and computes the answer in true (1) or false (0)

Logical Expression: It consists of logical operators (&& , || , and !) and combines relational expressions to compute answers in true (1) or false (0)

Conditional Expression: It consists of conditional statements that return true(1) if the condition is met and else false(0)

Pointer Expression: It consists of an ampersand (&) operator and returns address values.

Bitwise Expression: It consists of bitwise operators (>> , << , ~ , & , | , and ^) and performs operations at the bit level.

Evaluation of an expression involves two parts:

1. Evaluation of operands:

This is fetching the operands from the memory of the computer to the registers of the CPU. All the evaluations happen within the CPU. This order is not defined. The idea is to allow the compiler to optimize fetching of the operands. The compiler may generate code such a way that if an operand is used more than once, it may get fetched only once.

2. Evaluation of operators:

This follows the rules of precedence and if more than one operator has the same level of precedence, follows the rules of association.

Table of operator Precedence and Associativity is shown below and Link for reference:
<http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf>

Does the first rule have any effect on our programs? Yes. It affects our programs if the expressions we use are not pure and have side effects.

All assignments have side effects.

Let us consider an example.

a = 10;

b = a * a++;

The value of a++ is the old value of a and therefore 10. What is the value of the leftmost a? Is the value evaluated before incrementing or after incrementing? It depends on the compiler and the compiler options. So value of leftmost a is undefined. So, value of b is undefined.

```
// 10 * 10 => 100  
// 11 * 10 => 110  
//undefined; depends on the order of evaluation of operands  
// bad code
```

C Operator Precedence Table

C operators are listed in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
O [] . .> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of type) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %=&= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Happy Coding using Expressions using Operators!!



**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 string that contains the type specifier(s)

"..." (ellipsis) indicates that the function accepts a variable number of arguments; each argument must be a memory address where the converted result is written to.

On success, the function writes the 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 `commabetweentwo 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);

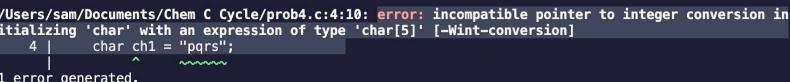
    return 0;
}

// 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.

```



/Users/sam/Documents/Chem C Cycle/prob4.c:4:10: error: incompatible pointer to integer conversion in
initializing 'char' with an expression of type 'char[5]' [-Wint-conversion]
4 | char ch1 = "pqrs";
| ^~~~~~
1 error generated.



```
#include <stdio.h>
int main() {
    char x, y;
    x = getchar();
    y = getchar();
    printf("%c\n%c", x,y);
    return 0;
}
```

34
3
4%

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**. The 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: **signed** and **unsigned**

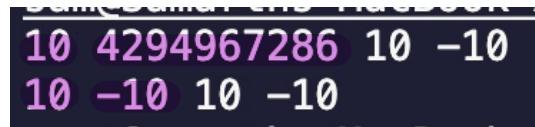
A **signed qualifier** specifies a variable which can hold both positive and negative integers.

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!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

*Lecture #9 & #10
Programming using C Constructs
[Variables, constants, types, typecasting, IO and formatting the data]*

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 Pujari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Programming using C Constructs - [Variables, constants, types, typecasting, IO and formatting the data]

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

Solve the below programs using C.

Link for Solution is here:

https://drive.google.com/file/d/1qqWwFs63ictEpQBveko8Aph31Akw1moh/view?usp=drive_link

Level- 1: Banana

1. An astronaut wants to display the distance to the Moon from Earth in kilometers. Write a C program that prompts the user to enter the distance between the Earth and the Moon, stores it in an appropriate data type, and then displays the entered value with proper formatting.
2. A farmer wants to count the total number of apples in his orchard. Write a C program that prompts the user to enter the number of apple trees and the average number of apples per tree. The program should then calculate and display the total number of apples in the orchard.
3. King Arthur wants to announce the Royal Banquet, but he speaks only one letter at a time. Write a C program where each letter of "Royal Banquet" is printed individually in a separate line.
4. A sorcerer wants to encode a magical spell using different number formats. Write a C program that takes a integer as input and displays it in **decimal, hexadecimal, and octal** formats, showcasing the spell's encoded power.
5. A futuristic robot has a unique identification number that is too long. Write a C program that allows the robot to input and display its ID, ensuring it can handle large numbers accurately.
6. An astronaut records the details of planets using numerical data only. Write a C program that stores and displays **diameter (in km), mass (in kg), and number of moons** for multiple planets in a structured tabular format as shown below.

Diameter (km)	Mass (kg)	Moons
12742	5.97e+24	1
6779	6.42e+23	2
139820	1.90e+27	79
116460	5.68e+26	83
12104	4.87e+24	0

7. Write a C program that takes a character input from the user and prints its ASCII value.
8. Write a C program that takes a negative floating-point number and typecasts it into an integer. Print both floating pointer and integer value in a separate line.
9. Given an integer N and a position K, set the K-th bit of integer N to 1. Display N and the new number for which Kth bit is set.
10. Check whether the integer entered by the user is Even or Odd using Bitwise operators and Ternary Operators.

Level- 2: Orange

11. **Captain Zog** is preparing for a space mission to Mars! As part of his mission briefing, he needs to **input and record** the following crucial details:

1. **Spaceship Speed (in km/s)** – The speed at which the spaceship is traveling.
2. **Distance to Mars (in million km)** – The estimated distance to Mars from the current location.
3. **Mission Code (a unique integer identifier)** – A secret mission identification number.

Write a **C program** that prompts Captain Zog to enter these values, stores them in appropriate variables, and then neatly displays the recorded information in a formatted manner as shown below.

```

Enter spaceship speed (km/s): 25.7
Enter distance to Mars (million km): 225.3
Enter mission code: 45789

-----
🔗 Mission Briefing: Captain Zog's Journey
-----
Spaceship Speed      : 25.70 km/s
Distance to Mars    : 225.30 million km
Mission Code         : 45789
-----
```

12. Write a program using C to swap two integer constants only using the XOR operator.
13. Given an integer N and a bit position K, toggle the K-th bit and print the new number.
14. Two brave knights—**Sir Andor** and **Sir Bixor**—are preparing for battle. Before they begin, the kingdom's wizard has a magical rule:
"Two knights can only duel if they have opposite energies!"

Each knight carries a unique **energy level**, represented as an **integer**. The wizard uses a special **bitwise spell** to determine if their energies are **oppositely charged** (one positive, one negative). Write a **C program** that takes two integer energy levels as input and determines **if they have opposite signs using bitwise operations**.

Expected IOs.

Input 1:

Enter energy level of Sir Andor: 10

Enter energy level of Sir Bixor: -25

Output 1:

The knights have opposite energies! Let the duel begin!

Input 2:

Enter energy level of Sir Andor: 15

Enter energy level of Sir Bixor: 8

Output 2:

The knights have similar energies. No duel today!

15. In the **kingdom of Numland**, the king has declared a **fixed tax rate** for all merchants. The tax rate **never changes** and is set by law.

Every merchant must:

1. Enter their total earnings (in gold coins).
2. Pay taxes based on the fixed tax rate.
3. Get their net income after tax deduction.

Write a **C program** that takes **merchant earnings** as input and calculates and displays **tax amount** and **net earnings** with proper formatting.

Level- 2: Jackfruit

16. Captain Pointers, a legendary pirate, has discovered an ancient map leading to a **single treasure** buried deep in memory. However, he must use **only one pointer** to find the treasure, retrieve its value, and print its location. Write a **C program** to do the following.

- Declares a **single treasure variable** (gold coins).
- Uses **only one pointer** to access its value and memory address.
- Displays the treasure amount and its memory location.

17: Given a number, check if the number is a Power of two or not using bitwise operator in C.

18. The **wizard** wants to **fuse** two knights' energies into a single **powerful spell**. The fusion follows these rules:

1. **If both knights have even energy levels**, the final power is their **bitwise OR** (`|`).
2. **If both knights have odd energy levels**, the final power is their **bitwise AND** (`&`).
3. **If one is even and the other is odd**, the final power is their **bitwise XOR** (`^`).
4. **If the result is a multiple of 4**, the wizard gets extra power and doubles the value.

Write a **C program** that takes two knights' energy levels as integer inputs and calculates the final fused energy based on these rules.

19. A **brilliant scientist** has built a **time machine** that can send objects into the future or past based on **three factors**:

1. **Object mass (floating-point number)** – needs to be **rounded down** to an integer for calculations.
2. **Time warp factor (integer)** – must be **converted into a float** for precision in calculations.
3. **Final destination year** – calculated using modulus on mass by 26 and encoded into an ASCII character for a **secret passcode** to activate time travel.

Simulate the experiment using typecasting in C.

Expected output:

Enter the object's mass (kg): 78.6

Enter the time warp factor: 125

Rounded mass (int): 78

Warp factor as float: 125.000000

Time travel passcode (char): A

20. In a mystical land, wizards use **runes (characters)** to cast spells. However, before a spell is activated, the **rune must be transformed** using bitwise operations based on the following magical rules:

1. **If the rune is an uppercase letter (A-Z)**, toggle its case using the **XOR (^) operation** with 32 (0x20).
 - o Example: '`A`' ^ 32 → '`a`', '`Z`' ^ 32 → '`z`'
2. **If the rune is a lowercase letter (a-z)**, transform it into the next uppercase letter using **bitwise AND (&)** with ~32 (0xDF).
 - o Example: '`b`' & ~32 → '`B`', '`y`' & ~32 → '`Y`'
3. **If the rune is a digit (0-9)**, shift its ASCII value forward by **one position** using **bitwise OR (|)** with 1.
 - o Example: '`4`' | 1 → '`5`', '`8`' | 1 → '`9`'
4. **If the rune is any other character**, leave it unchanged.

Write a **C program** that takes a **single character** as input and applies the **bitwise transformations**.

Master Programming with Variables, Pointers, Constants, Typecasting, IO and Neatly Formatted Data!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #11 & #13
Control Structures - Looping and Selection statements***

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: Control Structures – Looping and Selection Statements

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

A program is nothing but the execution of sequence of one or more instructions. It is most of the time desirable to alter the sequence of the statements in the program depending upon certain circumstances or as per the logic. Involves decision making to see whether a particular condition has been met or otherwise and direct the computer to execute certain statements accordingly. Based on logic, it becomes necessary to **alter the flow of a program, test the logical conditions and control the flow of execution** as per the selection of these conditions that can be placed in the program using decision-making statements.

Control Structures

C programming language supports a few looping and a few selection structures. It may also support some unconventional control structures.

1) Looping structures:

- while statement
- for statement
- do ... while statement

2) Selection structures/ Decision Making/ Conditional structures:

- Simple if Statement
- if – else Statement
- Nested if-else statement
- else if Ladder
- switch statement

3) Unconditional structures:

- goto statement
- break Statement
- continue Statement
- return Statement

Looping

Let us start the discussion with the while statement. The while statement starts with the **keyword while** and is **followed by an expression in parentheses** and is followed by a statement or a block.

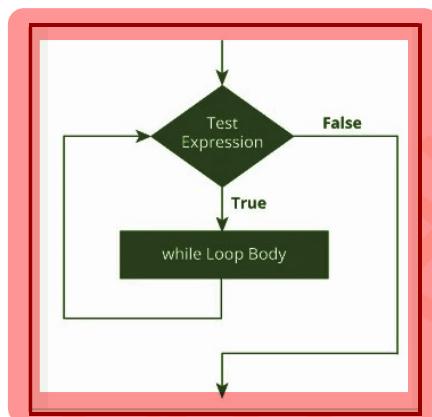
```
while(Expression)
```

```
{
```

```
    Statement(s); // block
```

```
}
```

The expression of the while is evaluated. It could be non-zero or zero. If it is non-zero, the body of the while is executed and again the control comes back to the expression of the while. If the expression of the while is false, the loop is exited.



The **loop evaluates the Test Expression** inside the **parentheses()**. If **Test Expression is TRUE**, **statements** inside the body of while loop are **executed**. For **looping to continue** **Test Expression has to be evaluated again as TRUE**; terminated if the evaluation results in **FALSE**. It is a good programming practice to always use a block even if only one statement is in the body of the loop. **The body of the while loop is executed 0 or more times** as the condition to be checked is at the top.

Coding Example_1: Display all the numbers from n to 0.

Version 1: Results in **infinite** loop as n remains to be 5 → is always true

```
int n = 5;  
while(n)  
    printf("%d ", n);  
    n = n - 1; // control will never execute this statement
```

Version 2: Displays 5 4 3 2 1. But the program layout does not indicate the logic structure of the program. Always indent the code to indicate the logic structure

```
int n = 5; while(n){  
    printf("%d ", n);  
    n = n - 1;}
```

Version 3: Indent the code. Always use the block.

```
int n =  
5;  
while(  
n)  
{  
    printf("%d ", n); // output : 5 4 3 2 1. This is nice.  
    n = n - 1;  
}
```

Version 4: Observe the value of n-- is the old value of n. When n is 1, the value of n becomes 0 but the decrement operator returns 1.

```
int n = 5;  
while(n)  
{  
    printf("%d ", n--); // 5 4 3 2 1  
}
```

Version 5: Pre-decrement operator returns the changed value.

```
int n = 5;  
while(n)  
{  
    printf("%d ", --n); // 4 3 2 1 0  
}
```

Version 6: The expression of while acts like a sequence point.

```
int n = 5;  
while(--n)  
{  
    printf("%d ", n); // 4 3 2 1 0  
}
```

Observe that the old value of n is used to check whether the condition is true or false and by the time, the body of the loop is entered, n would have the decremented value.

Version 7:

```
int n = 5;  
while(--n)  
{  
    printf("%d ", n); // 4 3 2 1  
}
```

Compare the last two versions. In the first case, the loop is executed n times and in the second the loop is executed n –1 times. In the first case, the value of n is -1 and in the second case, it is 0.

Coding Example _2: Find the sum of numbers from 1 to n.

```
int n; // read n from the user  
int i = 1; //  
initialization  
int sum = 0;
```

```

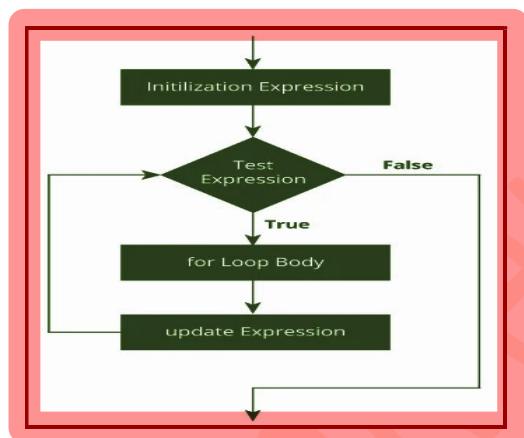
while(i <= n) // condition
{
    sum += i++; //equivalent to sum = sum + i
    // modification
}
    
```

There is clearly some **initialization**, **some processing** and **some modification** at the end of the loop. In such cases, we may want to use the **for statement**.

```

// <for stmt>:: for(e1; e2; e3) <block>|<stmt>
// e1, e2, e3 : expressions

// e1 : initialization
// e2 : condition
// e3 : modification
    
```



For loop in Detail:

e1 is executed once. e2 is checked for condition. If it is non-zero, execute the body of the loop and then execute e3. Again, check for e2 and thus the process continues till e2 results in Zero Value. Once e2 is Zero, come out of the loop and execute the statement outside the body of the loop. **for loop is also called as a pre-tested loop.** The general structure of for loop syntax in C is

```

for (initial value; condition; modification )
{
    statements; // loop body
}
    
```

Note: The semantics of for statement is same as the while statement.

Few questions to Answer:

- Can you have initialization outside the for loop and modification inside the body of the loop?
- Can you skip condition in for loop?
- Can you have only ;; in for loop?

Coding Example_3: What this code does? Why it is wrong?

```
int sum = 0;  
// wrong code  
for(int i = 1; i <= n; sum += i)  
{  
    ++i;  
}
```

Observe closely. You will find that the summation starts from 2 and not from 1. We will also end up adding $n + 1$. Check the way a for statement executes.

```
int sum =  
0;  
for(int i = 1; i <= n; ++i)  
{  
    sum += i;  
}
```

'C' for statement is same as the 'C' while statement –we say they are isomorphic. It is a question of choice and style while writing programs in 'C'.

Coding Example_4: Find what the digital root of a given number

The idea is to add the digits of a given number. If that sum exceeds a single digit, repeat the operation until the sum becomes a single digit. This is useful in parity checking.

```
for(s = 0; n; n /= 10)
```

```
{
```

```
s += n % 10;
}
```

This loop finds the sum of digits of n. The result is in s and n would have become 0. If s exceeds 9, we would like to copy s to n, repeat this operation.

Can we put this code under a `while(s > 9) { <this code> }`?

The answer is a NO as the value of s is not initialized until we enter the inner loop? Shall we initialize to 0? Then the loop itself is not entered. How about making s 10? Looks very unnatural. Can you realize here that the sum becomes available only after doing the summation once? We require in this case, a bottom tested looping structure which executes the code at least once

Here is the solution using do – while.

```
do
{   for(s=0; n; n/2)
{
    s+=n%10
}
n=s;
}while(s>9);
```

do –while loop:

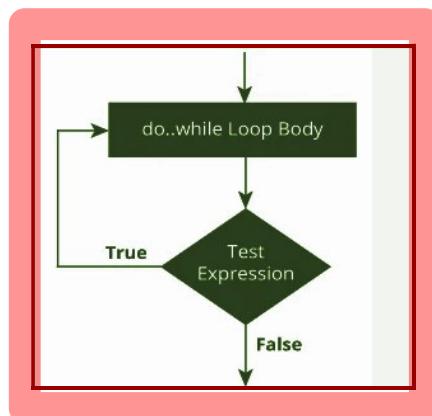
```
#include <stdio.h>

int main() {
    // Loop variable declaration and initialization
    int i = 0;

    // do while loop
    do {
        printf("Geeks\n");
        i++;
    } while (i < 3);

    return 0;
}
```

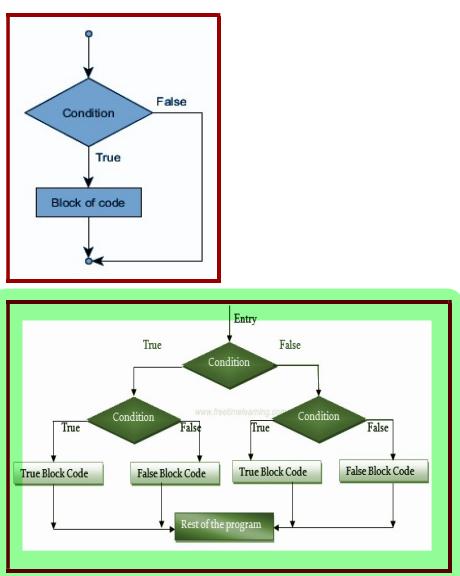
The body of do...while loop is executed at least once. For loop to continue, the test expression is evaluated for TRUE; terminated if the evaluation results in FALSE.



Selection statements

If the expression of if is true we execute the block following if, otherwise execute the else part if any. It is always preferred to use a block even if there is a single statement in the if and else portions.

```
//<if stat> ::= if (<expr>) <stat>|<block>
//<if stat> ::= if (<expr>) <stat>|<block> else <stat>|<block>
```



```
#include <stdio.h>

int main() {
    int n = 10;

    if (n > 5) {
        printf("%d is greater than 5", n);
    }
    else {
        printf("%d is less than 5", n);
    }

    return 0;
}
```

Coding Example_5: Classify triangles given the 3 sides as equilateral, isosceles or scalene. This is one possible solution.

```
int count = 0;
scanf("%d%d %d", &a, &b, &c);
if(a == b) ++count;
if(b == c)
    +count;if(c
== a)
    +count;
if(count == 0) printf("scalene\n");
if(count == 3)
```

```

printf("equi\n"); if(count
== 1) printf("iso\n");
  
```

Compare every pair of sides and increment the count each time –initialized to 0 on start.

Can the count be 2?

Observe a few points here.

- We are comparing integers (integral values)
- We are comparing a variable(an expression) with constants
- We are comparing for equality (no > or <)
- In all comparisons, the same variable is used.

In such cases, we may use switch statement.

```

switch(count)
{
    case 0: printf("scalene\n"); break;
    case 3: printf("equi\n"); break;
    case 1: printf("iso\n"); break;
}
  
```

```

#include <stdio.h>

int main() {
    // Switch variable
    int var = 1;

    // Switch statement
    switch (var) {
        case 1:
            printf("Case 1 is Matched.");
            break;

        case 2:
            printf("Case 2 is Matched.");
            break;

        case 3:
            printf("Case 3 is Matched.");
            break;

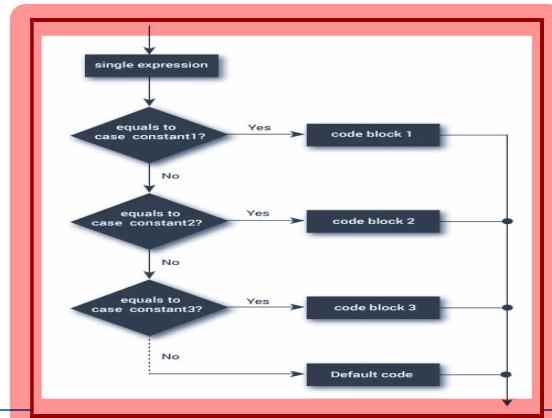
        default:
            printf("Default case is Matched.");
            break;
    }

    return 0;
}
  
```

The value of count is compared with case clauses. This comparison decides the entry into the switch and not exit from there. To avoid the code fall through, we use break statement.

The rule of 'C' is “break if switch”.

If we do not use the break statement, all statements after the matching label are also executed. These switch statements are more efficient compared to normal if statements. But can you include else part of if in switch case? -- Yes using default. But the default clause inside the switch statement is optional.



Coding Example_6: Inclusion of default in switch case to have the else facility

```
printf("enter three sides of a triangle\n");

int a,b,c;

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

int count = 0;

if(a == b) count++;

if(b == c) count++;

if(c == a) count++;

switch(count)

{

    case 1:printf("iso");break;

    case 3:printf("equi");break;

    case 0:printf("scalene");break;

    default:printf("not a valid count");break;

}

printf("switch ended");
```

Few points to think!!

- Is it compulsory to have break for default?
- Can we have default at the beginning of the switch case?

Coding Example_7: The count value cannot be anything other than 0,1 or 2. So change in the default case as below.

```
switch(count)

{
```

```
case 1: printf("iso");break;  
case 3: printf("equi");break;  
default: printf("scalene");break;  
}  
printf("switch ended");
```

Coding Example_7: Find whether the entered character is a vowel or not.

```
#include<stdio.h>  
int main()  
{  
    printf("enter the character\n");  
    char ch;  
    ch = getchar();  
    //scanf("%c",&ch);  
    // solution using if  
    /*if(ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u')  
    {  
        printf("you entered an vowel\n");  
    }  
    else{  
        printf("you did not enter the vowel\n");  
    }*/  
    // Solution using switch case  
    switch(ch)  
    {  
        //case 'a' || 'A': printf("hello 1"); break; // output of expression will be the  
        case label  
        case 'a':  
        case 'A':  
        case 'e':  
        case 'E':
```

```
case 'i':  
case 'o':  
case 'u': printf("you entered an vowel\n");break;  
default: printf("you did not enter the vowel\n");  
}  
// if(ch == 'a' || 'e'|| ) // Logical error  
return 0;  
}
```

Nested control structures

We may have loops and selections nested.

Coding Example_8: Generate all Armstrong numbers between 0 and 999.

The number obtained by raising each of the digits to the power of its length and adding together the sum of the terms obtained. If this sum is same as the given number, the number is same to be Armstrong.

Example: The three digit Armstrong number is 153. The sum of cubes of digits is same as the number. The program shows how not to write programs. We have a loop on hundredth digit, a loop on tenth digit and a loop on unit digit. We cube the digits, find the sum, form the number using these digits and compare the sum with the number formed.

```
#include<stdio.h>  
  
int main()  
{      int h,t,u,hc,tc,uc;  
      int number;  
      for(h = 0; h<10;h++)  
      {  
          for(t = 0; t< 10;t++)  
          {  
              for(u = 0;u< 10;u++)  
              {  
                  hc = h*h*h;
```

```
tc = t*t*t;
uc = u*u*u;
int sum = hc+tc+uc;
number = h*100 + t*10 + u*1;
if(number == sum)
    printf("%d\n",number);

}
}

return 0;
}
```

The above code is generating only few Armstrong numbers. Not printing all. Change the implementation to print all Armstrong numbers between 0 and 999. Think about adding ifs to check and count the number the digits

Also, think how many multiplications do we do for cubing? Should we repeatedly evaluate cube of t in the inner loop when it is not changing. Rearrange the code. Put the statements in the right blocks. You will be able to reduce the number of multiplications for cubing to approximately 1/3rd of what it is now.

Happy Coding using Control Structures



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

Lecture #14
Problem Solving using Control Structures
[Looping statements]

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

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

Unit #: 1**Unit Name: Problem Solving Fundamentals****Topic: Problem Solving using Control Structures - *[Looping Statements]*****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**

Solve the below programs using C.

Link for Solution: https://drive.google.com/file/d/1uyQJ7H3hl_sRFkw2bVLQ-bDeZemzeAoj/view?usp=drive_link

Level- 1: Banana

1. Implement a program that generates and prints the first n numbers in the Fibonacci sequence, where n is provided by the user. The number n is the positive integer entered by the user. If user enters invalid input, display appropriate message.
2. Develop a program that prints all prime numbers up to a specified limit which is a non-negative integer. Display appropriate message if negative number is entered by the user.
3. Imagine you are drawing a block on a digital canvas. Write a program that prints a solid rectangle of stars based on user-specified dimensions.



```
*****  
*****  
*****  
*****
```

4. Design a C program that asks the user for the number of rows and then print a pyramid where each row contains increasing numbers of stars. For example, if the user inputs 5, your program should output the following pyramid:



```
*  
* *  
* * *  
* * * *  
* * * * *
```

5. Create a program that calculates the value of a number raised to a given exponent using a loop. Both inputs from the user. Handle edge cases.
6. **Harshad Number Checker:** A Harshad number is divisible by the sum of its digits. Write a program that checks if a given number meets this criterion for a positive integer entered by the user.
7. **Alternating Sum Series:** Write a program that calculates the sum of the series $1 - 2 + 3 - 4 + \dots$ up to n terms. n is an integer entered by the user. Handle the negative value of n.
8. **Digit Occurrence Counter:** Create a program that counts and displays how many times a specific digit (0-9) appears in a given number entered by the user.

Level- 2: Orange

9. Write a program that calculates the **multiplicative persistence of a number** entered by the user as an integer—the number of steps required to reduce it to a single digit by multiplying its digits.
10. Develop a program that continuously reads numbers from the user, computes their average, and stops when a sentinel value (like 0) is entered and then display the average.
11. Develop a program that converts a binary number (entered as an integer) into its decimal equivalent.
12. Visualize a pyramid that starts wide at the top and narrows down as it descends. Write a program that prints an inverted pyramid of stars for an integer n, that is entered by the user.
 $n = 5$



13. Harmonic Series Sum:

Build a program to compute the harmonic sum $1+1/2+1/3+\dots+1/n$ for a given n, that is entered by the user.

Level- 3: Jackfruit

14. Create a beautiful diamond shape using stars for an integer value n entered by the user.
Sample Output for n = 9 is as below:

```
*  
***  
*****  
*****  
*****  
*****  
*****  
***  
*
```

Sample output for n = any even number is as below:

Invalid input. Please enter an odd positive integer greater than zero.

15. Maximum and Minimum Digits Finder:

Build an application which finds and prints the largest and smallest digit present in the number entered by the user.

16. Kaprekar's Routine Challenge:

Develop a C program that implements Kaprekar's routine for 4-digit numbers. The program should prompt the user to enter a 4-digit number that contains at least two different digits. If the input is invalid, display an appropriate message.

Repeatedly perform the following steps until the number becomes 6174:

1. Rearrange the digits in descending order to form the largest possible number.
2. Rearrange the digits in ascending order to form the smallest possible number (note: leading zeros are allowed).
3. Subtract the smaller number from the larger number.
4. Display the current iteration and the result of the subtraction.

Enjoy crafting code with loops!!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

Lecture #12
Problem Solving using Control Structures
[Selection statements]

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

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

Unit #: 1

Unit Name: Problem Solving Fundamentals

Topic: Problem Solving using Control Structures - *[Selection Statements]*

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

Solve the below programs using C.

Link for Solution: https://drive.google.com/file/d/1CSOBOIB2xvFzjGZ1hW_fk1d8CU-wFs8S/view?usp=drive_link

Level- 1: Banana

1. Write a program that takes the current temperature as input and displays a message such as "It's hot," "It's warm," "It's cool," or "It's cold" based on temperature ranges. Define the temperature range of your choice.
2. Write a program that asks the user for the current hour (in 24-hour format) and then displays an appropriate greeting, such as "Good morning," "Good afternoon," "Good evening," or "Good night."
3. Divisibility Checker : Build and run an application that accepts an integer and checks whether it is divisible by 2, by 3, or by both. Display a corresponding appropriate message based on the condition met.
4. Write a program that reads a vehicle's speed and outputs one of three messages: "Within limit" for speeds at or below 60 km/h, "Slightly over limit" for speeds between 61 and 80 km/h, and "Over speed limit" for speeds above 80 km/h.
5. Age Group Classifier : Develop an application that reads a person's age as a positive whole number and categorizes them into groups such as "Child" (0–12), "Teenager" (13–19), "Adult" (20–64), or "Senior" (65 and above).
6. Grade determinator: Build an application that accepts a student's score as a natural number(from 1 to 100) and assigns a grade. For example, use ranges like 90 and above for an 'A', 80–89 for a 'B', 70–79 for a 'C', 60–69 for a 'D', and below 60 for an 'F'.
7. Develop an application using C to find the eligibility of admission for a professional course based on the following Eligibility criteria:
A. Marks in Maths ≥ 65 and Marks in Physics ≥ 50 and Marks in Chemistry ≥ 55
B. Total in all three subjects ≥ 190 or Total in Maths and Physics ≥ 140
8. Construct a menu-based calculator to perform all arithmetic binary operations on two whole numbers entered by the user.

Level- 2: Orange

9. Imagine that each day of the week unfolds a tiny chapter of an adventure. Write a C program that asks the user to enter a number between 1 and 7 (with each number representing a day from Sunday to Saturday). **Based on the input, display a brief story snippet that represents a part of your adventure.** For example, when the user enters 1, the program might print a short narrative about discovering a mysterious letter on a quiet Sunday morning; for 2, a tale of encountering a secretive stranger on Monday; and so on. If the input is not between 1 and 7, the program should display an error message indicating an invalid day.

10. Create an application that calculates the Body Mass Index (BMI) from user-provided weight in kilograms and height in meters, then categorizes the result as "Underweight," "Normal weight," "Overweight," or "Obese" based on standard BMI ranges mentioned below.
Underweight: $BMI < 18.5$
Normal weight: $18.5 \leq BMI < 25.0$
Overweight: $25.0 \leq BMI < 30.0$
Obese: $BMI \geq 30.0$

$$BMI = \frac{\text{weight (in kilograms)}}{\text{height (in meters)}^2}$$

11. Write a C program that uses selection statements to check if a given year is a leap or not.

12. You are transporting some boxes through a tunnel whose height is only 41 feet. Any box, whose whose any one dimension is above 41 feet, it can't pass through the tunnel. Given the length, width, and height of the box, display the volume of those boxes and check if they pass through the tunnel. Display appropriate message.

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

Input = 1, 2, 40 Output = 80

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

Level- 3: Jackfruit

13. Imagine you find yourself trapped in a mysterious haunted house. Develop a program which prompts the user to **choose different rooms by entering numbers. Determine whether the chosen room hides a ghost, a secret key, or an exit route.** Each decision leads to a different twist in the story.
14. A brave knight embarks on a quest to rescue a princess. Based on obstacles like **encountering a dragon, crossing a river, or navigating through a dark forest,** the knight's fate is decided. Incorporate a switch statement to handle menu-driven challenges (e.g., choosing which path to take or which weapon to use).
15. At a mysterious carnival, the user is invited to play different games, each hiding its own secret. Using a switch statement, allow the user to select a game from a menu. Then, with if-else conditions, determine whether the game results in winning a prize, encountering a challenge, or triggering an unforeseen event that changes the carnival's atmosphere.

Elevate C programming skills by mastering selection statements!!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #15
Single Character Input and Output***

**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: Single Character Input and Output

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

A Character refers to a single input. 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.

Single Character Input-Output Functions:

- 1) **Character oriented Formatted IO Functions** – Described in Standard Input function.
`scanf("%c", &x);
printf("%c", x);`
- 2) **Character oriented Unformatted IO Functions:**
`char c=getchar();
putchar(c);`
- 3) **Character oriented macro implementations:** getc & putc
- 4) **Character oriented Non-Standard functions:** getch , getche & putch

Let us discuss the Character Oriented Unformatted IO functions in detail.

These functions deal with individual characters in C programming. Also, these are part of the stdio.h library and perform direct input and output operations on characters without any formatting.

getchar() : Reads a single character from the standard input (keyboard). Returns the character read as an integer (ASCII value). Returns EOF (-1) if an error occurs or the end of file (EOF) is reached.

Syntax: int `getchar(void);`

Coding Example_1: To read a character

```
#include <stdio.h>  
int main() {  
    char ch;  
    printf("Enter a character : ");  
    ch = getchar();  
    printf("You entered: %c\n", ch);  
    return 0;  
}
```

putchar() : Outputs a single character to the standard output (console). Returns the character printed as an integer (ASCII value). Returns EOF (-1) if an error occurs.

Syntax: int putchar(int ch);

Coding Example_2: To display a character

```
#include <stdio.h>
int main() {
    char ch;
    printf("Enter a character: ");
    ch = getchar();
    putchar(ch);
    return 0;
}
```

Think about this -> Can we use getchar and putchar in a loop and read and display a line?

Coding Example_3: Read and display a line of string from the user.

Version 1:

```
#include <stdio.h>
int main() {
    char ch;
    printf("Enter your name and press enter: ");
    ch = getchar();
    while (ch != '\n')
    {
        putchar(ch);
        ch = getchar();
    }
    return 0;
}
```

Version 2: Initialization and modification has the same statement. Can we combine and make it as a part of condition/ while expression?

```
#include <stdio.h>
int main() {
    char ch;
    printf("Enter your name and press enter: ");
    while ((ch = getchar()) != '\n')
    {   putchar(ch); }
    return 0;
}
```

Character oriented macro implementations:

getc() and putc(): Reads and writes a single character from the specified file or to a specified file. In detail, we will discuss about this Unit-4.

Syntax:

```
int getc(FILE *stream);
int putc(int ch, FILE *stream);
```

Now let us try understanding these functions wrt standard input and output devices. C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Screen

Coding Example_4: Demo of getc and putc

```
#include<stdio.h>
int main ()
{
    char c;
    printf("Enter character: ");
    c = getc(stdin);
    printf("Character entered: ");
    putc(c, stdout);
    return 0;
}
```

Coding Example_5: Demo of getc and putc to read a line from the user and display.

```
#include <stdio.h>
int main ()
{
    char ch;
    while((ch = getc(stdin)) != EOF)
    {
        putc(ch, stdout);
    }
}
```

Key differences between Character Oriented functions

Function	Purpose	Works with Files?	Reads/Writes From
getc()	Reads a character	✓ Yes	A file (FILE *)
putc()	Writes a character	✓ Yes	A file (FILE *)
getchar()	Reads a character	✗ No	Standard input (keyboard)
putchar()	Writes a character	✗ No	Standard output (screen)

Non-Standard functions:

getch(): Reads a single character from the keyboard **without echoing** it on the screen. Returns the character read as an integer. Does **not** wait for **Enter key**; **captures input immediately** and **does not display the character on the screen**. Available in conio.h

Syntax: int getch(void);

Coding Example_6: Demo of getch() function

```
#include <conio.h>

int main() {
    char ch; printf("Press any key: ");

    ch = getch();

    printf("\nYou pressed: %c\n", ch); //Does not wait for Enter; captures input immediately.

    return 0;
}
```

getche(): Reads a single character from the keyboard **and displays it**. Displays the character as it is typed. **Captures only one character** without requiring **Enter**.

Coding Example_7: Demo of getche() function

```
#include <stdio.h>
#include <conio.h>

int main() {
    char ch; printf("Press any key: ");

    ch = getche(); //Does not wait for Enter; captures input immediately.

    printf("\nYou pressed: %c\n", ch);

    return 0;
}
```

Note: **putch()** and **putche()** functions are again non-standard functions that are used to output a character to the console. These functions are specific to certain compilers (such as Turbo C) and are not part of the standard C library and hence not portable across all platforms.

Happy Coding using char IO functions !!



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

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #16
Problem Solving using Control Structures + Single
Character Input and Output***

**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: Problem Solving using Control Structures + Single Character IO**

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

Solve the below programs using Unformatted character oriented IO functions.

Link for Solution is here:

https://drive.google.com/file/d/1cXKomTDKyATzGQKS6j5ngPqgiNToGGCU/view?usp=drive_link

Level- 1: Banana

1. Develop an application to input a character from the user and print the character n times.
2. Implement a C program to read the name of the user and print each character line by line.
3. Bring out an application which displays all Upper case letters and All lower case letters based on the user's choice.
4. Develop an application in C to replace the space with * in a set of words entered by the user in one line.
5. Develop an application to print only the alphabets from the line of input characters entered by the user.

Level- 2: Orange

6. Develop an application to do the following:
 - A) Print the number of characters entered by the user in the input.
 - B) Display the count of alphabets in the user input.
 - C) Display the count of digits in the user input.
- Note: DO NOT use punctuations in any count.**
7. Create a **magic converter** that transforms typed letters into their opposite case and instantly displays them in the new case. This **magic converter works only for alphabets**. Other characters remain the same.
8. Build an application using C to convert the Lowercase letters entered by the user to Uppercase letters. Display the old and the new string in this format: old_char -> new_char. Avoid spaces and tabs in the output.
9. Write a C program that takes a string input from the user and converts it into an alternating uppercase and lowercase format. The transformation should start with an uppercase letter at index 0, followed by a lowercase letter at index 1, and so on. The program should then display the modified string.

10. Implement **Zigzag Case Converter** that reads a character and converts it into a **zigzag case format**:

- The **first letter** should be **uppercase**.
- The **second letter** should be **lowercase**.
- The **third letter** should be **uppercase** again, and so on.
- **Non-alphabetic characters (spaces, numbers, punctuation) should remain unchanged.**

11. Archaeologists have discovered an ancient manuscript where every letter, number, and symbol corresponds to a **unique numeric code**. However, the script is unreadable in its original form. Your challenge is to convert the given text into a sequence of **numeric values**, preserving the true essence of the lost language. Each number must be clearly separated, ensuring that future scholars can reconstruct the ancient text without confusion.

Level- 3: Jackfruit

12. You are a **secret agent** decoding a hidden message left by an unknown sender. However, the message is encoded using a strange cipher: uppercase letters are flipped to lowercase, lowercase letters are flipped to uppercase, digits are shifted forward (9 wraps to 0), and spaces are replaced with underscores. Your mission is to write a program that **deciphers** the message as you type, revealing its true meaning. The transmission continues until new line is encountered, ensuring no secrets remain hidden!

13. You are a **codebreaker** decoding a secret message hidden within ordinary text. Each letter, number, and symbol in the message has a unique **numeric identity(ascii code is unique)**, and your mission is to reveal these hidden codes. For every character in the input, display its corresponding **numerical value**, ensuring that each number is separated by a space for clarity. The message continues to unfold until the transmission ends, revealing the true sequence of hidden numbers behind the text.

14. In a distant future, secret agents communicate using a special code where each character in their message is **shifted by 3 positions forward** in the ASCII table. Your task is to decode their message by converting each character into its **corresponding number** and shifting it **backward by 3** to reveal the original text. The mission continues until the message ends, uncovering the true words hidden within the numbers.

15. Advanced Magic Converter to be developed in C on a line of text to do the following.

- ✓ Converts lowercase to uppercase and uppercase to lowercase
- ✓ Counts the number of uppercase and lowercase letters in the input.
- ✓ Counts the number of non-alphabetic characters (digits, spaces, symbols, etc.) in the input.
- ✓ Displays a final report of the counts with proper formatting.

Explore the power of Character I/O functions and Control Structures!!!
