Session 2

Monday, 7 July 2025 7:47 AM

C Introduction: A Powerful and Enduring Language

C is a general-purpose, procedural computer programming language developed in 1972 by Dennis Ritchie at Bell Labs. It was primarily designed to develop the Unix operating system and has since become one of the most widely used programming languages of all time. Its influence can be seen in many modern languages, including C++, Java, C#, and Objective-C.

Why is C Still So Popular?

Despite its age, C remains incredibly relevant and widely used for several key reasons:

1. **Performance:** C provides low-level memory management and direct hardware access, making it exceptionally fast. This makes it ideal for system programming, embedded systems, and applications where performance is critical.
2. **Portability:** C programs are highly portable. Code written on one system can often be compiled and run on another with minimal modifications, thanks to its standardized nature.
3. **Foundation for Other Languages:** Many modern programming languages are either directly or indirectly derived from C. Understanding C provides a strong foundation for learning these other languages.
4. **System Programming:** C is the language of choice for writing operating systems, compilers, interpreters, and device drivers.
5. **Rich Set of Libraries:** C has a vast collection of standard libraries that provide functions for various tasks, from input/output operations to mathematical calculations.
6. **Direct Memory Access:** Through pointers, C allows direct manipulation of memory, which is powerful for optimizing performance and managing resources efficiently, though it also requires careful handling.

Key Characteristics of C:

- **Procedural Language:** C programs are organized into functions, which are blocks of code that perform specific tasks. The program execution follows a sequence of function calls.
- **Statically Typed:** Variables in C must be declared with a specific data type (e.g., int, float, char) before they can be used. The type checking happens at compile time.
- **Mid-level Language:** C is often referred to as a mid-level language because it combines the features of both high-level languages (human-readable, abstract) and low-level languages (close to hardware).
- **Small Keyword Set:** C has a relatively small set of keywords, making it concise and easier to learn the core syntax.
- **Pointer Support:** Pointers are a fundamental and powerful feature of C, allowing direct memory manipulation. While powerful, they can also be a source of errors if not used carefully.

Common Applications of C:

- **Operating Systems:** Linux, Unix, Windows (parts of it) are written in C.
- **Compilers and Interpreters:** Many compilers for other languages are written in C.
- **Databases:** MySQL, PostgreSQL.
- **Embedded Systems:** Microcontrollers, IoT devices.
- **Gaming:** Game engines and some high-performance game logic.
- **Graphics and Animation:** Used in libraries like OpenGL.
- **Networking:** Network drivers and applications.

Basic Structure of a C Program:

A typical C program has the following basic structure:

C

```
#include <stdio.h> // Header file for standard input/output
int main() {       // Main function: program execution begins here
   printf("Hello, World!\n"); // Prints text to the console
   return 0;       // Indicates successful program execution
}
```

- #include <stdio.h>: This line includes the standard input/output header file, which provides functions like printf for printing to the console and scanf for reading input.
- int main(): This is the main function where the program execution begins. Every C program must have a main function.
- printf("Hello, World!\n");: This line uses the printf function to display the string "Hello, World!" on the console. \n is a newline character.
- return 0;: This statement indicates that the program has executed successfully.

Getting Started with C:

To write and run C programs, you'll need:

1. **A Text Editor:** To write your C code (e.g., VS Code, Sublime Text, Notepad++).
2. **A C Compiler:** To translate your C code into an executable program (e.g., GCC - GNU Compiler Collection, Clang). GCC is widely available for various operating systems.

Next Steps:

To truly learn C, you'll need to delve into:

- **Data Types:** int, float, double, char, void.
- **Variables:** Declaring and initializing variables.
- **Operators:** Arithmetic, relational, logical, bitwise.
- **Control Flow Statements:** if-else, switch, for, while, do-while.
- **Functions:** Defining and calling functions.
- **Arrays:** Storing collections of similar data types.
- **Pointers:** Understanding memory addresses and direct memory manipulation.
- **Strings:** Working with sequences of characters.
- **Structures and Unions:** Creating custom data types.
- **File I/O:** Reading from and writing to files.

In C, an **identifier** is a name given to a programming element, such as a variable, function, array, structure, union, label, or macro. It's how you refer to and differentiate these elements within your code. Think of it like giving a unique name to a box so you can easily find what's inside.

Rules for Naming Identifiers in C:

There are specific rules you must follow when creating identifiers in C:

1. **Characters Allowed:**
   - They can consist of uppercase letters (A-Z).
   - They can consist of lowercase letters (a-z).
   - They can consist of digits (0-9).
   - They can include the underscore character (_).
2. **Starting Character:**
   - An identifier **must** begin with an alphabet (A-Z or a-z) or an underscore (_).
   - It **cannot** start with a digit.
3. **Case-Sensitivity:**
   - C is a **case-sensitive** language. This means myVariable, myvariable, and MyVariable are all considered different identifiers.
4. **Reserved Keywords (Keywords):**
   - You **cannot** use C's reserved keywords (also known as keywords) as identifiers. These keywords have special meanings to the compiler (e.g., int, if, else, while, for, return, void, etc.).
5. **No Special Characters (Except Underscore):**
   - You cannot use any other special characters (like !, @, #, $, %, ^, &, *, (,), -, +, =, {, }, [,], |, \, :, ;, ', ", <, >, /, ?, ., ,) in identifiers.
6. **No Spaces:**
   - Identifiers cannot contain spaces. If you need a name with multiple words, use underscores (e.g., total_sum) or camelCase (e.g., totalSum).
7. **Length (Compiler Dependent):**
   - While there's no strict limit defined by the C standard for the length of an identifier, compilers typically support identifiers up to a certain length (often 31 characters for external identifiers and more for internal ones). It's good practice to keep them reasonably concise but descriptive.

Examples of Valid and Invalid Identifiers:

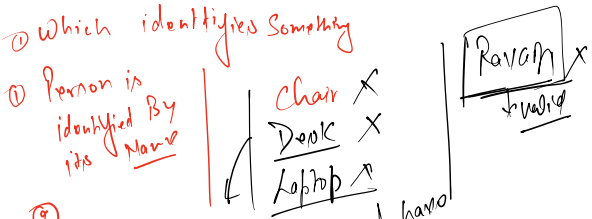
Valid Identifiers	Invalid Identifiers	Reason for Invalidity
myVariable	1variable	Starts with a digit
count	my variable	Contains a space
_temp	int	int is a reserved keyword
calculateSum	total#value	Contains a special character (#)
MAX_SIZE	for	for is a reserved keyword
data_entry_point	first-name	Contains a special character (-)

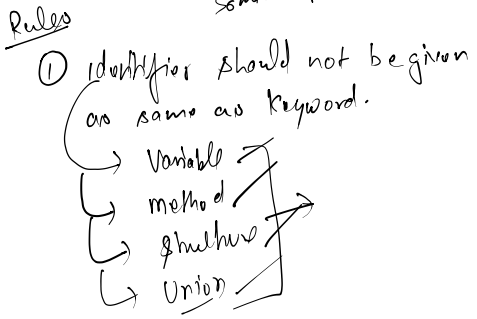
Export to Sheets

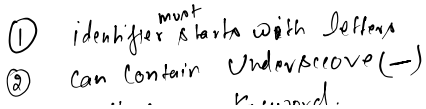
Good Practices for Naming Identifiers:

While the rules define what's *allowed*, good programming practices suggest how to make your identifiers *readable* and *maintainable*:

- **Descriptive Names:** Choose names that clearly indicate the purpose or content of the identifier. For example, studentAge is better than sa.
- **Consistency:** Stick to a consistent naming convention throughout your codebase. Common conventions include:
   - **camelCase:** myVariableName, calculateTotalSum (first letter of first word lowercase, subsequent words start with uppercase). Often used for variables and functions.
   - **PascalCase (or UpperCamelCase):** MyStructName, ClassName (first letter of every word uppercase). Often used for structures, unions, and sometimes function names.
   - **snake_case:** my_variable_name, calculate_total_sum (words separated by underscores). Often







    - **snake_case:** my_variable_name, calculate_total_sum (words separated by underscores). Often used for variables, functions, and macros.
    - **MACRO_CASE (or SCREAMING_SNAKE_CASE):** MAX_VALUE, PI (all uppercase with underscores). Commonly used for symbolic constants defined with #define.
- **Avoid Single-Letter Names (unless context is clear):** While i for a loop counter is common, avoid x or y for more significant variables.
- **Be Mindful of Scope:** In larger programs, consider adding prefixes or suffixes to avoid name clashes (though namespaces in C++ address this more directly).

In C programming, a **data type** is a classification that tells the compiler how the programmer intends to use a variable. Specifically, it determines:

1. **The amount of memory (storage size)** to be allocated for a variable.
2. **The type of values** that can be stored in that memory location (e.g., whole numbers, decimal numbers, characters).
3. **The range of values** that the variable can hold.
4. **The operations** that can be performed on the data stored in the variable.

Think of data types as blueprints for memory. When you declare a variable with a specific data type, you're essentially telling the compiler, "Hey, set aside this much space, and I'm going to put this *kind* of data in it."

C data types can be broadly categorized into:

1. **Basic (or Primitive) Data Types:** These are the fundamental data types built into the C language.
2. **Derived Data Types:** These are constructed from the basic data types.
3. **User-Defined Data Types:** These are custom data types created by the programmer.
4. **Void Type:** A special type indicating the absence of a type.

Let's explore each category:

1. Basic (or Primitive) Data Types

These are the most common and fundamental data types.

a) Integer Types

Used to store whole numbers (integers), both positive and negative, without any decimal points.

- **char**:
    - Used to store single characters (e.g., 'A', 'b', '5', '$').
    - Internally, characters are stored as their ASCII (or Unicode) integer values.
    - **Size:** Typically 1 byte.
    - **Range:**
        - signed char: -128 to 127
        - unsigned char: 0 to 255
    - **Format Specifier (for printf/scanf):** %c
- **int**:
    - Used to store integer values. This is the most commonly used integer type.
    - **Size:** Varies depending on the system/compiler (typically 2 or 4 bytes). On most modern systems, it's 4 bytes.
    - **Range (for 4-byte int):** Approximately -2 billion to +2 billion.
    - **Format Specifier:** %d or %i
- **short (or short int)**:
    - Used for smaller integer values, consuming less memory than int.
    - **Size:** Typically 2 bytes.
    - **Range:** -32,768 to 32,767
    - **Format Specifier:** %hd
- **long (or long int)**:
    - Used for larger integer values than int.
    - **Size:** Typically 4 or 8 bytes (often 8 bytes on 64-bit systems).
    - **Range (for 4-byte long):** Same as int.
    - **Range (for 8-byte long):** Extremely large.
    - **Format Specifier:** %ld
- **long long (or long long int)**:
    - Introduced in C99, used for even larger integer values.
    - **Size:** Typically 8 bytes.
    - **Range:** Very large (up to 9 quintillion approx.).
    - **Format Specifier:** %lld

Type Modifiers for Integer Types:

You can use signed and unsigned modifiers with integer types:

- **signed**: Allows the variable to store both positive and negative values (this is the default for int, short, long, long long).
- **unsigned**: Allows the variable to store only non-negative (zero and positive) values. This effectively doubles the positive range by eliminating the need to store a sign bit.
    - unsigned char (0 to 255)
    - unsigned int (0 to 4,294,967,295 for 4-byte int)
    - unsigned short (0 to 65,535)
    - unsigned long
    - unsigned long long
    - **Format Specifiers:** %u for unsigned int, %hu for unsigned short, %lu for unsigned long, %llu for unsigned long long.

b) Floating-Point Types

Used to store real numbers (numbers with decimal points or fractional parts).

- **float**:
    - Used for single-precision floating-point numbers.
    - **Size:** Typically 4 bytes.
    - **Precision:** Provides about 6-7 decimal digits of precision.
    - **Range:** Approx. 1.2times10−38 to 3.4times1038.
    - **Format Specifier:** %f
- **double**:
    - Used for double-precision floating-point numbers. Provides more precision and a larger range than float.
    - **Size:** Typically 8 bytes.
    - **Precision:** Provides about 15-17 decimal digits of precision.
    - **Range:** Approx. 2.3times10−308 to 1.7times10308.
    - **Format Specifier:** %lf (for scanf) and %f or %lf (for printf)
- **long double**:
    - Used for extended-precision floating-point numbers, offering even more precision than double.
    - **Size:** Varies, typically 10, 12, or 16 bytes.
    - **Format Specifier:** %Lf

c) Boolean Type (_Bool or bool in C99/C11 and later)

- C originally didn't have a built-in boolean type (true/false). Integer values were used, where 0 means false and any non-zero value means true.
- C99 introduced _Bool (and often <stdbool.h> provides a bool macro for convenience, along with true and false macros).
- **Size:** 1 byte.
- **Values:** 0 (false) or 1 (true).
- **Format Specifier:** %d

d) void Type

- The void type signifies the absence of a type or an unknown type. It cannot be used to declare variables directly.
- **Common uses:**
    - **Function return type:** void func_name() indicates that the function does not return any value.
    - **Function arguments:** void func_name(void) indicates that the function takes no arguments.
    - **Generic pointers:** void *ptr; A void pointer can point to data of any type (it's a "generic" pointer). It needs to be cast to a specific data type before dereferencing.

2. Derived Data Types

These are constructed using the basic data types.

- **Arrays:** A collection of elements of the *same* data type, stored in contiguous memory locations.
    - Example: int numbers[10]; (an array of 10 integers)
- **Pointers:** Variables that store memory addresses of other variables. They are crucial for dynamic memory allocation, working with arrays, and efficient function calls.
    - Example: int *ptr; (a pointer to an integer)
- **Functions:** Blocks of code designed to perform a specific task. They take input (arguments) and may produce output (return value).
    - Example: int add(int a, int b) { return a + b; }

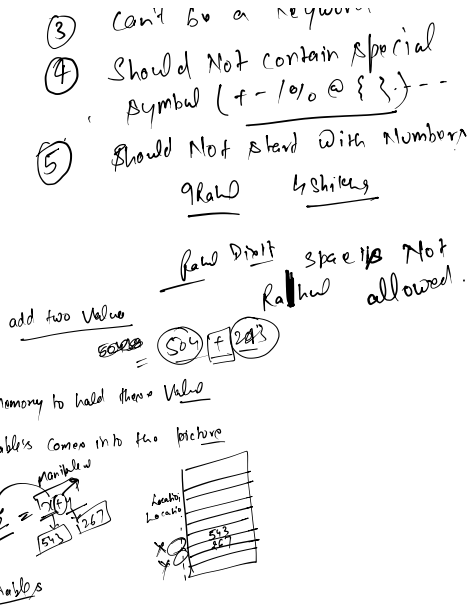
3. User-Defined Data Types

These data types are defined by the programmer to create custom data structures.

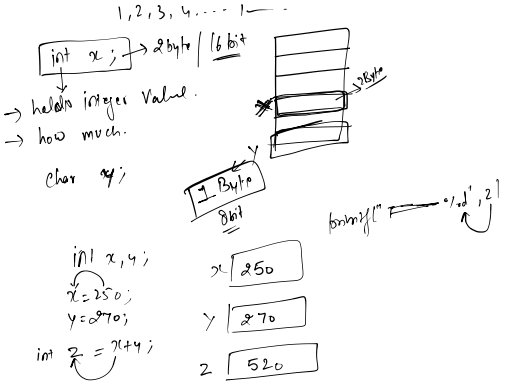
- **struct (Structures):** Allows you to group variables of *different* data types under a single name. Useful for representing complex entities (e.g., a Student structure could have name (char array), roll_no (int), marks (float)).
    - Example:
      C

      ```
      struct Student {
        char name[50];
        int roll_no;
        float marks;
      };
      ```
- **union (Unions):** Similar to structures, but all members share the *same* memory location. Only one member can hold a value at any given time. Useful for memory optimization when you know only one of several possible data types will be needed at a time.
    - Example:
      C

      ```
      union Data {
      ```



Type	Size (bits)	Size (bytes)	Range
char	8	1	-128 to 127
unsigned char	8	1	0 to 255
int	16	2	-2^{15} to $2^{15}-1$
unsigned int	16	2	0 to $2^{16}-1$
short int	8	1	-128 to 127
unsigned short int	8	1	0 to 255
long int	32	4	-2^{31} to $2^{31}-1$
unsigned long int	32	4	0 to $2^{32}-1$
float	32	4	3.4E-38 to 3.4E+38
double	64	8	1.7E-308 to 1.7E+308
long double	80	10	3.4E-4932 to 1.1E+4932



Variable Type	Size	32 bit	Format Specifier	Range
boolean	1		-	True or false
char or signed char	1		%c	-128 to 127
unsigned char	2		%c	0 to 255
short int or signed short int	2		%d	-32,768 to 32,767
unsigned short int	2		%d	0 to 65,535
int or signed int	4		%d, %i	-2,147,483,648 to 2,147,483,647
unsigned int	4		%u, %lu	0 to 4,294,967,295
float	4		%f	3.4E +/-38 (7 digits)
long int or signed long it	4		%l, %ld, %i	-2,147,483,648 to 2,147,483,647
unsigned long int	4		%lu	0 to 4,294,967,295
long long	8		%ld %li	-9,223,372,036,854,775,808 to

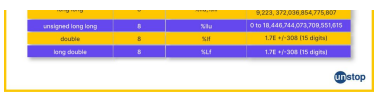
```
union Data {
    int i;
    float f;
    char str[20];
};
```

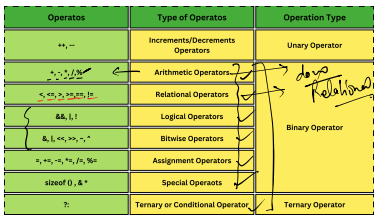
- **enum (Enumerations):** Allows you to define a set of named integer constants. This makes code more readable and maintainable by using meaningful names instead of "magic numbers."
  - Example:
  C

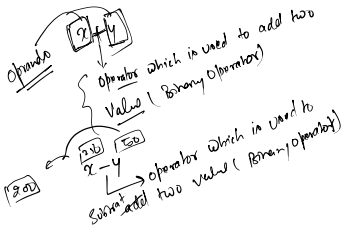
```
enum Weekday {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

- **typedef:** Not a data type itself, but a keyword used to create an **alias** (a new name) for an existing data type. This can improve code readability, especially for complex data types.
  - Example:
  C

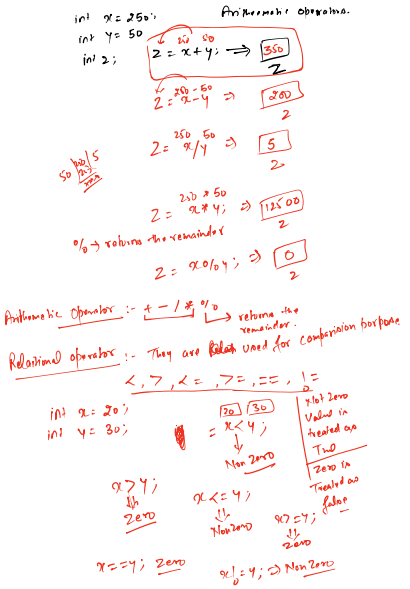
```
typedef long long int LLI; // Now LLI can be used instead of long long int
LLI big_number = 123456789012345LL;
```

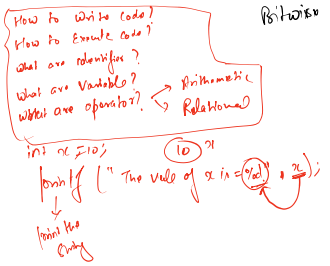






operator are used to perform operation on given operands.





Condition

... to print a Number from 1 to 10?

age1 =
Name = "Ram"

age2 = 26;
Name = "Zeetam"

Q. WAP to print ...

Q. WAP To print the Maximum Number from two?

Q WAP to to Print the Maximum AGE Between two?

Q WAP to Print the Name of the Student whose Age is Maximum?

if / else

unstop

List Of Keywords In C & Their Purpose

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

Datatype are used to declare Variable :-

int/long/short → Store the integer Value

float/double → 42.44 → float x = 44.4;

char → 'c' → Character Value.

Conditional Statement

int x = 27; ✓ x 27
int y = 26; ✓ y 26

{ true } { else }

```
#include <stdio.h>   → Starting Point
void main()
{
  int x = 27;
  int y = 26;
  if (x > y)            27 > 26 → true
  {
    printf("The value of x is greater");
  }
  else {
    printf("The value of y is greater");
  }
```

By the Curly Braces we define the scope of the code

```
int x = 29;     45
int y = 28;     45
if (x > y)  false     27 > 28
{
  printf(" x Value is Greater");
}
printf(" Hello");
```

{ x value is Greater
Hello

Hello Rahul

Literals :- / Constant Values :-

- integer → 1, 2, 3, 4, 5, 6, 7, 999,
- character = 'c', '7', '@', '!', ';'
- floating ⇒ 444, 1.3, 22.333
- String ⇒ "Rahul", "Dixit", "My Name is Rahul";

Q WAP to Print 10 Integer Starting from 1 to 100;

printf("1"); printf("2"); printf("3"); printf("4"); printf("5");

int x = 1;
printf("%d", x); x = x+1;
printf("%d", x); x = x+1;
printf("%d", x); x = x+1;

Loop :- for, while, do{ }while(); — iteration

```
#include<stdio.h>
int main() {
    int x = 1;
    while (x <= 100) {
        printf("%d", x);
        x = x + 1;
    }
```

1 to 100: 1 2 3 ... 100

condition: 2 <= 100 ⇒ true, 3 <= 100 ⇒ true, 101 <= 100 ⇒ false

initialization (int x = 1;)
while (condi) { (x <= 100)
x = x + 1; Updation
}

for (initialization ; condition ; updation)

initialization
do {
Executed
updation
} while (condi);

Entry Controlled

①

```
int x = 10;
while (x < 10)
{
  printf(" The Value of x %d ", x);
  x = x+1
}
```

x 10 < 10 No

No output

for / while are entry controlled loop. The condition check @ Entry level. if the initial condition becomes false then their is No output.

②

```
int x = 10;
do {
  printf(" The Value of x %d", x);
  x = x+1
} while (x < 10);
```

The val of x is 10;

Do while is Exit Controlled loop. If the Initial condi is false then also it executed once.

until condi
while(x) | do{ }

①

```
int x = 9;
while (x < 10)
{
  printf(" The Value of x %d", x);
  x = x+1;
}
```

x = 9
x < 10
The value of x is 9

②

```
int x = 9;
do {
  printf(" The value of x %d", x);
  x = x+1;
} while (x < 10);
```

10 < 10 x

while (condi) { → true
}

do {
} while (cond);
→ false