

# C Programming

Notes/Slides

Vraj Suratwala

B.sc IT -1 (Department of ICT, VNSGU, Surat)

# C Programming Language

fundamental of c

# History

- C was founded and developed by Dennis Ritchie in 1972 at Bell Labs (AT&T Bell Telephone Laboratories) in the United States.
- it uses Traditional Apporoch for programming
- it is a machine independent language.

# Features of C

- lowest level portable language.
- it can also reach very near to CPU instruction.
- used for less latency.

# What is coding ?

- Coding is the way in which /or that provide Communication between human and machine.
- Flow of any programming language.
- human -> code ->Compiler/Interpreter ->binary language -> processing -> Output -> humna

# What is IDE ?

- IDE - refers to Integrated Development Environment which provides combination of tools and setup to write a program or logic or block of code that can create or gives an output to us.
- Compiler : which will compile our code.
- C uses .c extension for programming!

# Preprocessors in C

- The C Preprocessor is a text substitution tool that runs before the actual compilation of code begins.
- It handles things like macros, file inclusion, conditional compilation, and more.

## Main Preprocessor Directives

Directive	Purpose
#define	Defines a macro
#include	Includes header files
#undef	Undefines a macro
#ifdef	If macro is defined
#ifndef	If macro is <b>not</b> defined
#if , #else , #elif , #endif	Conditional compilation
#pragma	Special instructions to compiler

## PRACTICE

# Comments in C

- Comments are the informative statement that never going to be execute.
- just provide the metadata about code .
- increases readability and understanding for other person.
- Two Types of Comments :
- 1. Single Line Comments
- 2. Multi Line Comments

- 1. Single Line Comments.

//This is the Sigle line Comment in C Programming Language.

- 2. Multi Line Comments

```
/*
This
is
MultiLine Comments in C
*/
```

[PRACTICE](#)

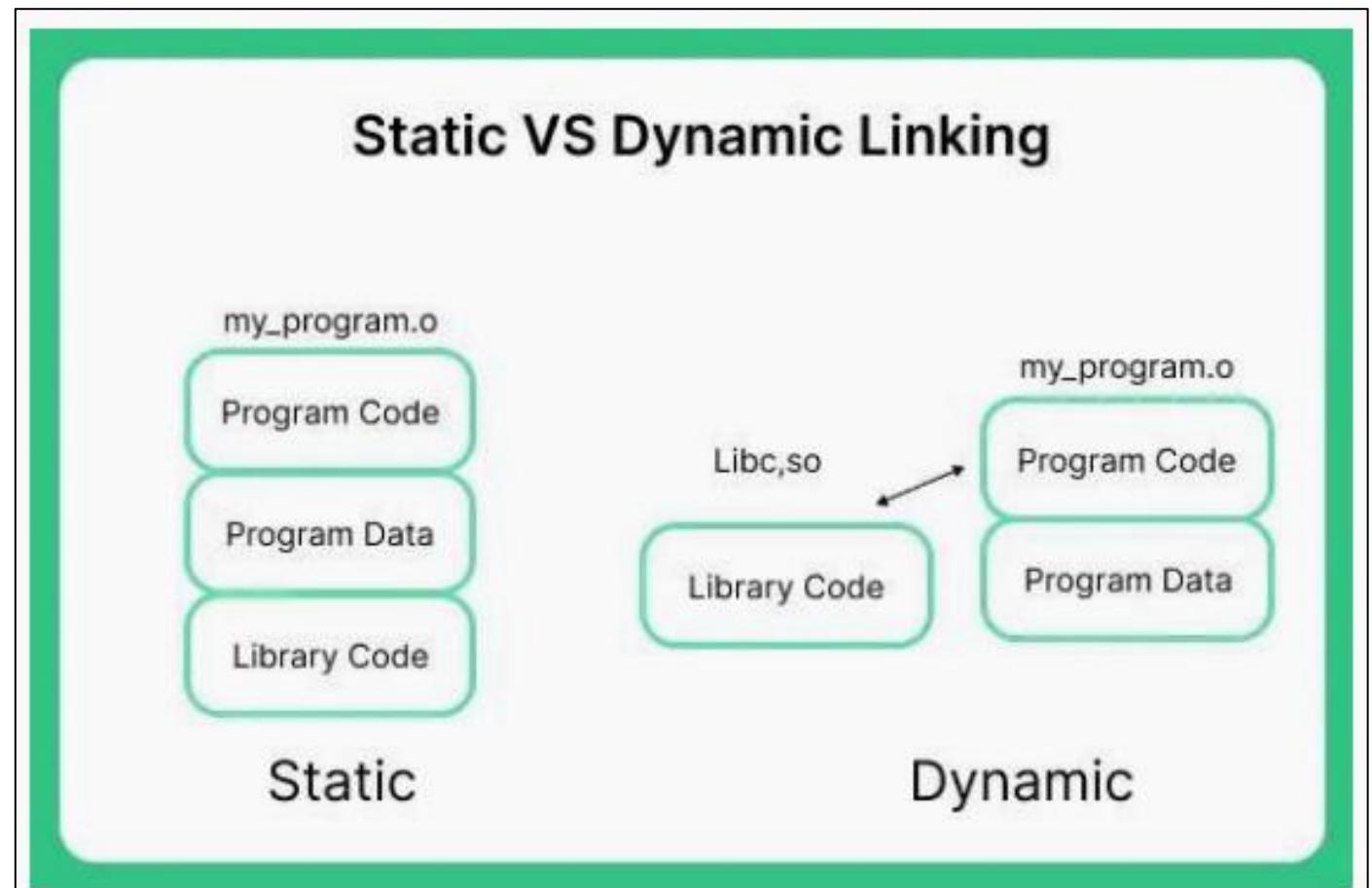
# Internal Process with OS

# Internal Flow of any Program while it is Executing

- Coding -> Compiling -> assembly -> linking -> loading in main memory(RAM)
- .i -> Preprocessing file extension
- .s -> Assembly file extension
- .o -> Machine/Binary file extension

# Linking of any file (OS PART)

- can be in two ways.
- 1. Static
- 2. Dynamic



# Identifiers and Keywords in C Programming

- Identifiers : are the names by which we can identify the variables.
- Ex. int a = 10;
- int this example a is an identifier that identifies the value 10 wrapped with data type integer.
- Keywords : are the reserved words in an any programming language that can not be able to used by programmer because it's already have the functionality or value with it's name.
- let's explore some keywords.



## List of All 32 C Keywords:

Keywords	Keywords	Keywords	Keywords
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

# Rules of Keywords

## Rule No. Rule Description

1. Keywords are reserved — they cannot be used as variable, function, or identifier names.
2. Keywords are case-sensitive — int is a keyword, but Int or INT is not.
3. Keywords must be used in proper syntax — they have predefined meaning and use.
4. Cannot be redefined or re-declared — you cannot create your own variable like float float = 1.0;
5. Must not be used as part of an identifier — avoid names like returnValue that can cause confusion (though it's technically allowed).
6. Each keyword serves a specific purpose — like if for condition, while for loop, return to exit a function, etc.
7. Cannot be overloaded or reinterpreted — unlike in some other languages, C does not allow overriding keyword behavior.
8. Only 32 keywords exist in ANSI C — and this list is fixed; no new keywords can be added in standard C.

# Data Type in C Language

- four types of data types in c.
- 1. void
- 2. Basic
- 3. Derived
- 4. Enumeration

# 1. Void Data Type

- the void keyword represents an empty data type — it means "nothing" or "no value".
- [Practice](#)

## ◆ **Uses of void in C**

Use Case	Meaning
void as return type	Function does <b>not return any value</b>
void as parameter list	Function <b>takes no arguments</b>
void as pointer type	void* is a <b>generic pointer</b> that can point to any data type

# Additional about void

- Important Notes
- You cannot declare a variable of type void:
- **✗** void x; → Invalid
- void\* must be cast before dereferencing.
- Useful in writing generic libraries and function callbacks.

## 2. Basic Data Types

- int - integer value
- char - character values
- float - floating point numbers
- double - long numbers
- PRACTICE

### 3. Derived Data Type

- Array - multiple data of same data type
  - Union - as like structure
  - Pointer - reference of memory location of anther variable
  - Structure - collection of multiple values of multiple data type
- 
- Note : Practice will be given in the separate section for each topic.

## 4. Enumeration

- enum : is a user-defined data type in C used to assign names to integral constants, making code more readable.
- [PRACTICE](#)

# Syntax and Example

Data\_Type Identifier\_Name = Value ;

Example

int ABC = 10;

float BCD = 10.10;

hence variable : name given to particular memory location.

# Operators in C Programming

- Operators able to do operation between two operands.
  - $a+b = c$
- in above example a and b are the operands and ‘+’ is an operator that perform addition operation and in the form result giving the operand c which is addition value of previous both.
- so this is the basic concept that operators are having!
- c is having various kind of operators as well!

# Type of Operators - Practice

- 1. Arithmetic Operators
- 2. Relational Operators
- 3. Logical Operators
- 4. Bitwise Operators
- 5. Assignment Operators
- 6. Miscellaneous Operators

# 1. Arithmetic Operators

Operator	Description	Example
+	Addition	$a + b$
-	Subtraction	$a - b$
*	Multiplication	$a * b$
/	Division	$a / b$
%	Modulus (remainder)	$a \% b$

## 2. Relational Operators

Operator	Description	Example
<code>==</code>	Equal to	<code>a == b</code>
<code>!=</code>	Not equal to	<code>a != b</code>
<code>&gt;</code>	Greater than	<code>a &gt; b</code>
<code>&lt;</code>	Less than	<code>a &lt; b</code>
<code>&gt;=</code>	Greater than or equal to	<code>a &gt;= b</code>
<code>&lt;=</code>	Less than or equal to	<code>a &lt;= b</code>

# 3. Logical Operators

Operator	Description	Example	Copy
<code>&amp;&amp;</code>	Logical AND	<code>a &gt; 0 &amp;&amp; b &lt; 10</code>	
<code>!</code>	Logical NOT	<code>!(a == b)</code>	

# 4. Bitwise Operators

Operator	Description	Example
&	Bitwise AND	a & b
$\vee$	$\vee$	Bitwise OR
$\wedge$	Bitwise XOR	a $\wedge$ b
$\sim$	Bitwise complement	$\sim$ a
$\ll$	Left shift	a $\ll$ 2
$\gg$	Right shift	a $\gg$ 2

# 5. Assignment Operators

Operator	Description	Example
=	Assign	a = b
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b
<<=	Left shift and assign	a <<= 2
>>=	Right shift and assign	a >>= 2
&=	Bitwise AND and assign	a &= b
^=	Bitwise XOR and assign	a ^= b
=`	Bitwise OR and assign	



# 6. Miscellaneous Operators

Operator	Description	Example
<code>sizeof</code>	Size of a data type or variable	<code>sizeof(int)</code>
<code>&amp;</code>	Address of variable	<code>&amp;a</code>
<code>*</code>	Pointer dereferencing	<code>*ptr</code>
<code>? :</code>	Ternary (conditional) operator	<code>(a &gt; b ? a : b)</code>
<code>,</code>	Comma (separates expressions)	<code>a = (1, 2)</code>
<code>-&gt;</code>	Access structure via pointer	<code>ptr-&gt;value</code>
<code>.</code>	Access structure member	<code>obj.member</code>
<code>[]</code>	Array subscript	<code>arr[2]</code>
<code>()</code>	Function call	<code>func()</code>

# Precedence of Operators



## PRECEDENCE & ASSOCIATIVITY

OPERATOR	DESCRIPTION	ASSOCIATIVITY
( ) [ ] . . -> ++ — $\circ$	Parentheses (function call) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	L → R
++ — + — ! ~ ( <i>type</i> ) * & sizeof	Prefix increment/decrement Unary plus/minus Logical NOT/bitwise Negate Cast Dereference Address (of operand) Determine size in bytes on this implementation	R → L
* / %	Multiplication/division/modulus	L → R

# Format Specifiers

- %d - for integer value
- %f - for floating point value
- %o - for octal value
- %Lf - for long double value
- \$lf - for double value
- %l - for long value
- %c - for character value
- %s - for string(character array)
- %% - for percentage sign
- %p/x/X - for hexadecimal value

# escape sequences - in c programming

Escape Sequence	Meaning	Example Output
\n	New line	Moves to a new line
\t	Horizontal tab	Adds a tab space
\\\	Backslash	Prints \
\"	Double quote	Prints "
'	Single quote	Prints '
\r	Carriage return (start of line)	May overwrite the line
\b	Backspace	Deletes one character
\f	Form feed	Advances to new page (rarely used)
\a	Alert (bell sound)	Produces a beep sound

# escape sequences - in c programming

\f	Form feed	Advances to new page (rarely used)
\a	Alert (bell sound)	Produces a beep sound
\v	Vertical tab	Vertical spacing (rare)
\?	Question mark	Prints ? (avoids trigraphs)
\0	Null character (end of string)	Terminates a string

# Beep() function rather than '/a'!!

```
#include<stdio.h>
#include<windows.h>
int main()
{
    Beep(800,800);
    return 0;
}
//TRY THIS FOR FUN!!!!
```

# Memory size of Basic Data Types

## Get the Memory Size

We introduced in the [data types chapter](#) that the memory size of a variable varies depending on the type:

Data Type	Size
int	2 or 4 bytes
float	4 bytes
double	8 bytes
char	1 byte

Question : What the memory size of Boolean ?

Answer : 1 byte - 8 bits - typically!

# Constants in C

- Constant is a variable value that cannot change during the program scope.
  - the value can't be able to alter during runtime.
  - the variable can be declared constant in two ways.
  - 1. using preprocessor - #define
  - 2. using 'const' keyword.
- 
- Syntax & Example : [Practice](#)

# C Programming Language

Conditional Statements

# what are the Conditional Statements ?

- the conditional statements are using to choose / or taking one decision based on particular condition.
- let say for an example , if you are above 18 then you can vote else not!
- so this is the conventional way for taking a decision.

# what are the Conditional Statements ?

- so in c programming there are various way for taking decision using if-else conditional , else if condition or we can also use the jumping statements and looping statements.
- Conditional Statement
  - if statement
  - if-else statement
  - if-else if -else statement
  - shorthand use of if-else laddar
  - nested if-else-else if laddar etc.
  - [PRACTICE](#)
  - [More](#)

# Jumping Statement

- Jumping statements are the statements that are used to jump one line to another line.
- in simple terms going to another part of execution from one part of execution.
- C includes these Jumping Statement.
  - goto statement
  - continue statement
  - break statement
  - [Practice](#)
  - [More](#)

# Looping Statement - Practice

- Loops in Python are used to repeat actions efficiently.
- the types of loop in c.
  - 1. for loop
  - 2. while loop
  - 3. do-while loop

# 1. for Loop

- Syntax :

```
for(initialization,condition,increament/decrement)
{
    //block of code
}
```

- Example :

```
for(int i=0;i<5;i++)
{
    printf("Hello World!");
}
```

## 2. While Loop

- Syntax :

```
while(condition)//condition will be return boolean expression
{
    //block of code
    // incrementation statement
}
```

- Example :

```
while(i<=5)
{
    printf("%d",i);
    i++;
}
```

# 3. Do-while Loop

- Syntax :

```
do{  
    //block of code  
}while(condition);
```

- Example :

```
do{  
    printf("hello");  
    i++;  
}while(i<=5);
```

# Questions - Interview based

- - What are conditional statements in C, and why are they important?
- - Explain the difference between if-else and switch statements in C.
- - What happens if you omit the else block in an if-else statement?
- - Can we nest conditional statements? If yes, provide an example.
- - How does the switch statement work internally in C?
- - What are the advantages of using switch-case over multiple if-else statements?
- - Is switch statement faster than if-else? Explain with an example.
- - What is the role of the break statement inside a switch case?
- [MCQ](#)

# C Programming Language

Arrays

# What is Arrays in C?

- Array is one of the derived data type in the c programming language which can be able to store the multiple elements of one data type.
- Key Points to Remember
- Array size must be a constant or a macro in standard C.
- Arrays are passed to functions by reference (i.e., the base address is passed).
- Out-of-bounds access leads to undefined behavior.

# More About Array

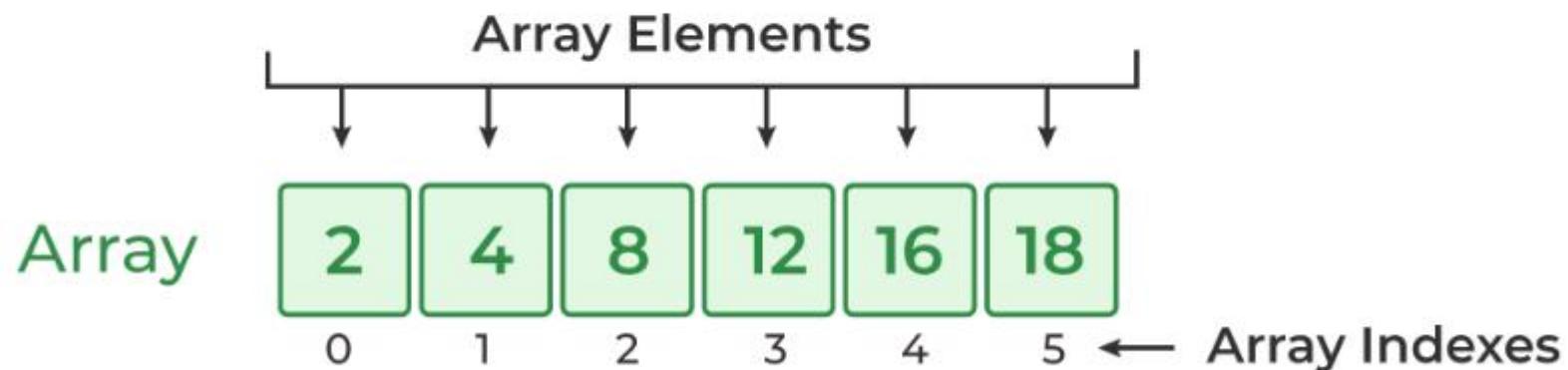
- To create an array, define the data type (like int) and specify the name of the array followed by square brackets [].
- To insert values to it, use a comma-separated list inside curly braces, and make sure all values are of the same data type:
- `int a[5] ={10,20,30,40,50};`

# Types of an Array

- 1. One Dimensional Array (1-D Array)
- 2. Two Dimensional Array (2-D Array)
- 3. Multi Dimensional Array (3-D Array)

# 1-D Array (one -Dimensional Array)

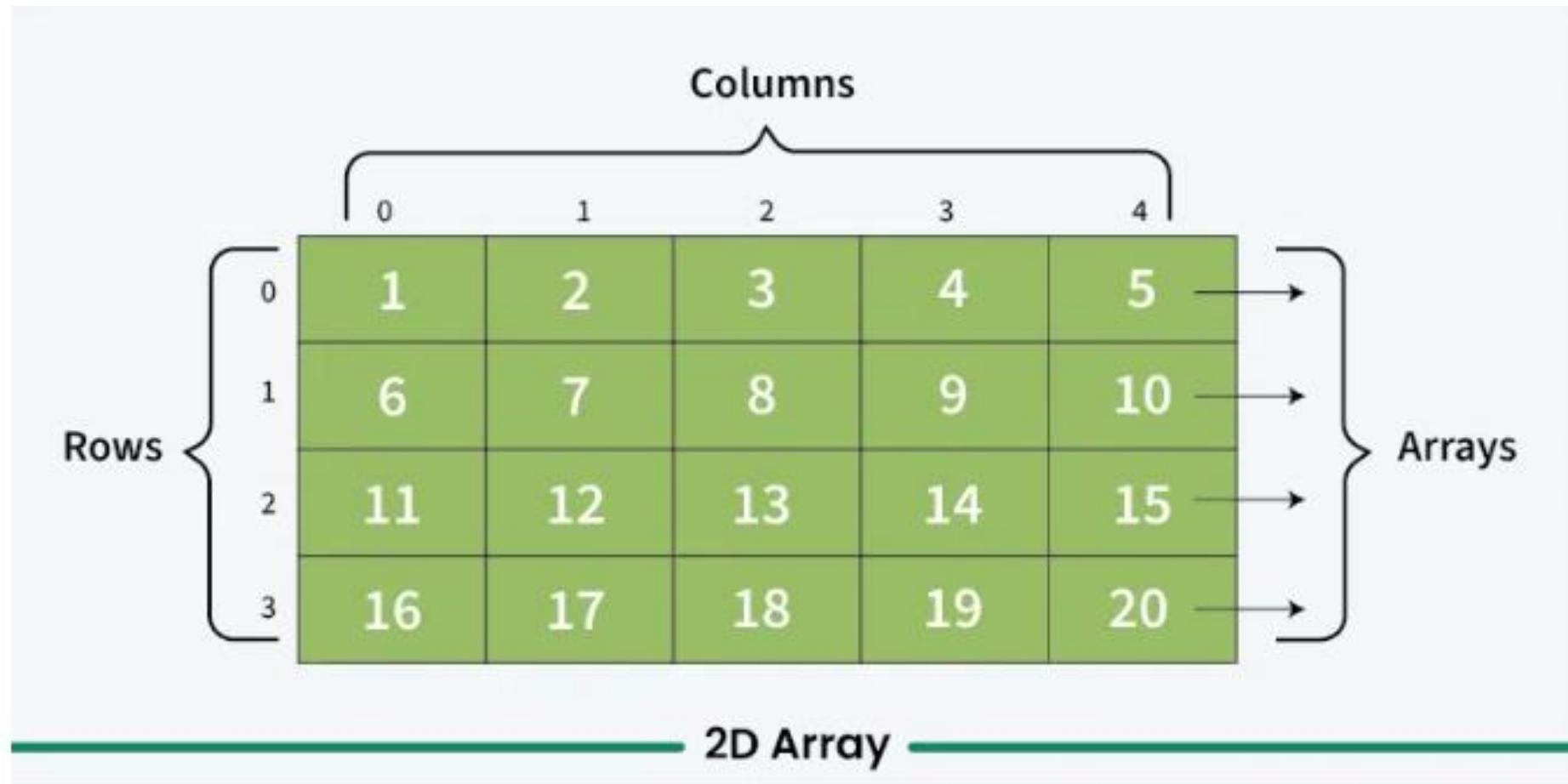
## One Dimensional Array in C



# 1-D Array

- as we seen in above picture the structure of the -D Array now let's move to Syntax as well!
- Syntax : `data_type array_name[size];`
  - `data_type`: The type of elements (e.g., `int`, `float`, `char`).
  - `array_name`: The name of the array.
  - `size`: The number of elements in the array.
- For more Information and Implementation : Practice

# 2-D Array - Concept



# 2-D Array

- A two-dimensional array or 2D array is the simplest form of the multidimensional array.
- We can visualize a two-dimensional array as one-dimensional arrays stacked vertically forming a table with 'm' rows and 'n' columns. In C, arrays are 0-indexed, so the row number ranges from 0 to (m-1) and the column number ranges from 0 to (n-1).
- Syntax :
- type arr\_name[m][n];
- Example : int arr[10][20];

# What is the Logic of Matrix Multiplication ?

- Matrix Multiplication is bit tricky for beginner but it will be easy with understanding of the concept or logic!
- so let's understand!
- link : [Matrix Multiplication Concept](#)
- link : [Code/Logic](#)
- Note :: The Remaining topic like Dynamic Array and all that will be covered in the section of Pointers.

# Competitive Exam Questions or Preparation on Array

- Questions : [Array Questions and MCQ based Exam.](#)

# C Programming Language Strings

# What is String ?

- the character data type is basic need to understand the strings because strings are nothing but it is just an array of characters.
- so in other language we can say that the strings are character array.
- so what is the need of String derived data type ?

# Need of String - Derived Data Type

- the string are needed to represent the group of character at a time or at a place to form a spelling or word!
- so the next Question is how can we define any character array or string?
  - before answring this Question i wanted to tell you that the string always terminalated by NULL '\0' character that represents the ending of any string literal!
  - let's understand the concept!

# Strings in C

## String in C

```
char str[ ] = "Geeks"
```

index → 0 1 2 3 4

str → 

G	e	e	k	s	
---	---	---	---	---	--

Address → 

--	--	--	--	--	--

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is work as an array of characters.

The difference between a character array and a C string is that the string in C is terminated with a unique character '\0'.

# Declaration of Strings

- Declaring a string in C is as simple as declaring a one-dimensional array of character type. Below is the syntax for declaring a string.
- `char string_name[size];`
- In the above syntax `string_name` is any name given to the string variable and `size` is used to define the length of the string, i.e. the number of characters strings will store.
- Like array, we can skip the size in the above statement:
- `char array_name[];`

# Initialization of String

- We can initialize a string either by specifying the list of characters or string literal.
- // Using character list
- char str[] = {'G', 'e', 'e', 'k', 's', '\0'};
- // Using string literal
- char str[] = "Geeks";
- Note: When a Sequence of characters enclosed in the double quotation marks is encountered by the compiler, a null character '\0' is appended at the end of the string by default.

- Accessing
- We can access any character of the string by providing the position of the character, like in array. We pass the index inside square brackets [] with the name of the string.

```
#include <stdio.h>

int main() {

    char str[] = "Geeks";

    // Access first character
    // of string
    printf("%c", str[0]);
    return 0;
}
```

# Update any String

```
#include <stdio.h>

int main() {
    char str[] = "Geeks";

    // Update the first
    // character of string
    str[0] = 'R';
    printf("%c", str[0]);
    return 0;
}
```

# Input from User

```
#include<stdio.h>

int main() {
    char str[5];

    // Read string
    // from the user
    scanf("%s",str);

    // Print the string
    printf("%s",str);
    return 0;
}
```

# Using scanf() with a Scanset

- We can also use scanf() to read strings with spaces by utilizing a scanset.
- A scanset in scanf() allows specifying the characters to include or exclude from the input.

```
#include <stdio.h>

int main() {
    char str[20];

    // Using scanset in scanf
    // to read until newline
    scanf("%[^\\n]s", str);

    // Printing the read string
    printf("%s", str);

    return 0;
}
```

# Using fgets()

- If someone wants to read a complete string, including spaces, they should use the fgets() function.
- Unlike scanf(), fgets() reads the entire line, including spaces, until it encounters a newline.

```
#include <stdio.h>

int main() {
    char str[20];

    // Reading the string
    // (with spaces) using fgets
    fgets(str, 20, stdin);

    // Displaying the string using puts
    printf("%s", str);
    return 0;
}
```

# functions of String in c programming using ‘string.h’

- The C language comes bundled with <string.h> which contains some useful string-handling functions. Some of them are as follows:

Function	Description
<a href="#"><u>strlen()</u></a>	Returns the length of string name.
<a href="#"><u>strcpy()</u></a>	Copies the contents of string s2 to string s1.
<a href="#"><u>strcmp()</u></a>	Compares the first string with the second string. If strings are the same it returns 0.
<a href="#"><u>strcat()</u></a>	Concat s1 string with s2 string and the result is stored in the first string.
<a href="#"><u>strlwr()</u></a>	Converts string to lowercase.
<a href="#"><u>strupr()</u></a>	Converts string to uppercase.
<a href="#"><u>strstr()</u></a>	Find the first occurrence of s2 in s1.

# strcspn() in C

- The C library function strcspn() calculates the length of the number of characters before the 1st occurrence of character present in both the string. Syntax :
- `strcspn(const char *str1, const char *str2)`
- Parameters:
- `str1` : The Target string in which search has to be made.
- `str2` : Argument string containing characters
- to match in target string.
- Return Value:
- This function returns the number of characters before the 1st occurrence
- of character present in both the string.

```
1 // C code to demonstrate the working of
2 // strcspn()
3 #include <stdio.h>
4 #include <string.h>
5
6 int main()
7 {
8
9     int size;
10
11    // initializing strings
12    char str1[] = "geeksforgeeks";
13    char str2[] = "kfc";
14
15    // using strcspn() to calculate initial chars
16    // before 1st matching chars.
17    // returns 3
18    size = strcspn(str1, str2);
19
20    printf("The unmatched characters before first matched character : %d\n", size);
21 }
```

Output:

The unmatched characters before first matched character : 3

# What happens if strcpy() or strcat() is used with NULL?

- If strcpy() or strcat() is used with NULL, it leads to undefined behavior and can cause a segmentation fault (crash).
- Understanding the Issue
- Both strcpy() and strcat() expect valid memory locations to operate on. If you pass NULL, it means the pointer does not point to any allocated memory, and attempting to access or modify it will likely cause a runtime error.

# Example : with issue!

- #include <stdio.h>
- #include <string.h>
- int main() {
- char \*str1 = NULL; // Uninitialized pointer
- char str2[] = "Hello";
- strcpy(str1, str2); // This causes a segmentation fault
- printf("%s\n", str1); // Undefined behavior
- return 0;
- }

# Why does this fail?

- str1 is NULL, meaning it does not point to valid memory.
- strcpy(str1, str2) tries to copy "Hello" into an invalid location, leading to a segmentation fault.

# How to Use These Functions Correctly ?

C

 Copy

```
int main() {
    char *str1 = (char *)malloc(20); // Allocate memory
    char str2[] = "Hello";

    if (str1 == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    strcpy(str1, str2); // Now safe
    printf("%s\n", str1); // Prints "Hello"

    free(str1); // Clean up memory

    return 0;
}
```

# Takeaways

- Never pass NULL to strcpy() or strcat()—always ensure the destination pointer points to a valid memory location.
- Use malloc() or define a static buffer (like char str1[20];) to avoid segmentation faults.
- Want to refer ‘string.h’ ? - [More about it!](#)

# C Programming Langauge functions

# what is function ?

- A function in C is a set of statements that, when called, perform some specific tasks.
- It is the basic building block of a C program that provides modularity and code reusability.
- They are also called subroutines or procedures in other languages.

# Function Definition

- A function definition informs the compiler about the function's name, its return type, and what it does.
- It is compulsory to define a function before it can be called.

# Syntax

- `return_type name () {`
- `// Body of function`
- `};`
- **where,**
  - `return_type`: type of value the function return.
  - `name`: Name of the function
  - `Body of function`: Statements inside curly brackets `{ }` are executed when function call.

# Example

- void hello(){
  - printf("Hello World!");
  - }
- Description : In the above code, hello is the name assigned to the function and its return type is void.

# Calling a function from main() function

- After defining a function, you can use it anywhere in the program by simply calling it with its name followed by parentheses ()�

```
#include <stdio.h>

// Function definition
void hello() {
    printf("GeeksforGeeks");
}

int main() {

    // Function call
    hello();
    return 0;
}
```

# Return type in function

- A function can return a value to its caller as a result. It is called the return value, and the type of this value is called return type of the function.
- The function only returns one value, and the value type is the same as the function's return type.
- The return keyword is used to return some values from the function.

# Example :

```
#include <stdio.h>

int getThree(){
    int n = 3;
    return n;
}

int main() {

    // Print the value that
    // is return by getThree()
    // function
    printf("%d", getThree());
    return 0;
}
```

# Function Parameter

- A function can be provided some values by its caller. These values are called arguments and are provided at the time of function call. In the function definition, we use the placeholder variables inside the parentheses () to receive these values.
- These placeholders are called parameters.
- // Parameters in the function definition
- `return_type func(type1 name1, type2 name2, ...){`
- `// ....`
- `}`
- `name1, name2, ....` are the names given to the parameters of the function. They will be referred with the same name inside the function.

# Example :

```
#include <stdio.h>

// Defining a function that
// print square of given number
void printVal(int num, float real){
    printf("%d %f\n", num, real);
}

int main() {
    int a = 3;

    // call the printVal function and pass
    // desired values
    printVal(a, 1.5);
    return 0;
}
```

# Forward Declaration of a function

- In C, a function must be defined before it is called, or the compiler will generate an error. To prevent this, we can declare the function ahead of the call and definition, providing the compiler with its name, return type, and parameters. This is known as a forward declaration.
- `return_type name();`
- The above statement tells the compiler about a function's name and return type. Mentioning parameters in the declaration is optional but valid.
- `return_type name(type1, type2, ...);`
- This type of function declaration is known as a function prototype or function signature. If it appears before the function call, we can define the function anywhere in the program.
- Note: Function declarations are required, but since the function definition includes the declaration, it's not necessary to declare it separately when the function is defined at the beginning.

# Example

```
#include<stdio.h>
int getnumber(int number); //forward declaration including paramater's way!
int main()
{
    int number = 10;
    printf("\n Function_With_Parameter_and_Return_Type");

    printf("\n NUMBER : %d",getnumber(number));
    return 0;
}

// Definition of that function.
int getnumber(int number)
{
    return number*number;
}
```

# Concept of Local Variable

- Variables declared inside a function are called local variables because they are only accessible within that function. We cannot access them outside the function.

```
#include <stdio.h>

int getThree(){
    int n = 3;
    return n;
}

int getThreeDummy(){

    // Try to access a variable
    // of another function
    return n;
}

int main() {

    printf("%d", getThreeDummy());
    return 0;
}
```

# Types of function in c

- there are two types of function in c.
- 1. Inbuilt function (also known as library function).
- 2. User defined function.(that we seen already!)
- so let's explore bit of library functions.

# Library functions in c

- These functions are part of C libraries, which are pre-defined and perform specific tasks.
- By using them, you can save time and avoid errors with different inputs, as these functions are already tested and optimized for use.

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

int main() {
    int n = 3;
    int m = 6;

    // Using fmax() from math.h to find the maximum
    printf("%.0f", fmax(n, m));

    return 0;
}
```

# Library functions : ‘stdio.h’

- - printf() – Outputs formatted text
- - scanf() – Reads formatted input
- - gets() / fgets() – Reads a string
- - puts() – Outputs a string
- - fopen() / fclose() – Opens and closes files
- - fprintf() / fscanf() – Reads/writes formatted data to a fil

# Library functions : ‘string.h’

- - `strlen()` – Gets string length
- - `strcpy()` / `strncpy()` – Copies a string
- - `strcmp()` / `strncmp()` – Compares strings
- - `strcat()` / `strncat()` – Concatenates strings
- - `strstr()` – Finds a substring

# Library functions : ‘math.h’

- - `sqrt()` – Square root
- - `pow()` – Exponentiation
- - `abs()` – Absolute value
- - `ceil()` / `floor()` – Rounding up/down
- - `sin()` / `cos()` / `tan()` – Trigonometric functions

# Library functions : ‘stdlib.h’

- - malloc() / calloc() / realloc() – Allocates memory dynamically
- - free() – Deallocates memory
- - atoi() / atof() – Converts strings to numbers
- Refers to Memory Management

# Library functions : ‘ctype.h’

- - `isdigit()` – Checks if a character is a digit
- - `isalpha()` – Checks if a character is a letter
- - `toupper()` / `tolower()` – Converts case
- Refers to character classification.

# Library function : ‘time.h’

- - `time()` – Gets the current time
- - `clock()` – Gets processor time
- - `difftime()` – Calculates time difference
- - `strftime()` – Formats time/date

# main() - function in c

- In C programming, there is an entry point where the program starts its execution.
- So main() function is nothing it is an entry point of a program.
- The return value of main function indicates a program successfully or unsuccessfully execute or not.

# Memory Management of C Functions

- When a function is called, memory for its variables and other data is allocated in a separate block in a stack called a stack frame.
- The stack in which it is created is called function call stack. When the function completes its execution, its stack frame is deleted from the stack, freeing up the memory.

# Advanced and Detailed Discussion : function call stack

- What is the Call Stack?
- The call stack is a data structure used by the program during runtime to manage function calls and local variables. It operates in a Last-In-First-Out (LIFO) manner, meaning the last function called is the first one to complete and exit.
- How are Functions Executed?
- A function in C needs memory for the following components:
- Function Arguments: Values passed to the function.
- Local Variables: Variables declared inside the function.
- Compiler also saves the address to return to after the function's execution, referred to as the return address.

## Continue...

- All the data related to a single function call is grouped together and stored in the call stack as a stack frame or activation record.
- The stack pointer (SP) points to the top of the call stack, which is the most recently added stack frame. This allows the program to efficiently track the stack size and manage memory during function calls.
- When a function is called, a new stack frame is pushed onto the call stack. Upon returning from a function, the stack frame is removed (popped), and control goes back to the previous execution point using the return address stored in the popped stack frame.

# Lets' Understand By an Example.

```
#include <stdio.h>

// Definition of function D
void D() {
    float d = 40.5f;
    printf("In function D");
}

// Definition of function C
void C() {
    double c = 30.5;

    printf("In function C\n");
}

// Definition of function B
void B() {
    int b = 20;

    // Calling function C
    C();

    // This will be printed after C() is executed
    printf("In function B\n");
}

// Definition of function A
void A() {
    int a = 10;

    // Calling function B
    B();

    // This will be printed after B() is executed
    printf("In function A\n");
}
```

# Continue...

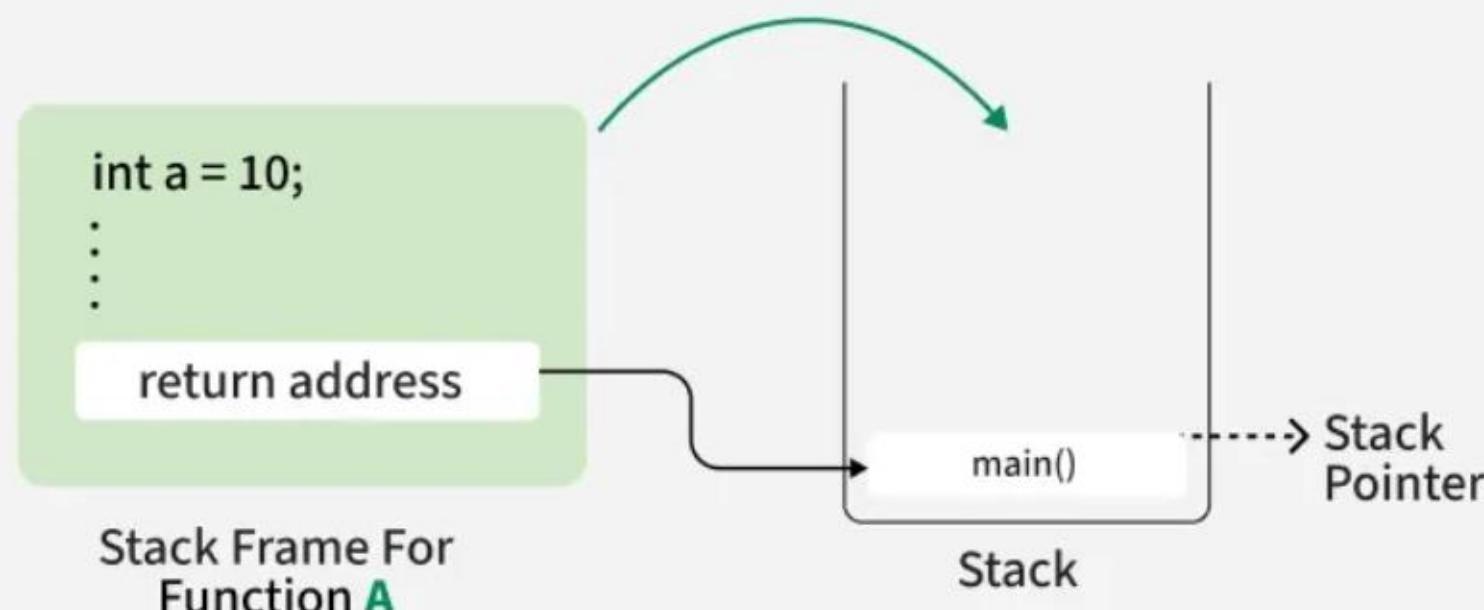
```
int main() {  
  
    // Calling function A from main  
    A();  
  
    // Calling function D from main, it will be called  
    // after A, B and C's execution  
    D();  
  
    return 0;  
}
```

## Output

```
In function C  
In function B  
In function A  
In function D
```

# Call Stack Process :

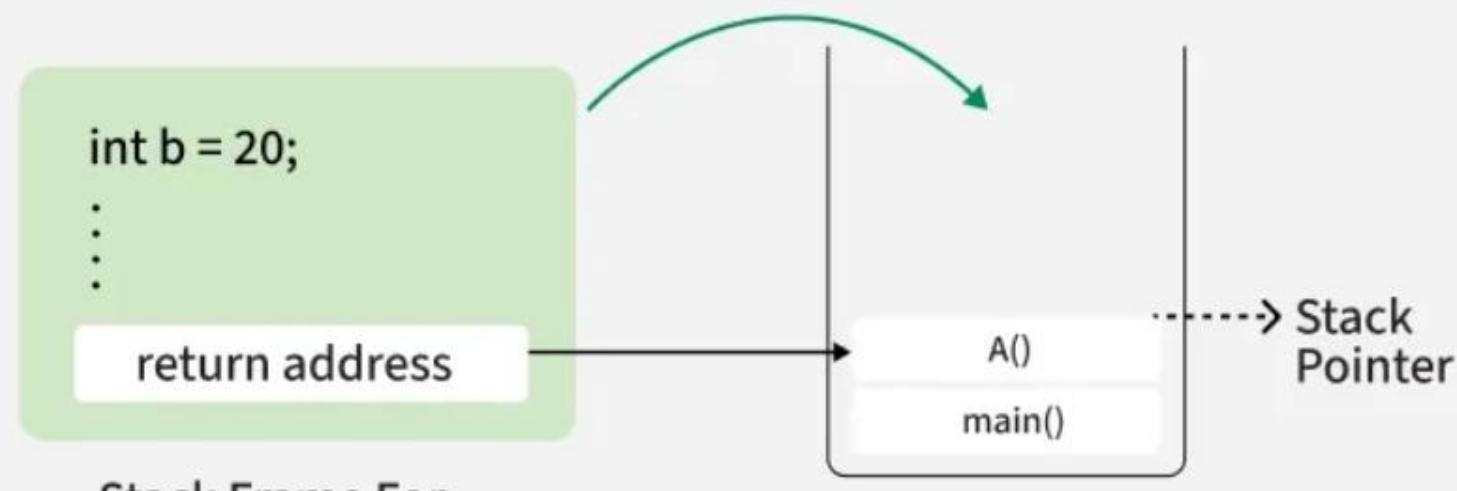
Stack frame of A() will be created and pushed into the Call Stack. It will save the address of where to return after executing A() as return address.



How Functions Are Stored in Call Stack

# Call Stack Process :

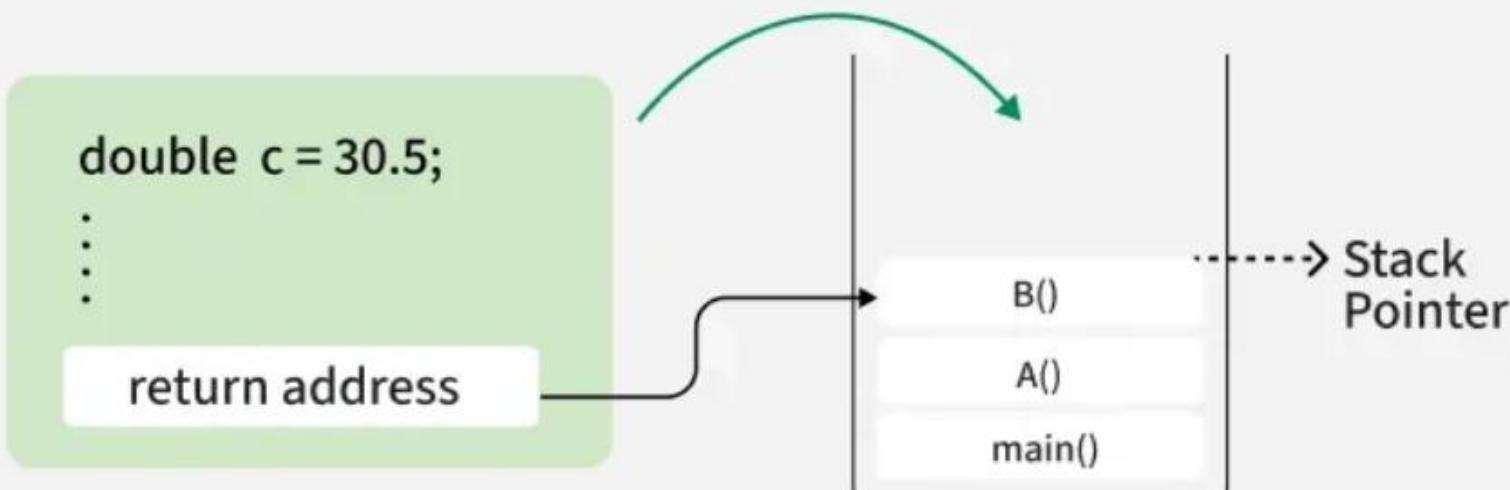
As B() is called by A(), stack frame of B() will be created and pushed to Call Stack.  
The return address will be the point after the call of b() in A()



How Functions Are Stored in Call Stack

# Call Stack Process :

As B() is called by A(), stack frame of B() will be created and pushed to Call Stack.  
The return address will be the point after the call of c() in B()



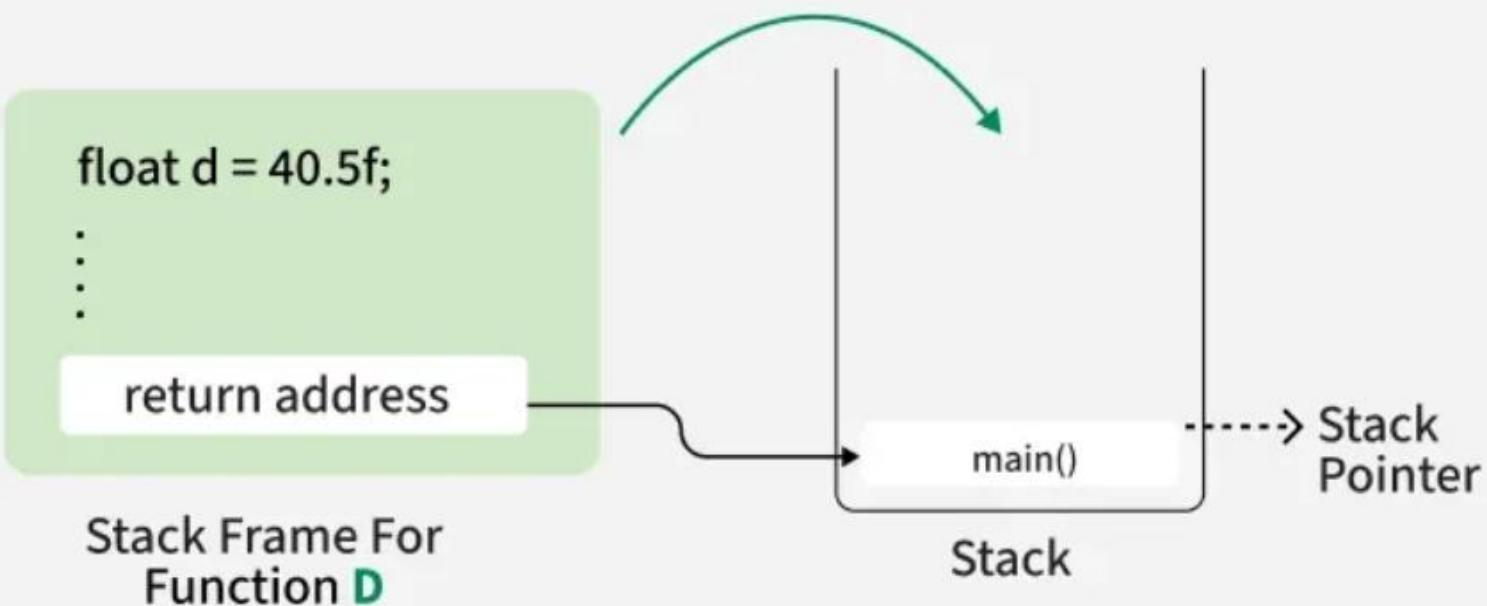
Stack Frame For  
Function C

Stack

How Functions Are Stored in Call Stack

# Call Stack Process :

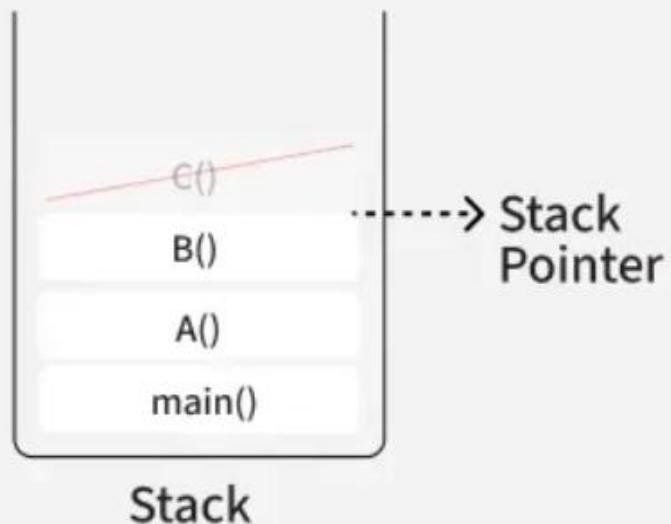
Since D() is called, its stack frame is created and pushed to the call stack just over the main()



How Functions Are Stored in Call Stack

# Call Stack Process :

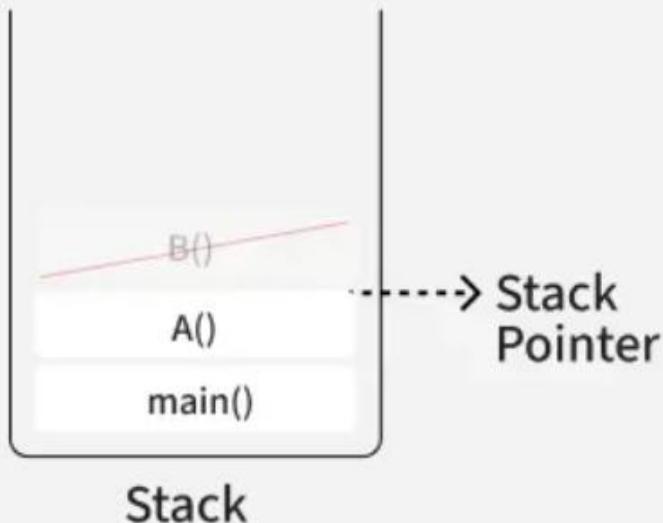
Now, C() does not call any other function. So, after its complete execution, stack frame of C() is removed from the stack and program control comes back to B() using return address.



How Functions Are Stored in Call Stack

# Call Stack Process :

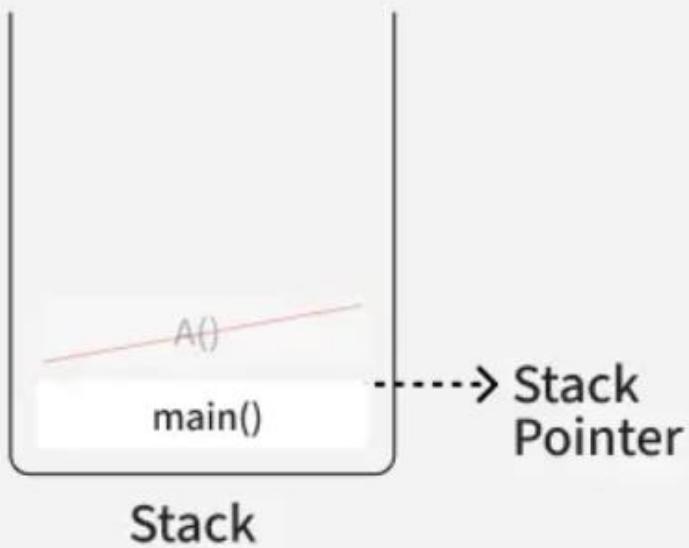
B() executes completely. So the stack frame of B() is removed from the call stack and control comes to A() using return address.



How Functions Are Stored in Call Stack

# Call Stack Process :

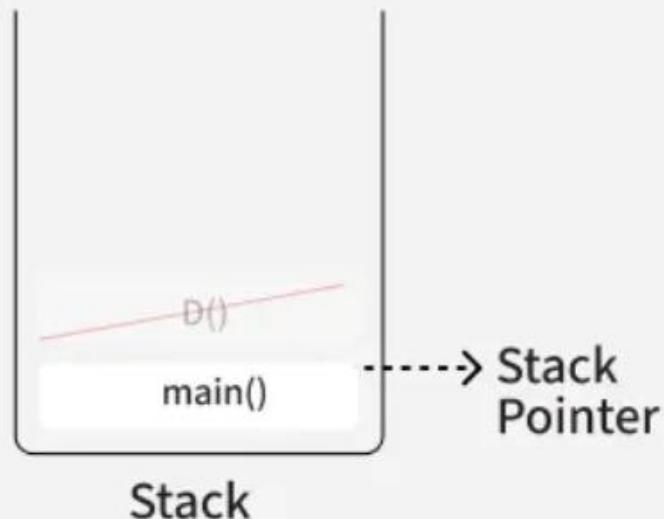
A() executes completely. So the stack frame of A() is removed from the call stack and control comes to main() using return address.



How Functions Are Stored in Call Stack

# Call Stack Process :

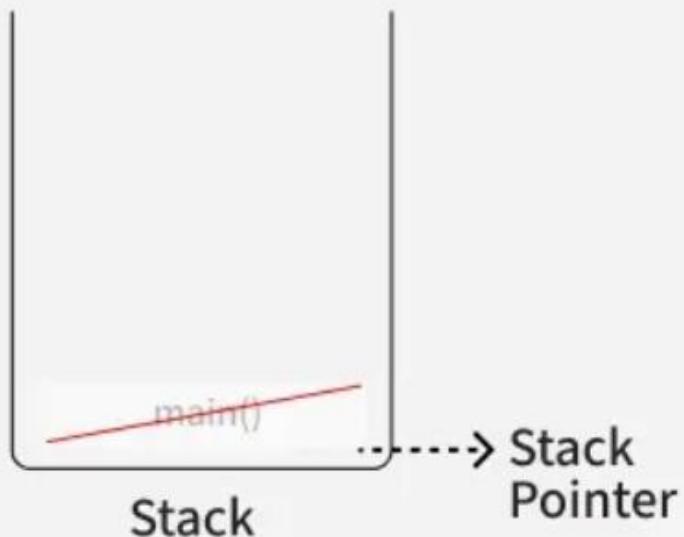
After D() executes completely, the stack frame of D() is removed and control comes back to main().



How Functions Are Stored in Call Stack

# Call Stack Process :

After the main() returns 0, the stack frame of main() is also deleted leading to the termination of the program.



How Functions Are Stored in Call Stack

# Let's Explore something New : ispunct()

- The ispunct() function checks whether a character is a punctuation character or not.
- The term "punctuation" as defined by this function includes all printable characters that are neither alphanumeric nor a space.
- For example '@', '\$', etc. This function is defined in ctype.h header file.

# Example : ispunct()

```
// Program to check punctuation
#include <stdio.h>
#include <ctype.h>
int main()
{
    // The punctuations in str are '!' and ','
    char str[] = "welcome! to GeeksForGeeks, ";

    int i = 0, count = 0;
    while (str[i]) {
        if (ispunct(str[i]))
            count++;
        i++;
    }
    printf("Sentence contains %d punctuation"
           " characters.\n", count);
    return 0;
}
```

Output:

Sentence contains 2 punctuation characters.

# Recursion in c - Basic Topic again!!

- Recursion is a technique where a problem is solved by breaking it into smaller subproblems.
- It involves a function calling itself, known as a recursive call, and the function is called a recursive function.

# Advantages of Functions in C

- Functions in C is a highly useful feature of C with many advantages as mentioned below:
- The function can reduce the repetition of the same statements in the program.
- We can achieve recursion only if functions exist.
- Using functions, the code becomes more readable by converting repetitive tasks into functions.
- Functions can call any number of times.
- Once the function is declared you can just use it without thinking about the internal working of the function.

# Disadvantages of Functions in C

- Disadvantages of Functions in C
- The following are the major disadvantages of functions in C:
- Cannot return multiple values.
- Memory and time overhead due to stack frame allocation and transfer of program control.
- Call stack of Recursion : <https://www.geeksforgeeks.org/function-call-stack-in-c/>

# Example : factorial with Recursion

- #include <stdio.h>
- // Function to calculate factorial using recursion
- long long factorial(int n) {
- if (n == 0 || n == 1) {
- return 1; // Base case
- }
- return n \* factorial(n - 1); // Recursive call
- }
- Full Example : [Practice](#)

# C Programming Langauge Structure

# What is Structure ?

- In C, a structure is a user-defined data type that can be used to group items of possibly different types into a single type.
- The '**struct**' keyword is used to define a structure.
- The items in the structure are called its member and they can be of any valid data type.

# Need of Structure!

- Structures in C are used to group related data of different types under one entity, improving code organization and efficiency.
- They help in better memory management, easier data handling, and serve as a foundation for advanced concepts like linked lists and queues.

## Example:

```
1 #include <stdio.h>
2
3 // Defining a structure
4 struct A {
5     int x;
6 }
7
8 int main() {
9
10    // Creating a structure variable
11    struct A a;
12
13    // Initializing member
14    a.x = 11;
15
16    printf("%d", a.x);
17    return 0;
18 }
```

## Explanation:

In this example, a structure **A** is defined to hold an integer member **x**.

A variable **a** of type **struct A** is created and its member **x** is initialized to **11** by accessing it using dot operator.

The value of **a.x** is then printed to the console.

## Output

```
11
```

# Syntax of Structure

- There are two steps of creating a structure in C:
- Structure Definition
- Creating Structure Variables

# Structure Definition

- A structure is defined using the `struct` keyword followed by the structure name and its members. It is also called a structure template or structure prototype, and no memory is allocated to the structure in the declaration.
- `struct structure_name {`
- `data_type1 member1;`
- `data_type2 member2;`
- `...`
- `};`
- `structure_name`: Name of the structure.
- `member1, member2, ...`: Name of the members.
- `data_type1, data_type2, ...`: Type of the members.
- Be careful not to forget the semicolon at the end.

# Creating Structure Variable

- in the order of creating a structure variable , you can create it in two ways as well!
- 1. using struct keyword
- 2. along with structure definition

# 1. using Struct Keyword

- After structure definition, we have to create variable of that structure to use it. It is similar to the any other type of variable declaration:
- `struct strcuture_name var;`

## 2. Along with structure definition

- We can also declare structure variables with structure definition.
- ```
struct structure_name {  
    ...  
}var1, var2....;
```

# Basic Operation of Structure

- 1. Access Structure Members
- To access or modify members of a structure, we use the ( . ) dot operator. This is applicable when we are using structure variables directly.
- `structure_name . member1;`
- `strcuture_name . member2;`
- In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.
- `structure_ptr -> member1`
- `structure_ptr -> member2`

# Basic Operation of Structure

- 2. Initialize Structure Members!
- Structure members cannot be initialized with the declaration. For example, the following C program fails in the compilation.
- ```
struct structure_name {  
    data_type1 member1 = value1; // COMPILER ERROR: cannot initialize members here  
    data_type2 member2 = value2; // COMPILER ERROR: cannot initialize members here  
    ...  
};
```
- The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created. So there is no space to store the value assigned.
- We can initialize structure members in 4 ways which are as follows:

# Basic Operation of Structure

- i) Default Initialization
- By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.
- `struct structure_name = {0}; // Both x and y are initialized to 0`

# Basic Operation of Structure

- ii) Initialization using Assignment Operator
- struct structure\_name str;
- str.member1 = value1;
- ....
- Note: We cannot initialize the arrays or strings using assignment operator after variable declaration.

# Basic Operation of Structure

- iii) Initialization using Initializer List
- `struct structure_name str = {value1, value2, value3 ....};`
- In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

# Basic Operation of Structure

- iv) Initialization using Designated Initializer List
- Designated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.
- `struct structure_name str ;`
- `str = { .member1 = value1, .member2 = value2, .member3 = value3 };`

```
1 #include <stdio.h>
2
3 // Defining a structure to represent a student
4 struct Student {
5     char name[50];
6     int age;
7     float grade;
8 }
9
10 int main() {
11
12     // Declaring and initializing a structure
13     // variable
14     struct Student s1 = {"Rahul", 20, 18.5};
15
16     // Designated Initializing another structure
17     struct Student s2 = {.age = 18, .name =
18         "Vikas", .grade = 22};
19
20     // Accessing structure members
21     printf("%s\t%d\t%.2f\n", s1.name, s1.age,
22         s1.grade);
23     printf("%s\t%d\t%.2f\n", s2.name, s2.age,
24         s2.grade);
25
26     return 0;
27 }
```

### Output

Rahul	20	18.50
Vikas	18	22.00

### 3. Copy Structure

- Copying structure is simple as copying any other variables. For example, s1 is copied into s2 using assignment operator.
- `s2 = s1;`
- But this method only creates a shallow copy of s1 i.e. if the structure s1 have some dynamic resources allocated by malloc, and it contains pointer to that resource, then only the pointer will be copied to s2. If the dynamic resource is also needed, then it has to be copied manually (deep copy).

# Example :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Student {
5     int id;
6     char grade;
7 };
8
9 int main() {
10     struct Student s1 = {1, 'A'};
11
12     // Create a copy of student s1
13     struct Student s1c = s1;
14
15     printf("Student 1 ID: %d\n", s1c.id);
16     printf("Student 1 Grade: %c", s1c.grade);
17     return 0;
18 }
```

## Output

```
Student 1 ID: 1
Student 1 Grade: A
```

## 4. Passing Structure to a function

- Structure can be passed to a function in the same way as normal variables.
- Though, it is recommended to pass it as a pointer to avoid copying a large amount of data.

# Example : Practice

C Input and Output   C Control Flow   C Functions   C Arrays   C Strings   C Pointers   C Preprocessors   C File Handling

```
2
3 // Structure definition
4 struct A {
5     int x;
6 }
7
8 // Function to increment values
9 void increment(struct A a, struct A* b) {
10    a.x++;
11    b->x++;
12 }
13
14 int main() {
15    struct A a = { 10 };
16    struct A b = { 10 };
17
18    // Passing a by value and b by pointer
19    increment(a, &b);
20
21    printf("a.x: %d \tb.x: %d", a.x, b.x);
22    return 0;
23 }
```

## Output

```
a.x: 10          b.x: 11
```

## Example :2

```
// Needs the knowledge of pointer!
#include<stdio.h>
struct books{
    int a;
};
// function declaration.
int returning_data_from_structure(struct books b1,struct books *b2)
{
    printf("\n The copy of The Structure Variable without pointer : %d",b1.a);
    printf("\n The Variable value from structure pointer : %d",(*b2).a);

    if(*b2.a == 10)
    {
        return 1;
    }
}
```

```
int main()
{
    struct books b1 ={10};
    struct books b2={10};

    int result = returning_data_from_struct(b1,&b2);

    if(result == 1)
    {
        printf("\n\n\n Pointer with Structure is working Successfully!!!!");
    }
    return 0;
}
```

The Copy of The Structure Variable without pointer : 10  
The Variable value from structure pointer : 10

Pointer with Structure is working Successfully!!!!

# 5. typedef for Structures

- The `typedef` keyword is used to define an alias for the already existing datatype.
- In structures, we have to use the `struct` keyword along with the structure name to define the variables.
- Sometimes, this increases the length and complexity of the code.
- We can use the `typedef` to define some new shorter name for the structure.

# Example :

```
1 #include <stdio.h>
2
3 // Defining structure
4 typedef struct {
5     int a;
6 } str1;
7
8 // Another way of using typedef with structures
9 typedef struct {
10     int x;
11 } str2;
12
13 int main() {
14
15     // Creating structure variables using new names
16     str1 var1 = { 20 };
17     str2 var2 = { 314 };
18
19     printf("var1.a = %d\n", var1.a);
20     printf("var2.x = %d\n", var2.x);
21     return 0;
22 }
```

## Explanation:

In this code, **str1** and **str2** are defined as aliases for the unnamed structures, allowing the creation of structure variables (**var1** and **var2**) using these new names. This simplifies the syntax when declaring variables of the structure.

## Output

```
var1.a = 20
var2.x = 314
```

# Size of Structure and understanding the concept of Slack Bytes

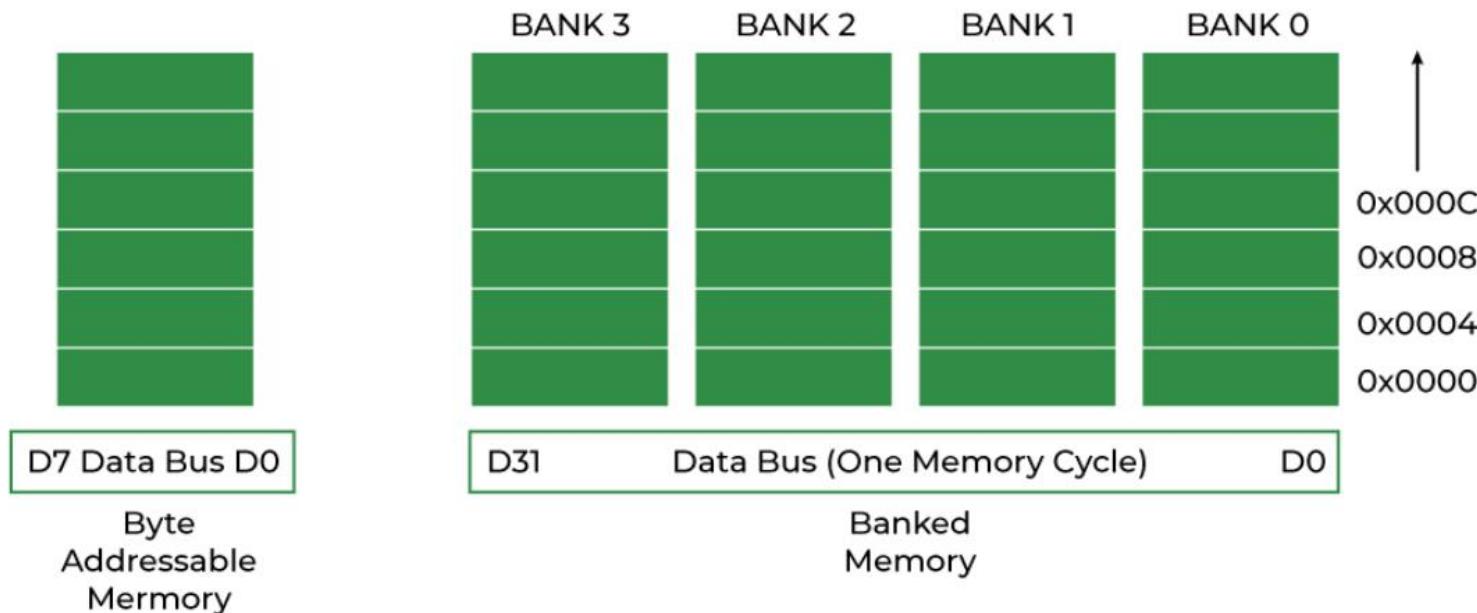
- Technically, the size of the structure in C should be the sum of the sizes of its members.
- But it may not be true for most cases. The reason for this is Structure Padding.
- **Structure padding** : is the concept of adding multiple empty bytes in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.

# Continue...

- There are some situations where we need to pack the structure tightly by removing the empty bytes. In such cases, we use Structure Packing. C language provides two ways for structure packing:
- Using `#pragma pack(1)`
- Using `__attribute__((packed))`

# Structure Member Alignment

- Every data type in C will have alignment requirements (in fact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32-bit machine, the processing word size will be 4 bytes.



- Historically memory is byte addressable and arranged sequentially. If the memory is arranged as a single bank of one-byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of an integer in one memory cycle. To take such advantage, the memory will be arranged as a group of 4 banks as shown in the above figure.
- The memory addressing still be sequential. If bank 0 occupies an address  $X$ , bank 1, bank 2 and bank 3 will be at  $(X + 1)$ ,  $(X + 2)$ , and  $(X + 3)$  addresses. If an integer of 4 bytes is allocated on  $X$  address ( $X$  is a multiple of 4), the processor needs only one memory cycle to read the entire integer. Whereas, if the integer is allocated at an address other than a multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycles to fetch the data.



Layout of misaligned data (0X01ABCDEF)

# Structure Padding

- Structure padding is the addition of some empty bytes of memory in the structure to naturally align the data members in the memory.
- It is done to minimize the CPU read cycles to retrieve different data members in the structure.

Try to calculate the size of the following structures:

[Skip to content](#)

[C Input and Output](#) [C Control Flow](#) [C Functions](#) [C Arrays](#) [C Strings](#) [C Pointers](#) [C Prepro](#)

```
2  typedef struct structa_tag {  
3      char c;  
4      short int s;  
5  } structa_t;  
6  
7  // structure B  
8  typedef struct structb_tag {  
9      short int s;  
10     char c;  
11     int i;  
12  } structb_t;  
13  
14 // structure C  
15 typedef struct structc_tag {  
16     char c;  
17     double d;  
18     int s;  
19  } structc_t;  
20  
21 // structure D  
22 typedef struct structd_tag {  
23     double d;  
24     int s;  
25     char c;  
26  } structd_t;
```

Calculating the size of each structure by directly adding the size of all the members, we get:

- **Size of Structure A** = Size of (char + short int)  
 $= 1 + 2 = 3.$
- **Size of Structure B** = Size of (short int + char + int)  $= 2 + 1 + 4 = 7.$
- **Size of Structure C** = Size of (char + double + int)  $= 1 + 8 + 4 = 13.$
- **Size of Structure A** = Size of (double + int + char)  $= 8 + 4 + 1 = 13.$

# Now Let's Confirm the size of Structure

```
1 // C Program to demonstrate the structure padding property
2 #include <stdio.h>
3
4 // Alignment requirements
5 // (typical 32 bit machine)
6
7 // char      1 byte
8 // short int 2 bytes
9 // int       4 bytes
10 // double    8 bytes
11
12 // structure A
13 typedef struct structa_tag {
14     char c;
15     short int s;
16 } structa_t;
17
18 // structure B
19 typedef struct structb_tag {
20     short int s;
21     char c;
22     int i;
23 } structb_t;
24
25 // structure C
26 typedef struct structc_tag {
27     char c;
28 } structc_t;
29
30 } structc_t;
31
32 // structure D
33 typedef struct structd_tag {
34     double d;
35     int s;
36     char c;
37 } structd_t;
38
```

Skip to content

C Input and Output   C Control Flow   C Functions   C Arrays   C Strings   C Pointers   C Preprocessors

```
38
39     int main()
40     {
41         printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
42         printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
43         printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
44         printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));
45
46     return 0;
47 }
```

## Output

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

As we can see, the size of the structures is different from those we calculated.

- This is because of the alignment requirements of various data types, every member of the structure should be naturally aligned. The members of the structure are allocated sequentially in increasing order.
- now understanding the reason that why it is happening !?

# For Structure A

## Structure A

The `structa_t` first element is `char` which is one byte aligned, followed by `short int`. `short int` is 2 bytes aligned. If the `short int` element is immediately allocated after the `char` element, it will start at an odd address boundary. The compiler will insert a padding byte after the `char` to ensure `short int` will have an address multiple of 2 (i.e. 2 byte aligned). The total size of `structa_t` will be,

```
sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.
```

# For Structure B

## Structure B

The first member of *structb\_t* is short int followed by char. Since char can be on any byte boundary no padding is required between short int and char, in total, they occupy 3 bytes. The next member is int. If the int is allocated immediately, it will start at an odd byte boundary. We need 1-byte padding after the char member to make the address of the next int member 4-byte aligned. On total,

the *structb\_t* requires ,  $2 + 1 + 1 \text{ (padding)} + 4 = 8$  bytes.

# For Structure C

**Structure C - Every structure will also have alignment requirements**

Applying same analysis, `structc_t` needs `sizeof(char)` + 7-byte padding + `sizeof(double)` + `sizeof(int)` =  $1 + 7 + 8 + 4 = 20$  bytes. However, the `sizeof(structc_t)` is 24 bytes. It is because, along with structure members, structure type variables will also have natural alignment. Let us understand it by an example. Say, we declared an array of `structc_t` as shown below

```
structc_t structc_array[3];
```

Assume, the base address of `structc_array` is 0x0000 for easy calculations. If the `structc_t` occupies 20 (0x14) bytes as we calculated, the second `structc_t` array element (indexed at 1) will be at  $0x0000 + 0x0014 = 0x0014$ . It is the start address of the index 1 element of the array. The double member of this `structc_t` will be allocated on  $0x0014 + 0x1 + 0x7 = 0x001C$  (decimal 28) which is not multiple of 8 and conflicts with the alignment requirements of double. As we mentioned at the top, the alignment requirement of double is 8 bytes.

In order to avoid such misalignment, **the compiler introduces alignment requirements to every structure**. It will be as that of the largest member of the structure. In our case alignment of `structa_t` is 2, `structb_t` is 4 and `structc_t` is 8. If we need nested structures, the size of the largest inner structure will be the alignment of an immediate larger structure.

In `structc_t` of the above program, there will be a padding of 4 bytes after the `int` member to make the structure size multiple of its alignment. Thus the size of (`structc_t`) is 24 bytes. It guarantees correct alignment even in arrays.

# For Structure D

## Structure D

In a similar way, the size of the structure D is :

```
sizeof(double) + sizeof(int) + sizeof(char) + padding(3) = 8 + 4 + 1 + 3 = 16 bytes
```

# So, How to Reduce the Structure Padding ?

- By now, it may be clear that padding is unavoidable. There is a way to minimize padding.
- The programmer should declare the structure members in their increasing/decreasing order of size.
- An example is `structd_t` given in our code, whose size is 16 bytes in lieu of 24 bytes of `structc_t`.

# What is Structure padding ?

- What is Structure Packing?
- Sometimes it is mandatory to avoid padded bytes among the members of the structure. For example, reading contents of ELF file header or BMP or JPEG file header. We need to define a structure similar to that of the header layout and map it. However, care should be exercised in accessing such members. Typically reading byte by byte is an option to avoid misaligned exceptions but there will be a hit on performance.
- Most of the compilers provide nonstandard extensions to switch off the default padding like pragmas or command line switches. Consult the documentation of the respective compiler for more details.
- In GCC, we can use the following code for structure packing:
- #pragma pack(1)
- or
- struct name {
- ...
- } \_\_attribute\_\_((packed));

# Example : Structure Packing!

```
// C Program to demonstrate the structure packing
#include <stdio.h>
#pragma pack(1)

// structure A
typedef struct structa_tag {
    char c;
    short int s;
} structa_t;

// structure B
typedef struct structb_tag {
    short int s;
    char c;
    int i;
} structb_t;

// structure C
typedef struct structc_tag {
    char c;
    double d;
    int s;
} structc_t;

// structure D
typedef struct structd_tag {
    double d;
    int s;
    char c;
} structd_t;

int main()
{
    printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

## Output

```
sizeof(structa_t) = 3
sizeof(structb_t) = 7
sizeof(structc_t) = 13
sizeof(structd_t) = 13
```

# How to know the displacement ?

- void argument\_alignment\_check( char c1, char c2 )
- {
- // Considering downward stack
- // (on upward stack the output will be negative)
- printf("Displacement %d\n", (int)&c2 - (int)&c1);
- }

# Additional :

**What will happen if we try to access misaligned data?**

*It depends on the processor architecture. If the access is misaligned, the processor automatically issues sufficient memory read cycles and packs the data properly onto the data bus. The penalty is on performance. Whereas few processors will not have the last two address lines, which means there is no way to access the odd byte boundary. Every data access must be aligned (4 bytes) properly. Misaligned access is a critical exception on such processors. If the exception is ignored, read data will be incorrect and hence the results.*

**Is there any way to query the alignment requirements of a data type?**

*Yes. Compilers provide non-standard extensions for such needs. For example, `__alignof()` in Visual Studio helps in getting the alignment requirements of data type. Read MSDN for details.*

# Additional :

**When memory reading is efficient in reading 4 bytes at a time on a 32-bit machine, why should a double type be aligned on an 8-byte boundary?**

*It is important to note that most of the processors will have a math co-processor, called Floating Point Unit (FPU). Any floating point operation in the code will be translated into FPU instructions. The main processor is nothing to do with floating-point execution. All this will be done behind the scenes.*

*As per standard, the double type will occupy 8 bytes. And, every floating point operation performed in FPU will be of 64-bit length. Even float types will be promoted to 64 bits prior to execution.*

*The 64-bit length of FPU registers forces double type to be allocated on an 8-byte boundary. I am assuming (I don't have concrete information) in the case of FPU operations, data fetch might be different, I mean the data bus since it goes to FPU. Hence, the address decoding will be different for double types (which are expected to be on an 8-byte boundary). It means the address decoding circuits of the floating point unit will not have the last 3 pins.*

in the Summary Let's Understand the core concept of Slack bytes in c programming!!!

- Definition of Slack Bytes : Slack bytes in C refer to unused memory spaces within a structure that arise due to alignment requirements.
- These bytes are introduced by the compiler to ensure that data members are stored at memory addresses that comply with the alignment rules of the system.

# Why Do Slack Bytes Exist?

- **1. Alignment Requirements:** Different data types have specific alignment needs (e.g., int may require 4-byte alignment, double may require 8-byte alignment).
- **2. Padding Between Members:** If a smaller data type (like char) is followed by a larger one (like double), the compiler inserts extra bytes to align the larger type correctly.
- **3. Efficient Memory Access:** Some processors access memory faster when data is aligned properly. Misaligned data can lead to performance penalties or even hardware exceptions.

# Example :

## Example of Slack Bytes

C

 Copy

```
#include <stdio.h>

struct Example {
    char a;      // 1 byte
    int b;       // 4 bytes (requires 4-byte alignment)
    double c;   // 8 bytes (requires 8-byte alignment)
};

int main() {
    printf("Size of struct Example: %lu bytes\n", sizeof(struct Example));
    return 0;
}
```

# Mamory Layout

## Memory Layout

Member	Size	Address
char a	1 byte	0x0000
Padding	3 bytes	0x0001 - 0x0003
int b	4 bytes	0x0004 - 0x0007
double c	8 bytes	0x0008 - 0x000F

Here, **3 slack bytes** are added after `char a` to align `int b` correctly, ensuring efficient memory access.

# Key Takeaways

- Slack bytes **do not store useful data** but help maintain proper alignment.
- They **increase structure size** beyond the sum of individual member sizes.
- Some compilers allow **packing** (`#pragma pack`) to reduce slack bytes, but this may lead to **performance issues**.

# More about Slack Bytes : [More](#)

## Need for Slack Bytes?

In computers and technology ***speed*** is a very important factor. We always try to make designs and algorithms in such a way that the access to the memory is much faster or the solution is much faster to obtain. The microprocessors access the data that is present in the even addresses faster than that of the data that is present in the odd addresses. So, we try to allocate the memory of the data in such a way the starting address of the particular variable is in the even address.

Microprocessors access the data that is stored in the even address faster than the data that is stored in the odd address. So, it becomes the responsibility of the compilers to allocate the memory for the members such that they have even addresses so that the data can be accessed very fast. This leads to the concept of slack bytes.

# More about Slack Bytes : [More](#)

## SLACK BYTES:

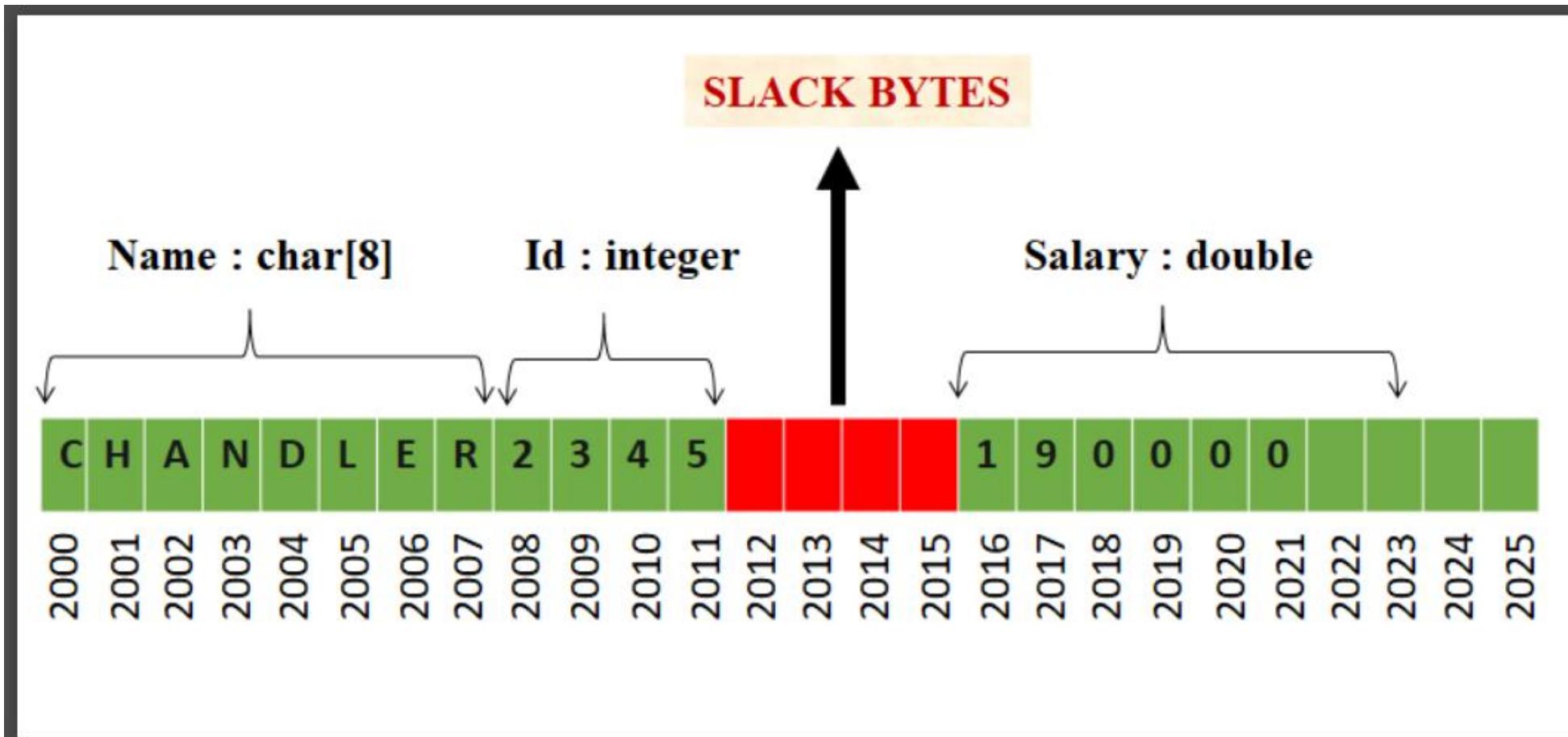
The optimized compilers will always assign even addresses to the members of the structure so that the data can be accessed very fast. The even addresses may be multiples of 2, 4, 8 or 16. This introduces some unused bytes or holes between the boundaries of some members. These ***unused bytes or holes between boundaries of the members of structure are known as slack bytes.***

The slack bytes do not contain any useful information and they are actually a wastage of memory space. But the access of the data would be much faster in the case of slack bytes concept. When the slack bytes are used, the size of the structure would be greater or the same as the sum of the size of the individual members of the structure. Some optimized [compilers](#) assign the address such that the addresses of each member will be multiple of the size of the largest data type of the structure.

# Example :

- Example: Consider the following declaration of a structure:
- struct EMPLOYEE
- {
- char name[8];
- int id;
- double salary;
- };

# Understanding by the Daigram as well!



# Nested Structures

- Nested Structures
- In C, a nested structure refers to a structure that contains another structure as one of its members.
- This allows you to create more complex data types by grouping multiple structures together, which is useful when dealing with related data that needs to be grouped within a larger structure.

# Ways to define Nested Structures

- There are two ways in which we can nest one structure into another:
- Embedded Structure Nesting: The structure being nested is also declared inside the parent structure.
- Separate Structure Nesting: Two structures are declared separately and then the member structure is nested inside the parent structure.

# 1. By separate nested structure

- A nested structure allows the creation of complex data types according to the requirements of the program.
- Syntax:

```
struct name_1 {  
    member1;  
    member2;  
    membern;  
    struct name_2 {  
        member_1;  
        member_2;  
        member_n;  
    }, var1  
} var2;
```

## Access Syntax :

- Outer\_Structure.Nested\_Structure.data member
- Example:
  - Consider there are two structures Employee (depended structure) and another structure called Organisation(Outer structure).
  - The structure Organisation has the data members like organisation\_name,organisation\_number.
  - The Employee structure is nested inside the structure Organisation and it has the data members like employee\_id, name, salary.

# Understanding The Access Syntax : Example

For accessing the members of Organisation and Employee following syntax will be used:

```
org.emp.employee_id;  
org.emp.name;  
org.emp.salary;  
org.organisation_name;  
org.organisation_number;
```



Here, org is the structure variable of the outer structure Organisation and emp is the structure variable of the inner structure Employee.

# Understand Full Example

- [Refer this link to understand the full example](#)

## 2. By Embedded nested structure:

- In this method the nested structure is declared inside the outer structure, this method allows to declare structure inside a structure and requires fewer lines of code.

```
1 #include <stdio.h>
2
3 // Declaration of the outer
4 // structure
5 struct Organisation {
6     char organisation_name[20];
7     char org_number[20];
8
9     // Declaration of the employee
10    // structure
11    struct Employee {
12        int employee_id;
13        char name[20];
14        int salary;
15
16        // This will cause error because
17        // datatype struct is present ,
18        // but structure variable is missing.
19    };
20 };
21
22 int main() {
23
24     // Structure variable of organisation
25     struct Organisation org;
26     printf("%ld", sizeof(org.Employee.name));
27 }
```

## Output:

```
./Solution.c: In function 'main':  
./Solution.c:28:27: error: 'struct Organisation' has no member named 'Employee'  
    printf("%ld", sizeof(org.Employee.name));
```

**Note:** Whenever an embedded nested structure is created, the variable declaration is compulsory at the end of the inner structure, which acts as a member of the outer structure. It is compulsory that the **structure variable** is created at the end of the inner structure.

[Explore Full Example Here](#)

# How to Access Nested Structure ?

- 1. Using Normal Variable : Outer and inner structure variables are declared as normal variables and the data members of the outer structure are accessed using a single dot(.) and the data members of the inner structure are accessed using the two dots. Below is the C program to implement this concept.
- 2. Using Pointer Variable: One normal variable and one pointer variable of the structure are declared to explain the difference between the two. In the case of the pointer variable, a combination of dot(.) and arrow(->) will be used to access the data members. Below is the C program to implement the above approach:

# 1. Using Normal Variable

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Declaration of inner structure
5 struct college_details {
6     int college_id;
7     char college_name[50];
8 }
9
10 // Declaration of Outer structure
11 struct student_detail {
12     int student_id;
13     char student_name[20];
14     float cgpa;
15
16     // Inner structure variable
17     struct college_details clg;
18 } stu;
19
20
21 int main() {
22     struct student_detail stu = {12, "Kathy", 7.8,
23                                 {14567, "GeeksforGeeks"}};
24
25     // Printing the details
26     printf("College ID : %d \n", stu.clg.college_id);
27     printf("College Name : %s \n", stu.clg.college_name);
28     printf("Student ID : %d \n", stu.student_id);
29     printf("Student Name : %s \n", stu.student_name);
30     printf("Student CGPA : %f \n", stu.cgpa);
31
32     return 0;
33 }
```

## Output

```
College ID : 14567
College Name : GeeksforGeeks
Student ID : 12
Student Name : Kathy
Student CGPA : 7.800000
```

# 1. Using Pointer Variable

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Declaration of inner structure
5 struct college_details {
6     int college_id;
7     char college_name[50];
8 }
9
10
11 // Declaration of Outer structure
12 struct student_detail {
13     int student_id;
14     char student_name[20];
15     float cgpa;
16
17     // Inner structure variable
18     struct college_details clg;
19 } stu, *stu_ptr;
20
21 int main() {
22     struct student_detail stu = {12, "Kathy", 7.8,
23         {14567, "GeeksforGeeks"}
24     };
25
26     stu_ptr = &stu;
27
28     printf("College ID : %d \n", stu_ptr->clg.college_id);
29     printf("College Name : %s \n", stu_ptr->clg.college_name);
30     printf("Student ID : %d \n", stu_ptr->student_id);
31     printf("Student Name : %s \n", stu_ptr->student_name);
32     printf("Student CGPA : %f \n", stu_ptr->cgpa);
```

## Output

```
College ID : 14567
College Name : GeeksforGeeks
Student ID : 12
Student Name : Kathy
Student CGPA : 7.800000
```

# Drawback of Nested Structure

- The drawback in nested structures are:
- Independent existence not possible: It is important to note that structure Employee doesn't exist on its own. One can't declare structure variable of type struct Employee anywhere else in the program.
- Cannot be used in multiple data structures: The nested structure cannot be used in multiple structures due to the limitation of declaring structure variables within the main structure. So, the most recommended way is to use a separate structure and it can be used in multiple data structures

# Drawback of Nested Structure

- Note: Nesting of the same structure within itself is not allowed.
- Example:

```
struct student
{
    char name[50];
    char address[100];
    int roll_no;
    struct student geek; // Invalid
}
```

This will cause error as we cannot nest a structure in itself.

# Passing a Structure to the function!

- Passing nested structure to function
- A nested structure can be passed into the function in two ways:
  - 1. Pass the nested structure variable at once.
  - 2. Pass the nested structure members as an argument into the function.
- Let's discuss each of these ways in detail.

# 1. Pass the nested structure variable at once.

- Just like other variables, a nested structure variable can also be passed to the function. Below is the C program to implement this concept:

## Example :

```
#include <stdio.h>

// Declaration of the inner
// structure
struct Employee {
    int employee_id;
    char name[20];
    int salary;
};

// Declaration of the Outer
// structure
struct Organisation {
    char organisation_name[20];
    char org_number[20];

    // Nested structure
    struct Employee emp;
};


```

```
void show(struct Organisation);

int main() {
    struct Organisation org = {"GeeksforGeeks", "GFG111",
                                {278, "Paul", 5000}
};

    // Organisation structure variable
    // is passed to function show
    show(org);
}

void show(struct Organisation org ) {
    printf("Printing the Details :\n");
    printf("Organisation Name : %s\n",
           org.organisation_name);
    printf("Organisation Number : %s\n",
           org.org_number);
    printf("Employee id : %d\n",
           org.emp.employee_id);
    printf("Employee name : %s\n",
           org.emp.name);
    printf("Employee Salary : %d\n",
           org.emp.salary);
}
```

# Output

## Output

```
Printing the Details :
```

```
Organisation Name : GeeksforGeeks
```

```
Organisation Number : GFG111
```

```
Employee id : 278
```

```
Employee name : Paul
```

```
Employee Salary : 5000
```

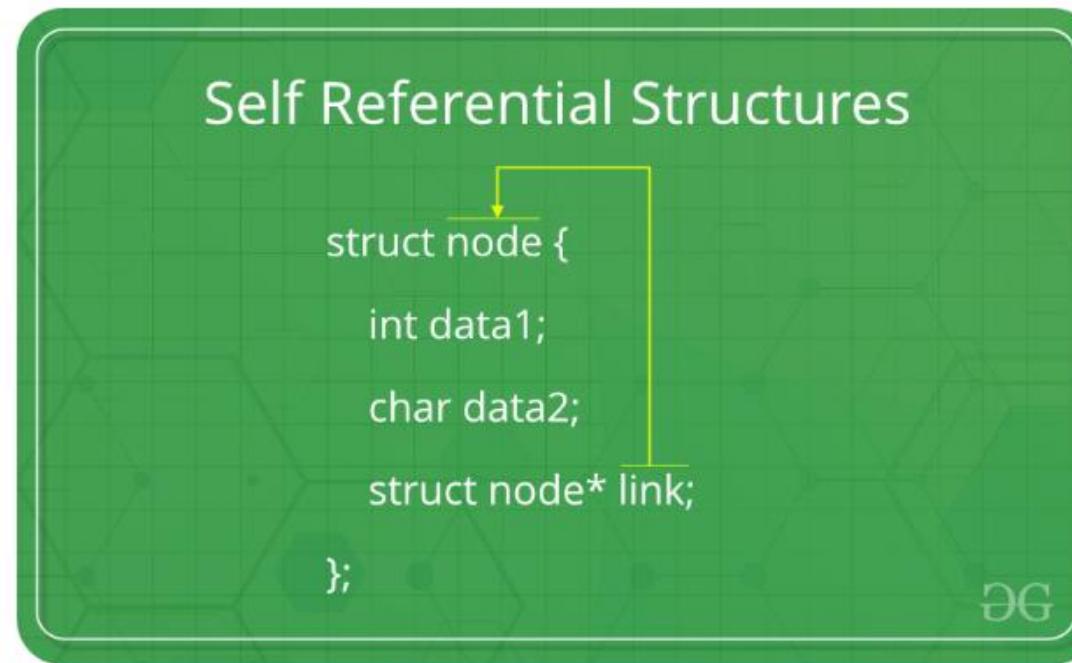
2. Pass the nested structure members as arguments into the function:

- Consider the following example to pass the structure member of the employee to a function display() which is used to display the details of an employee.
- [Refer full example and Output.](#)
- Note : Structure Pointer will be covered in the section of Pointer!!!

# Self-Referential Structures

Note : Pls refer this topic after understanding the pointers!!!

- Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



- In other words, structures pointing to the same type of structures are self-referential in nature

# Self-Referential Structures

Note : Pls refer this topic after understanding the pointers!!!

```
1 struct node {  
2     int data1;  
3     char data2;  
4     struct node* link;  
5 };  
6  
7 int main()  
8 {  
9     struct node ob;  
10    return 0;  
11 }
```

In the above example 'link' is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.

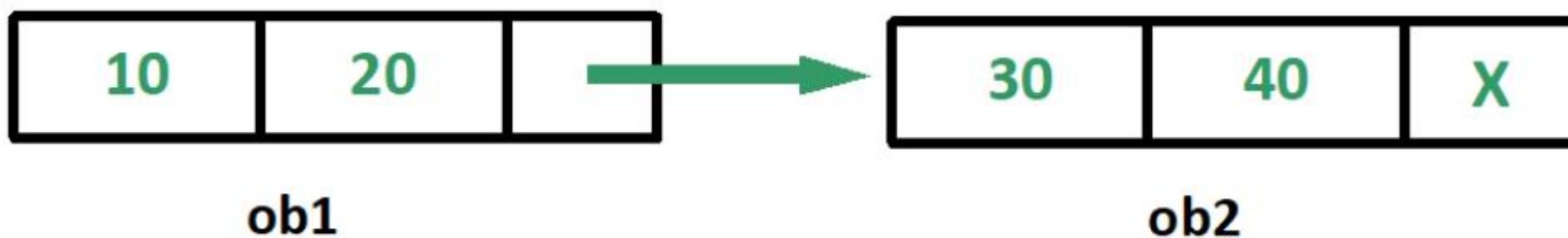
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

# Types of Self-Referential Structures

- Types of Self Referential Structures
  - 1. Self Referential Structure with Single Link
  - 2. Self Referential Structure with Multiple Links

# 1. Self Referential Structure with Single Link:

- These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



# Example:

```
#include <stdio.h>

struct node {
    int data1;
    char data2;
    struct node* link;
};

int main()
{
    struct node ob1; // Node1

    // Initialization
    ob1.link = NULL;
    ob1.data1 = 10;
    ob1.data2 = 20;

    struct node ob2; // Node2

    // Initialization
    ob2.link = NULL;
    ob2.data1 = 30;
    ob2.data2 = 40;

    ob1.link = &ob2; // Linking ob1 and ob2

    // Accessing data members of ob2 using ob1
    printf("%d", ob1.link->data1);
    printf("\n%d", ob1.link->data2);

    return 0;
}
```

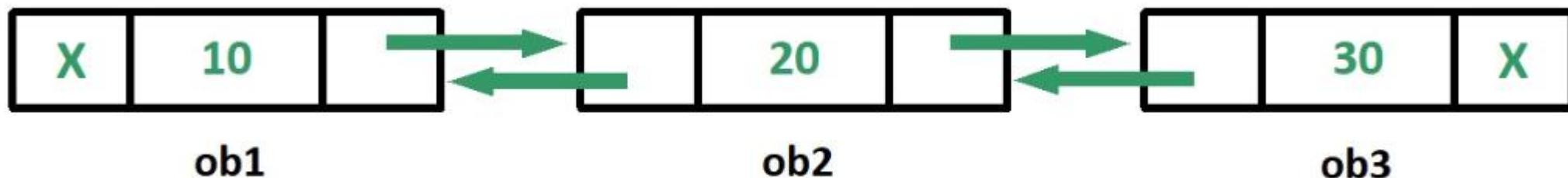
## Output

30

40

## 2. Self Referential Structure with Multiple Links:Practice

- Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.



# The Use cases of Self-Referencial Structures!

- Stack
  - Queue
  - Linked List
  - Tree
  - Graph etc.
- 
- We will look these all topics in DSA in C!

# Bit Fields in Structures!!!

- In C, we can specify the size (in bits) of the structure and union members.
- The idea of bit-field is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.
- C Bit fields are used when the storage of our program is limited.

# Bit Fields in Structures!!!

- Need of Bit Fields in C
- Reduces memory consumption.
- To make our program more efficient and flexible.
- Easy to Implement.

# Declaration of C Bit Fields

- Bit-fields are variables that are defined using a predefined width or size. Format and the declaration of the bit-fields in C are shown below:
- Syntax of C Bit Fields
- struct
- {
- data\_type member\_name : width\_of\_bit-field;
- };
- where,
- data\_type: It is an integer type that determines the bit-field value which is to be interpreted. The type may be int, signed int, or unsigned int.
- member\_name: The member name is the name of the bit field.
- width\_of\_bit-field: The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

# Application of C Bit fields

- If storage is limited, we can go for bit-field.
- When devices transmit status or information encoded into multiple bits for this type of situation bit-field is most efficient.
- Encryption routines need to access the bits within a byte in that situation bit-field is quite useful.
- struct date
- {
- // month has value between 0 and 15,
- // so 4 bits are sufficient for month variable.
- int month : 4;
- };

# Example :

Below is the same code but with signed integers:

```
1 // C program to demonstrate use of Bit-fields
2 #include <stdio.h>
3
4 // Space optimized representation of the date
5 struct date {
6     // d has value between 0 and 31, so 5 bits
7     // are sufficient
8     int d : 5;
9
10    // m has value between 0 and 15, so 4 bits
11    // are sufficient
12    int m : 4;
13
14    int y;
15};
16
17 int main()
18 {
19     printf("Size of date is %lu bytes\n",
20           sizeof(struct date));
21     struct date dt = { 31, 12, 2014 };
22     printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
23     return 0;
24 }
```

## Output

```
Size of date is 8 bytes
Date is -1/-4/2014
```

# Interesting Facts About C Bit Fields

- 1. A special unnamed bit field of size 0 is used to force alignment on the next boundary.

```
// C Program to illustrate the use of forced alignment in
// structure using bit fields
#include <stdio.h>

// A structure without forced alignment
struct test1 {
    unsigned int x : 5;
    unsigned int y : 8;
};

// A structure with forced alignment
struct test2 {
    unsigned int x : 5;
    unsigned int : 0;
    unsigned int y : 8;
};

int main()
{
    printf("Size of test1 is %lu bytes\n",
           sizeof(struct test1));
    printf("Size of test2 is %lu bytes\n",
           sizeof(struct test2));
    return 0;
}
```

### Output

```
Size of test1 is 4 bytes
Size of test2 is 8 bytes
```

## 2. We cannot have pointers to bit field members as they may not start at a byte boundary.

**Example:** The below code demonstrates that taking the address of a bit field member directly is not allowed.

```
// C program to demonstrate that the pointers cannot point
// to bit field members
#include <stdio.h>
struct test {
    unsigned int x : 5;
    unsigned int y : 5;
    unsigned int z;
};
int main()
{
    struct test t;

    // Uncommenting the following line will make
    // the program compile and run
    printf("Address of t.x is %p", &t.x);

    // The below line works fine as z is not a
    // bit field member
    printf("Address of t.z is %p", &t.z);
    return 0;
}
```

```
prog.c: In function 'main':
prog.c:14:1: error: cannot take address of bit-field 'x'
printf("Address of t.x is %p", &t.x);
^
```

### 3. It is implementation-defined to assign an out-of-range value to a bit field member.

```
// C Program to show what happens when out of range value
// is assigned to bit field member
#include <stdio.h>

struct test {
    // Bit-field member x with 2 bits
    unsigned int x : 2;
    // Bit-field member y with 2 bits
    unsigned int y : 2;
    // Bit-field member z with 2 bits
    unsigned int z : 2;
};

int main()
{
    // Declare a variable t of type struct test
    struct test t;
    // Assign the value 5 to x (2 bits)
    t.x = 5;

    // Print the value of x
    printf("%d", t.x);

    return 0;
}
```

## 4. Array of bit fields is not allowed.

```
// C Program to illustrate that we cannot have array bit
// field members
#include <stdio.h>

// structure with array bit field
struct test {
    unsigned int x[10] : 5;
};

int main() {}
```

[Most Asked Question for Bit feilds in an Interviews!](#)

# Uses of Structures in C

- C structures are used for the following:
- The structure can be used to define the custom data types that can be used to create some complex data types such as dates, time, complex numbers, etc. which are not present in the language.
- It can also be used in data organization where a large amount of data can be stored in different fields.
- Structures are used to create data structures such as trees, linked lists, etc.
- They can also be used for returning multiple values from a function.

# Disadvantages of Structures

- 1. **\*\*No Data Hiding\*\*** – All members are public, leading to potential unintended modifications.
- 2. **\*\*No Member Functions\*\*** – Requires separate functions to manipulate structure data.
- 3. **\*\*Memory Overhead\*\*** – Padding and alignment can waste memory.
- 4. **\*\*No Inheritance\*\*** – Limits code reuse and extensibility.
- 5. **\*\*Limited Type Safety\*\*** – Accidental modifications can cause unexpected behavior.
- 6. **\*\*No Polymorphism\*\*** – Unlike C++, structures don't support dynamic method binding.

# Array vs Structures

Feature	Array	Structure
<b>Data Type</b>	Stores elements of the <b>same</b> data type	Stores elements of <b>different</b> data types
<b>Memory Allocation</b>	Contiguous memory allocation	May or may not be contiguous
<b>Access Method</b>	Accessed using <b>index</b> <code>( array[i] )</code>	Accessed using <b>dot operator</b> <code>( struct.member )</code>
<b>Size</b>	Fixed size, defined at declaration	Size depends on individual member types and alignment
<b>Pointer Concept</b>	Acts as a pointer to the first element	Does not act as a pointer
<b>Performance</b>	Faster for searching and traversal	Slower due to varied data types
<b>Declaration</b>	<code>data_type array_name[size];</code>	<code>struct struct_name { data_type memb</code>
<b>Usage</b>	Best for storing multiple values of the same type	Best for grouping related data of different types

# C Programming Unions

# What is Union ?

- In C, union is a user-defined data type that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location.
- Due to this, only one member can store data at the given point in time.

# Union Declaration

- A union is declared similarly to a structure. Provide the name of the union and define its member variables:
- `union union_name{`
- `type1 member1;`
- `type2 member2;`
- `type3 member3;`
- `.`
- `.`
- `.`
- `}`

# Variable Creation type

- 1. Along with Union Declaration.
- `union_name variable_name;`
- 2. in function using ‘union’ keyword.
- `union union_name{`
- `type1 member1;`
- `type2 member2;`
- `type3 member3;`
- `. . .`
- `. . .`
- `} variable_name;`

# Initialization and Accessing

- The value of a union variable can be accessed using the dot (.) operator. A value can be assigned to the union variable using the assignment operator (=).
- In a union, all the variables share the same memory, so only one variable can store a value at a time. If we try to access the value of another variable, the behavior will be undefined.

# Example :

```
1 #include <stdio.h>
2
3 // Define a union with
4 // different data types
5 union Student {
6     int rollNo;
7     float height;
8     char firstLetter;
9 };
10
11 int main() {
12
13     // Declare a union variable
14     union Student data;
15
16     // Assign and print the roll number
17     data.rollNo = 21;
18     printf("%d\n", data.rollNo);
19     data.height = 5.2;
20     printf("%.2f\n", data.height);
21     data.firstLetter = 'N';
22     printf("%c", data.firstLetter);
23
24     return 0;
25 }
```

## Output

```
21
5.20
N
```

# Size of Union

- Size of Union
- The size of the union will always be equal to the size of the largest member of the union. All the less-sized elements can store the data in the same space without any overflow. Let's take a look at the code example:

```
#include <stdio.h>

union A{
    int x;
    char y;
};

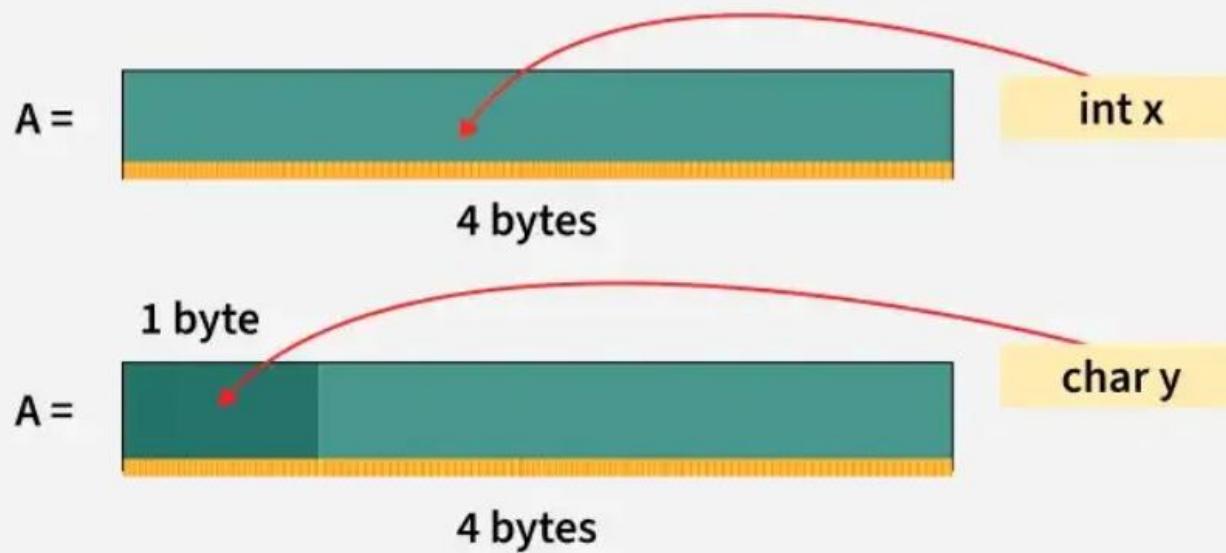
union B{
    int arr[10];
    char y;
};

int main() {
    // Finding size using sizeof() operator
    printf("Sizeof A: %ld\n", sizeof(union A));
    printf("Sizeof B: %ld\n", sizeof(union B));
    return 0;
}
```

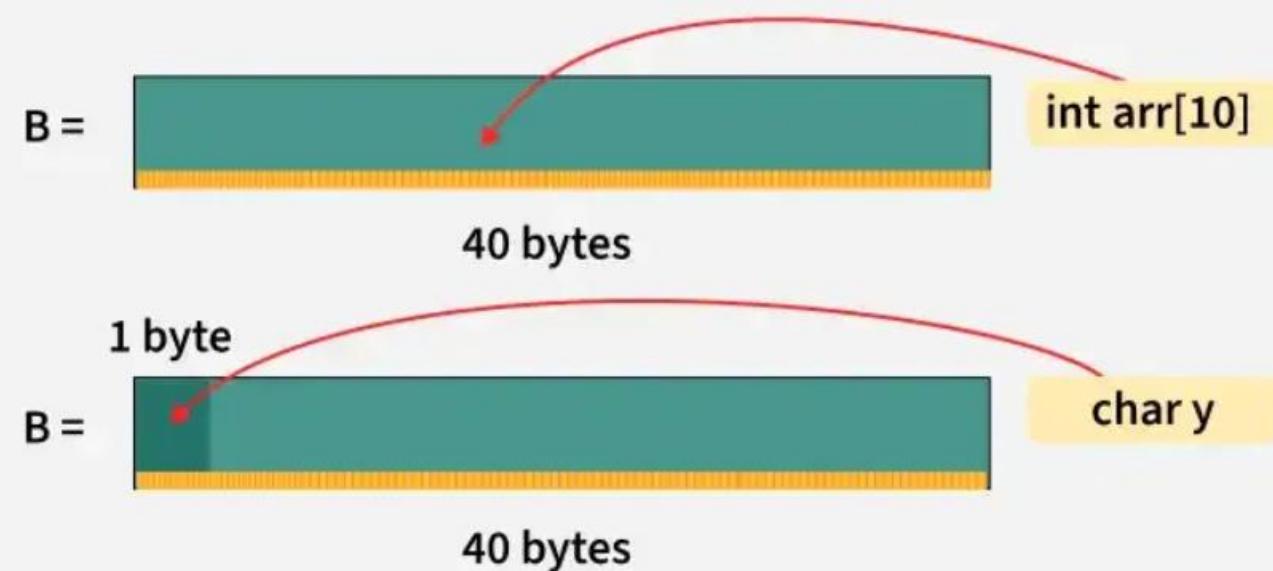
### Output

```
Sizeof A: 4
Sizeof B: 40
```

## Union Memory Representation



## Union Memory Representation



# Nested Union - Concept

- In C, we can define a union inside another union like structure. This is called nested union and is commonly used when you want to efficiently organize and access related data while sharing memory among its members.
- Syntax:
- `union name1{`
- `// Data members`
- `union name2{`
- `// Data members`
- `};`
- `};`
- Accessing members of union using dot(.) operator:
- `outer.inner.innerMember`

```
1 #include <stdio.h>
2
3 // Define a union with
4 // different data types
5 union Student {
6     int rollNo;
7     union Academic{
8         int marks;
9     } performance;
10 };
11
12 int main() {
13
14     // Declare a union variable
15     union Student abc;
16
17     // Assign and print the
18     // roll number
19     abc.rollNo = 21;
20     printf("%d\n", abc.rollNo);
21
22     // Assign and print the
23     // member of inner union
24     abc.performance.marks = 91;
25     printf("%d", abc.performance.marks);
26     return 0;
27 }
```

# Example

## Output

21

91

# Anonymous Union

- An anonymous union in C is a union that does not have a name. Instead of accessing its members through a named union variable, you can directly access the members of the anonymous union.
- This is useful when you want to access the union members directly within a specific scope, without needing to declare a union variable.

# Example :

## Output

```
21  
91
```

```
#include <stdio.h>

// Define a union with
// different data types
struct Student {
    int rollNo;

        // Anonymous union
    union {
        int marks;
    } performance;
};

int main() {

    // Declare a structure variable
    struct Student abc;

    abc.rollNo = 21;
    printf("%d\n", abc.rollNo);

    // Assign and print the member of anonymous union
    abc.performance.marks = 91;
    printf("%d", abc.performance.marks);

    return 0;
}
```

# Structure vs Unions

Parameter	Structure	Union
<b>Definition</b>	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.
<b>Keyword</b>	The keyword <b>struct</b> is used to define a structure	The keyword <b>union</b> is used to define a union
<b>Size</b>	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
<b>Memory Allocation</b>	Each member within a structure is allocated unique storage area of location.	Memory allocated is shared by individual members of union.
<b>Data Overlap</b>	No data overlap as members are independent.	Full data overlap as members shares the same memory.
<b>Accessing Members</b>	Individual member can be accessed at a time.	Only one member can be accessed at a time.

# Similarities between Structures and Unions

## Similarities Between Structure and Union

Structures and unions are also similar in some aspects listed below:

- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
- ‘.’ operator or selection operator, which has one of the highest precedences, is used for accessing member variables inside both the user-defined datatypes.

# Additional : What is C Language HTML Parser?

- A C Language HTML Parser is a program that processes HTML documents to extract useful information while ignoring HTML tags. It helps in parsing, analyzing, and modifying HTML content using C.
- **Key Features of an HTML Parser in C**
- - Extracts Text – Removes HTML tags and retrieves meaningful content.
- - Traverses HTML Structure – Allows navigation through elements like <div>, <p>, etc.
- - Handles Special Characters – Parses entities like &lt;, &amp;, etc.
- - Supports DOM Manipulation – Some parsers allow modifying HTML elements.

# C Programming

Static Variables and functions

# Static Variables in C Programming

- Static variables in C retain their value across multiple function calls and exist throughout the program's execution. They are declared using the **static keyword**.
- **Key Features:**
- **Preserve Value:** Unlike normal local variables, static variables do not lose their value when a function exits.
- **Local Scope:** If declared inside a function, they remain restricted to that function.
- **Global Static Variables:** If declared outside functions, they are accessible only within the same file.

# Example :

C Copy

```
#include <stdio.h>

void counter() {
    static int count = 0; // Initialized only once
    count++;
    printf("Count: %d\n", count);
}

int main() {
    counter(); // Output: Count: 1
    counter(); // Output: Count: 2
    return 0;
}
```

# Static function in c

- Static functions are restricted to the file in which they are declared. They cannot be accessed from other files, making them useful for encapsulation.
- Key Features:
- File Scope: Static functions are private to the file they are declared in.
- Encapsulation: Prevents accidental access from other files. (OOP's Concept which we will understand in the C++)
- Optimized Performance: The compiler may optimize static functions better.

# Example :

C

 Copy

```
#include <stdio.h>

static void greet() { // Static function
    printf("Hello, World!\n");
}

int main() {
    greet(); // Output: Hello, World!
    return 0;
}
```

Here, greet() cannot be accessed from another file.

# Additional : Interesting Facts

- Following are some interesting facts about static variables:
- Static variables (like global variables) are initialized as 0 if not initialized explicitly. See this for more details.
- In C, static variables can only be initialized using constant literals. See this for more details.

# Additional : Local vs Static Variables

Local Variable	Static Variable
Local to the function or block	Local to the function or block
Exists only during function execution	Exists throughout the program execution
Reinitialized each time function is called	Initialized only once
Stored in the stack	Stored in the data segment

# C Programming

## Pointers

# What is Pointer?

Overview : Pointer is variable that stores the address(memory location address) of any other variable rather than having it's own value , it just store a address value that pointing to a value of another value!

Pointers can be used to indirectly reference data, often in a more compact and efficient way than a direct reference. They are commonly used in C programming for

the processing of arrays, and are often essential for passing information out of a procedure. It can be difficult to avoid the use of pointers for some programming applications.

# Definition of Pointer

- A pointer is a variable that stores the memory address of another variable.
- Instead of holding a direct value, it has the address where the value is stored in memory.
- This allows us to manipulate the data stored at a specific memory location without actually using its variable.
- It is the backbone of low-level memory manipulation in C.

# Pointer Declarations

- A pointer is declared by specifying its name and type, just like simple variable declaration but with an asterisk (\*) symbol added before the pointer's name.
- Syntax : data\_type\* name;
- Example : int \*ptr;

# Intialize the pointer

- Pointer initialization means assigning some address to the pointer variable. In C, the (&) addressof operator is used to get the memory address of any variable.
- This memory address is then stored in a pointer variable.
- `int var = 10;`
- `// Initializing ptr`
- `int *ptr = &var;`
- Note: We can also declare and initialize the pointer in a single step. This is called pointer definition.

# Example : Basic Pointer

- #include <stdio.h>
- int main() {
- int var = 10;
- // Store address of var variable
- int\* ptr = &var;
- // Directly accessing ptr
- printf("%d", ptr);
- return 0;
- }
- It will print the address of the variables.

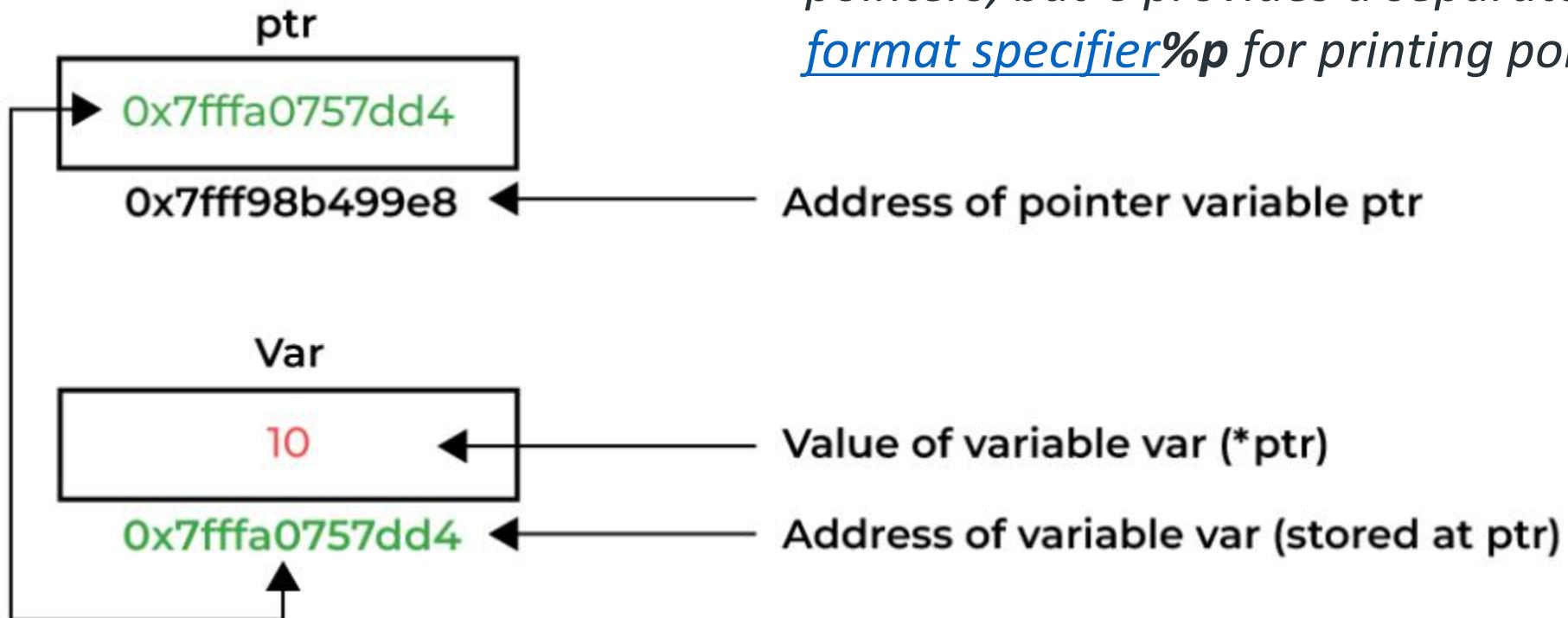
# dereferencing operator(\*)

- #include <stdio.h>
- int main() {
- int var = 10;
- // Store address of var variable
- int\* ptr = &var;
- // Dereferencing ptr to access the value
- printf("%d", \*ptr);
- }

Output

10

# Understanding by Diagram!



**Note:** Earlier, we used `%d` for printing pointers, but C provides a separate format specifier `%p` for printing pointers.

# Size of Pointers

- The size of a pointer in C depends on the architecture (bit system) of the machine, not the data type it points to.
- On a 32-bit system, all pointers typically occupy 4 bytes.
- On a 64-bit system, all pointers typically occupy 8 bytes.

# Example : for 64 bit machine

- #include <stdio.h>
- int main() {
- int \*ptr1;
- char \*ptr2;
- 
- // Finding size using sizeof()
- printf("%zu\n", sizeof(ptr1));
- printf("%zu", sizeof(ptr2));
- 
- return 0;
- }

*The actual size of the pointer may vary depending on the **compiler and system architecture**, but it is always **uniform across all data types** on the same system.*

Output

8

8

# Special Type of Pointers

- 1. NULL POINTER
- The Null Pointer is the pointer that does not point to any location but NULL. According to C11 standard:
- “An integer constant expression with the value 0, or such an expression cast to type `void *`, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.”

# Syntax of Null Pointer Declaration in C

- `type pointer_name = NULL;`
- `type pointer_name = 0;`
- We just have to assign the `NULL` value. Strictly speaking, `NULL` expands to an implementation-defined null pointer constant which is defined in many header files such as “`stdio.h`”, “`stddef.h`”, “`stdlib.h`” etc.

# Use of NULL Pointer in C

- To initialize a pointer variable when that pointer variable hasn't been assigned any valid memory address yet.
- To check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer-related code, e.g., dereference a pointer variable only if it's not NULL.
- To pass a null pointer to a function argument when we don't want to pass any valid memory address.
- A NULL pointer is used in data structures like trees, linked lists, etc. to indicate the end.

# Checking the Pointer Value

- It is a valid operation in pointer arithmetic to check whether the pointer is NULL. We just have to use isequal to operator ( == ) as shown below:
- `ptr == NULL;`

# Example : NULL POINTER

```
// C NULL pointer demonstration
#include <stdio.h>

int main()
{
    // declaring null pointer
    int* ptr = NULL;

    // derefencing only if the pointer have any value
    if (ptr == NULL) {
        printf("Pointer does not point to anything");
    }
    else {
        printf("Value pointed by pointer: %d", *ptr);
    }
    return 0;
}
```

## Output

Pointer does not point to anything

## 2. Void Pointer

- A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.
- Time Complexity:  $O(1)$
- Auxiliary Space:  $O(1)$

# Example :

```
// C Program to demonstrate that a void pointer  
// can hold the address of any type-castable type
```

```
#include <stdio.h>  
int main()  
{  
    int a = 10;  
    char b = 'x';  
  
    // void pointer holds address of int 'a'  
    void* p = &a;  
    // void pointer holds address of char 'b'  
    p = &b;  
}
```

The  
[C standard](#) doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of the void as 1.

# Advantages of Void Pointers

- malloc() and calloc() return void \* type and this allows these functions to be used to allocate memory of any data type (just because of void \*).
- void pointers in C are used to implement generic functions in C. For example, compare function which is used in qsort().
- void pointers used along with Function pointers of type void (\*)(void) point to the functions that take any arguments and return any value.
- void pointers are mainly used in the implementation of data structures such as linked lists, trees, and queues i.e. dynamic data structures.
- void pointers are also commonly used for typecasting.

# 3. Wild Pointers

- The wild pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values are updated using wild pointers, they could cause data abort or data corruption.
- #include <stdio.h>
- int main() {
- // Wild Pointer
- int \*ptr;
- 
- return 0;
- }

# How can we avoid the wild pointers ?

- If a pointer points to a known variable then it's not a wild pointer.
- int main()
- {
- int\* p; /\* wild pointer \*/
- int a = 10;
- /\* p is not a wild pointer now \*/
- p = &a;
- /\* This is fine. Value of a is changed \*/
- \*p = 12;
- }

## 4. Dangling Pointers

- A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

# Example : Dangling Pointer

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

int main() {
    int* ptr = (int*)malloc(sizeof(int));

    // After below free call, ptr becomes a dangling pointer
    free(ptr);
    printf("Memory freed\n");

    // removing Dangling Pointer
    ptr = NULL;

    return 0;
}
```

There are three different ways where a pointer acts as a dangling pointer:

- 1. De-allocation of memory
- 2. function call
- 3. variable goes out of scope

# 1. De-allocation of memory

```
// C program to demonstrate Deallocating a memory pointed by
// ptr causes dangling pointer
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* ptr = (int*)malloc(sizeof(int));

    // After below free call, ptr becomes a dangling pointer
    free(ptr);
    printf("Memory freed\n");

    // removing Dangling Pointer
    ptr = NULL;

    return 0;
}
```

## 2. function call

- What happens?
- A dangling pointer occurs when a function returns a pointer to a local variable, and that variable gets destroyed once the function finishes execution. Since local variables exist only within the function's scope, they get deallocated from memory as soon as the function returns. So, if you try to access that memory using the returned pointer, it's like trying to use something that no longer exists—leading to undefined behavior.

## Example in C++

Cpp

Copy

```
#include <iostream>

int* getPointer() {
    int localVar = 10; // Local variable inside function
    return &localVar; // Returning address of local variable
}

int main() {
    int* ptr = getPointer(); // ptr now holds address of localVar
    std::cout << *ptr << std::endl; // Trying to dereference ptr

    return 0;
}
```

Why is this dangerous?

- The variable `localVar` only exists inside `getPointer()`. Once the function ends, `localVar` is removed from memory.
- `ptr` holds an invalid memory address (the memory location of a variable that no longer exists).
- Dereferencing `ptr` (`*ptr`) leads to undefined behavior—it might print garbage, crash the program, or cause unexpected results.
- Solution: Use dynamic memory allocation (will be covered)!

### 3. variable goes out of scope

- When a variable goes out of scope the pointer pointing to that variable becomes a dangling pointer.
- Let's Explore the example.

# Example :

```
// C program to demonstrate dangling pointer when variable
// goes out of scope
#include <stdio.h>
#include <stdlib.h>

// driver code
int main()
{
    int* ptr;
    // creating a block
    {
        int a = 10;
        ptr = &a;
    }

    // ptr here becomes dangling pointer
    printf("%d", *ptr);

    return 0;
}
```

# C Pointer Arithmetic

- The pointer arithmetic refers to the arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:
- Increment/Decrement
- Addition/Subtraction of Integer
- Subtracting Two Pointers of Same Type
- Comparing/Assigning Two Pointers of Same Type
- Comparing/Assigning with NULL

# 1. Increment / Decrement of a Pointer

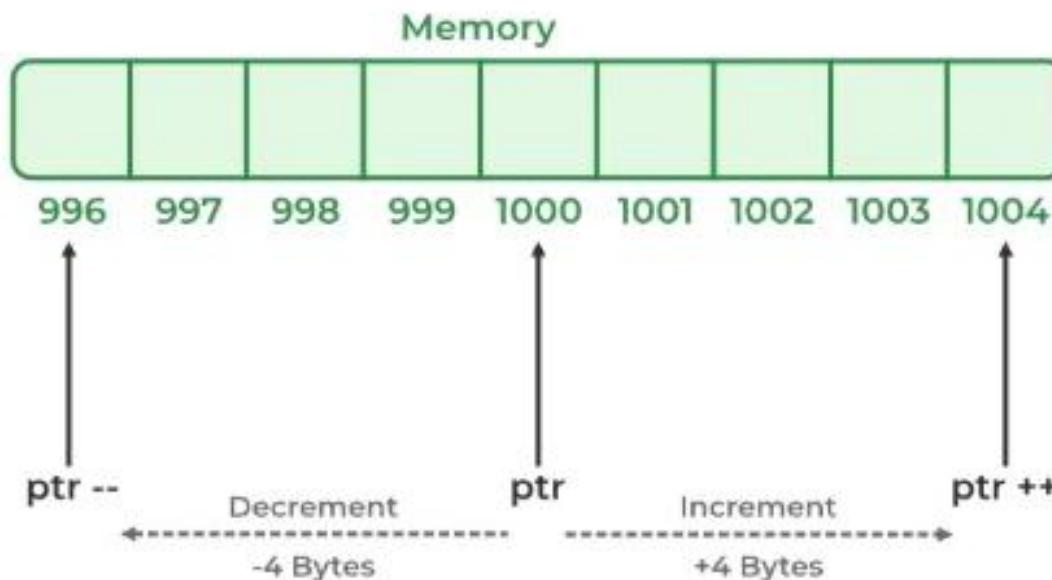
- **Increament :** It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.
- For Example:
- If an integer pointer that stores address 1000 is incremented, then it will increment by 4(size of an int), and the new address will point to 1004. While if a float type pointer is incremented then it will increment by 4(size of a float) and the new address will be 1004.

# 1. Increment / Decrement of a Pointer

- **Decrement :** It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.
- For Example:
- If an integer pointer that stores address 1000 is decremented, then it will decrement by 4(size of an int), and the new address will point to 996. While if a float type pointer is decremented then it will decrement by 4(size of a float) and the new address will be 996.

# Understanding by Diagram

## Pointer Increment & Decrement



**Note:**

*It is assumed here that the architecture is 64-bit and all the data types are sized accordingly. For example, integer is of 4 bytes.*

# Exempel : Int

- int a = 22;
- int \*p = &a;
- printf("p = %u\n", p); // p = 6422288
- p++;
- printf("p++ = %u\n", p); //p++ = 6422292 +4 // 4 bytes
- p--;
- printf("p-- = %u\n", p); //p-- = 6422288 -4 // restored to original value

# Example : float

- float b = 22.22;
- float \*q = &b;
- printf("q = %u\n", q); //q = 6422284
- q++;
- printf("q++ = %u\n", q); //q++ = 6422288      +4 // 4 bytes
- q--;
- printf("q-- = %u\n", q); //q-- = 6422284      -4 // restored to original value

# Exampel : char

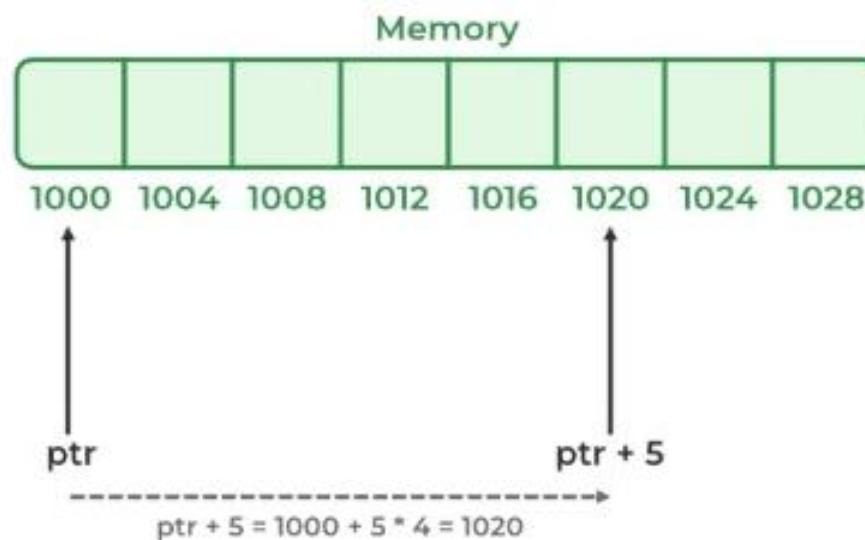
- `char c = 'a';`
- `char *r = &c;`
- `printf("r = %u\n", r); //r = 6422283`
- `r++;`
- `printf("r++ = %u\n", r); //r++ = 6422284 +1 // 1 byte`
- `r--;`
- `printf("r-- = %u\n", r); //r-- = 6422283 -1 // restored to original value`

## 2. Addition of Integer Pointer!

- When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.
- For Example:
- Consider the same example as above where the ptr is an integer pointer that stores 1000 as an address. If we add integer 5 to it using the expression,  $\text{ptr} = \text{ptr} + 5$ , then, the final address stored in the ptr will be  $\text{ptr} = 1000 + \text{sizeof(int)} * 5 = 1020$ .

# Diagram Representation : Addition of Pointers

## Pointer Addition



## Example of Addition of Integer to Pointer

```
// C program to illustrate pointer Addition
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Addition: ");
    printf("%p \n", ptr2);

    // Addition of 3 to ptr2
    ptr2 = ptr2 + 3;
    printf("Pointer ptr2 after Addition: ");
    printf("%p \n", ptr2);

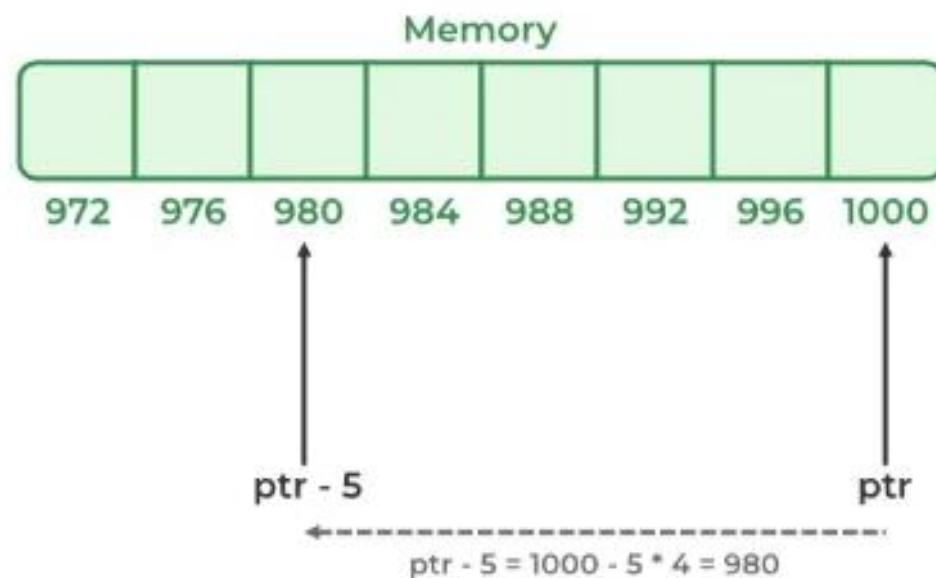
    return 0;
}
```

### 3. Subtraction of Integer to Pointer

- When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.
- For Example:
- Consider the same example as above where the ptr is an integer pointer that stores 1000 as an address. If we subtract integer 5 from it using the expression,  $\text{ptr} = \text{ptr} - 5$ , then, the final address stored in the ptr will be  $\text{ptr} = 1000 - \text{sizeof(int)} * 5 = 980$ .

# Understanding by Diagram

## Pointer Subtraction



```
// C program to illustrate pointer Subtraction
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Subtraction: ");
    printf("%p \n", ptr2);

    // Subtraction of 3 to ptr2
    ptr2 = ptr2 - 3;
    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);

    return 0;
}
```

# Example

## 4. Subtraction of Two Pointers

- The subtraction of two pointers is possible only when they have the same data type.
- The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bytes of data it is according to the pointer data type.
- The subtraction of two pointers gives the increments between the two pointers.

## Example :

- Two integer pointers say ptr1(address:1000) and ptr2(address:1004) are subtracted.
- The difference between addresses is 4 bytes.
- Since the size of int is 4 bytes, therefore the increment between ptr1 and ptr2 is given by  $(4/4) = 1$ .

- int x = 6; // Integer variable declaration
- int N = 4;
- // Pointer declaration
- int \*ptr1, \*ptr2;
- ptr1 = &N; // stores address of N
- ptr2 = &x; // stores address of x
- printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
  - // %p gives an hexa-decimal value,
  - // We convert it into an unsigned int value by using %u
- // Subtraction of ptr2 and ptr1
- x = ptr1 - ptr2;
- // Print x to get the Increment
- // between ptr1 and ptr2
- printf("Subtraction of ptr1 "
  - "& ptr2 is %d\n",
  - x);

# 5. Comparision between Pointers

- We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C >, >=, <, <=, ==, !=. It returns true for the valid condition and returns false for the unsatisfied condition.
- Step 1: Initialize the integer values and point these integer values to the pointer.
- Step 2: Now, check the condition by using comparison or relational operators on pointer variables.
- Step 3: Display the output.

## Example of Pointer Comparision

```
// C Program to illustrate pointer comparision
#include <stdio.h>

int main()
{
    // declaring array
    int arr[5];

    // declaring pointer to array name
    int* ptr1 = &arr;
    // declaring pointer to first element
    int* ptr2 = &arr[0];

    if (ptr1 == ptr2) {
        printf("Pointer to Array Name and First Element "
               "are Equal.");
    }
    else {
        printf("Pointer to Array Name and First Element "
               "are not Equal.");
    }

    return 0;
}
```

### Output

Pointer to Array Name and First Element are Equal.

- Comparision to NULL

```
// C Program to demonstrate the pointer comparison with NULL
// value
#include <stdio.h>

int main()
{
    int* ptr = NULL;

    if (ptr == NULL) {
        printf("The pointer is NULL");
    }
    else {
        printf("The pointer is not NULL");
    }
    return 0;
}
```

Output

The pointer is NULL

# Pointer with array using comparsion

```
// Pointer Comparision in Array
#include <stdio.h>

int main()
{
    int n = 10; // length of an array

    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int* ptr; // Declaration of pointer variable

    ptr = arr; // Pointer points the first (0th index)
                // element in an array
    int count_even = 0;
    int count_odd = 0;

    for (int i = 0; i < n; i++) {

        if (*ptr % 2 == 0) {
            count_even++;
        }
        if (*ptr % 2 != 0) {
            count_odd++;
        }
        ptr++; // Pointing to the next element in an array
    }
    printf("No of even elements in an array is : %d",
           count_even);
    printf("\nNo of odd elements in an array is : %d",
           count_odd);
}
```



# Concept of Constant Pointer

- In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined.
- It will always point to the same memory address.

```
#include <stdio.h>

int main() {
    int a = 90;
    int b = 50;

    // Creating a constant pointer
    int* const ptr = &a;

    // Trying to reassign it to b
    ptr = &b;

    return 0;
}
```

## Output

```
solution.c: In function ‘main’:
solution.c:11:9: error: assignment of read-only variable ‘ptr’
  11 |     ptr = &b;
                  ^
```

# Difference between constant pointer, pointers to constant, and constant pointers to constants

- 1. Pointer to Constant
- In the pointers to constant, the data pointed by the pointer is constant and cannot be changed.
- Although, the pointer itself can change and points somewhere else (as the pointer itself is a variable).
- Let's Explore the example.

# Example :

- #include <stdio.h>
- int main() {
- const int num = 10; // Constant variable
- const int \*ptr = &num; // Pointer to a constant
- printf("Value: %d\n", \*ptr);
- // \*ptr = 20; // ✗ Error: Cannot modify a constant value
- int anotherNum = 30;
- ptr = &anotherNum; // ✓ Allowed: Pointer can change its target
- printf("New Value: %d\n", \*ptr);
- return 0;
- }

# Explanation from Previous Example.

- Key Points
- - const int \*ptr → You cannot modify the value at ptr, but the pointer itself can be reassigned.
- - Trying \*ptr = 20; → Compilation error, because ptr points to a const int.
- - ptr = &anotherNum; → Allowed, since the pointer itself isn't constant.

## 2. Constant Pointer

- In constant pointers, the pointer points to a fixed memory location, and the value at that location can be changed because it is a variable, but the pointer will always point to the same location because it is made constant here.



```
int *const ptr = &a; // ptr points to a
*ptr = 56; // ok we can change value of a which is *ptr by dereferencing
ptr = &b; // Error because the value of ptr cannot be changed as it is a constant
```

# Example :

```
int main() {
    int num = 10;
    int anotherNum = 30;

    int *const ptr = &num; // Constant pointer to int

    printf("Value: %d\n", *ptr);

    *ptr = 20; // ✓ Allowed: You can modify the value
    printf("Modified Value: %d\n", *ptr);

    // ptr = &anotherNum; // ✗ Error: Cannot change pointer's address

    return 0;
}
```

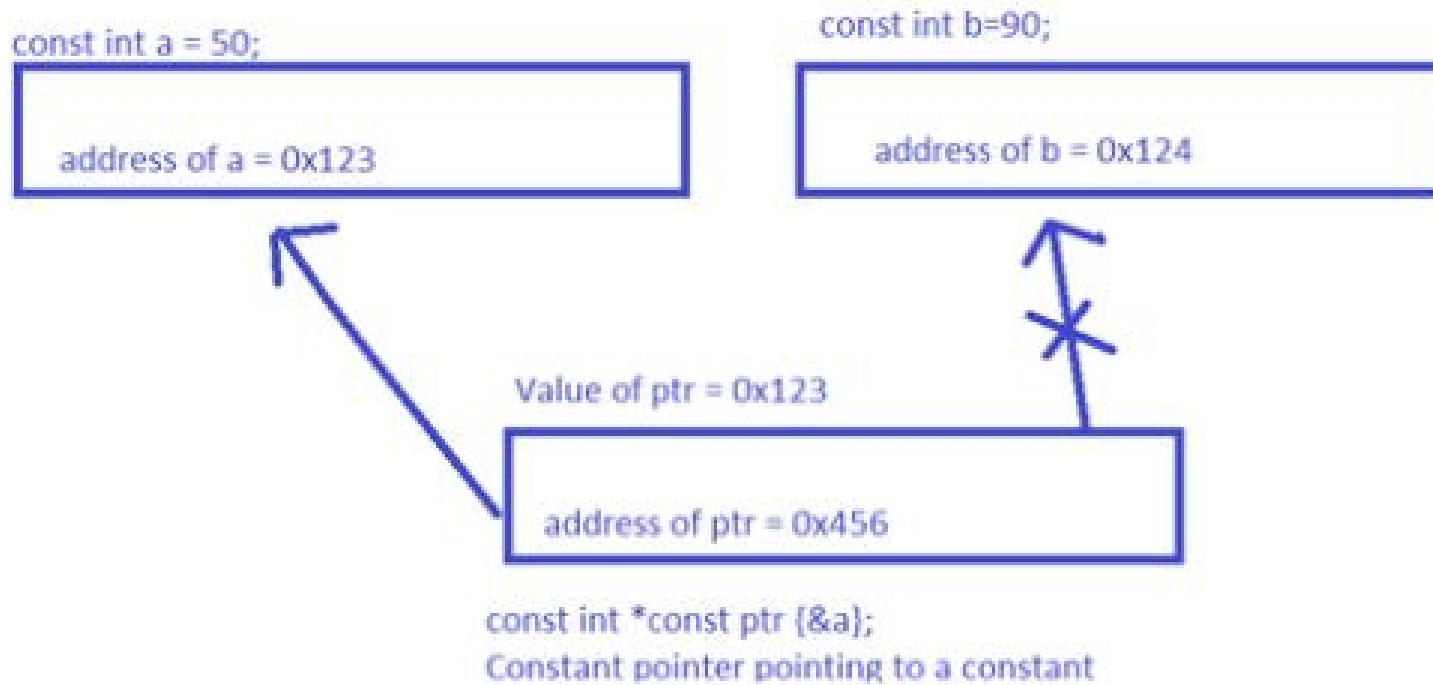
# Key Points from Previous Example :

- - int \*const ptr = &num; → The pointer itself is constant.
- - \*ptr = 20; → Allowed, since the value at ptr can be modified.
- - ptr = &anotherNum; → Compilation error, because ptr is constant and cannot point elsewhere.

### 3. Constant Pointers to constants:

- In the constant pointers to constants, the data pointed to by the pointer is constant and cannot be changed.
- The pointer itself is constant and cannot change and point somewhere else. Below is the image to illustrate the same:

# Diagram



ptr points to a (const int \*const\_ptr{&a})  
`*ptr = 90; // Error`  
`ptr = & b; //Error`

# Example :

```
#include <stdio.h>

int main() {
    const int num = 10;
    const int anotherNum = 30;

    const int *const ptr = &num; // Constant pointer to a constant

    printf("Value: %d\n", *ptr);

    // *ptr = 20; // ✗ Error: Cannot modify a constant value
    // ptr = &anotherNum; // ✗ Error: Cannot change pointer's address

    return 0;
}
```

# Key points from Previous Example :

- - const int \*const ptr = &num; → Both the pointer and the value are constant.
- - \*ptr = 20; → Compilation error, because num is constant.
- - ptr = &anotherNum; → Compilation error, because ptr is constant and cannot point elsewhere.

# Topic : Pointers and Array

- Two Sub-topic will be covered in this Slides :
  - 1. Relationship between Pointers and Array.
  - 2. Pointer to Array

# 1. Relationship between Array and Pointer in C

- In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. If we assign this value to a non-constant pointer of the same type, then we can access the elements of the array using this pointer.
- For example, if we have an array named arr then arr and &arr[0] can be used interchangeably.

# Example :

```
#include <stdio.h>

int main() {
    // Declare an array
    int arr[3] = { 5, 10, 15 };

    // Access first element using index
    printf("%d\n", arr[0]);

    // Access first element using pointer
    printf("%d\n", *arr);

    return 0;
}
```

Not only that, as the array elements are stored continuously, we can perform arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

## Output

```
5
5
```

# Traversing Exampel :

```
#include<stdio.h>
int main()
{
    int *ptr;
    int a[5] = {10,20,30,40,50};

    ptr = &a[0];
    for(int i=0;i<5;i++)
    {
        printf("\n Array Element [ i ] : %d",ptr[i]);
    }
}
```

# • Passing Array to the function

```
#include <stdio.h>

void f1(int arr[3]) {
    printf("Size in f1: %lu bytes\n", sizeof(arr));
}

void f2(int arr[]) {
    printf("Size in f2: %lu bytes\n", sizeof(arr));
}

void f3(int *arr) {
    printf("Size in f3: %lu bytes\n", sizeof(arr));
}

int main() {
    int arr[3] = { 1, 2, 3 };

    printf("Size in main(): %lu bytes\n", sizeof(arr));

    f1(arr);
    f2(arr);
    f3(arr);

    return 0;
}
```

## Output

```
Size in main(): 12 bytes
Size in f1: 8 bytes
Size in f2: 8 bytes
Size in f3: 8 bytes
```

# Pointers and 2-D Array

- In a two-dimensional array, we can access each element by using two subscripts, where the first subscript represents the row number, and the second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element  $\text{arr}[i][j]$  of the array using the pointer expression:
- $*(*(\text{arr} + i) + j)$ .

```
#include <stdio.h>

int main() {
    int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8},
                     {9, 10, 11, 12} };

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++)

            // using pointer notation for 2d array
            printf("%d ", *(*(arr + i) + j));
        printf("\n");
    }

    return 0;
}
```

### Output

```
1 2 3 4
5 6 7 8
9 10 11 12
```

# Pointers and 3-D Array

Syntax	Description
arr	Points to 0 <sup>th</sup> 2D Array.
arr + i	Points to i <sup>th</sup> 2D Array.
*(arr + i)	Gives base address of i <sup>th</sup> 2D array, so point to 0 <sup>th</sup> element of i <sup>th</sup> 2D array, so it points to 0 <sup>th</sup> 1D array of i <sup>th</sup> 2D array.
*(arr + i) + j	Points to the j <sup>th</sup> 1D array of i <sup>th</sup> 2D array.
*(*(arr + i) + j)	Gives base address of j <sup>th</sup> 1D array of i <sup>th</sup> 2D array.
*(*(arr + i) + j) + k	Represent the value of j <sup>th</sup> element of the i <sup>th</sup> 1D array.
*(*(arr + i) + j) + k	Gives the value of the k <sup>th</sup> element of j <sup>th</sup> 1D array i <sup>th</sup> 2D array.

```
#include <stdio.h>

int main() {
    int arr[2][3][2] = {{{5, 10}, {6, 11}, {7, 12}},
                        {{20, 30}, {21, 31}, {22, 32}}};

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 2; k++)

                // Accessing using pointer notation
                printf("%d ", *(*(*(arr + i) + j) + k));
                printf("\n");
    }
    printf("\n");
}

return 0;
}
```

### Output

```
5 10
6 11
7 12

20 30
21 31
22 32
```

# How to Pass a 3-D Array to a function ?

- We cannot directly pass a 3D array to a function just like we do one-dimensional arrays. Instead, we must pass the 3D array to function as a pointer. When doing so, the array undergoes array decay, losing information about its dimensions. Therefore, we must pass the dimensions of the array separately
- Let's Explore the Syntax :

## Syntax : Refer Example as well

- functionType funcName(type (\*arr)[cols][depth], int rows, int cols, int depth)
- Here,
- funcName: It is the name of the function.
- arr: It is the pointer which points to the 3D array.
- rows: It represents the number of 2D arrays.
- cols: It represents the number of rows in each 2D array.
- depth: It represents the number of columns in each 2D array.

## 2. Pointer to Array : Detailed

- A pointer to an array is a pointer that points to the whole array instead of the first element of the array. It considers the whole array as a single unit instead of it being a collection of given elements.

```
#include<stdio.h>
int main()
{
    int array[3] = {10,15,20};
    int (*a)[3] = &array;
    for(int i=0;i<3;i++)
    {
        printf("\n Element : %d",(*a)[i]);
    }
}
```

# Pointer to function / function Pointer

- In C, a function pointer is a type of pointer that stores the address of a function, allowing functions to be passed as arguments and invoked dynamically.
- It is useful in techniques such as callback functions, event-driven programs, and polymorphism (a concept where a function or operator behaves differently based on the context,Part of OPP)

```
#include <stdio.h>

Example : int add(int a, int b) {
    return a + b;
}

int main() {
    // Declare a function pointer that matches
    // the signature of add() fuction
    int (*fptr)(int, int);

    // Assign to add()
    fptr = &add;

    // Call the function via ptr
    printf("%d", fptr(10, 5));
    return 0;
}
```

## Output

15

**Explanation:** In this program, we define a function **add()**, assigns its address to a function pointer **fptr**, and invokes the function through the pointer to print the sum of two integers.

# Let's Break and Understand : 1. Declaration

- Function pointers are declared according to the signature of the function they will be pointing to. Below is the generic syntax of function pointer declaration:

```
return_type (*pointer_name)(parameter_types);
```

- return\_type: The type of the value that the function returns.
- parameter\_types: The types of the parameters the function takes.
- pointer\_name: The name of the function pointer.

# Initialization

- Initialization
- A function pointer is then initialized by assigning the address of the function.
- `pointer_name = &function_name`
- We can also skip the address of operator as function name itself behaves like a constant function pointer.
  
- `pointer_name = function_name;`
- It is compulsory to assign the function with similar signature as specified in the pointer declaration. Otherwise, the compiler may show type mismatch error.

# Properties of Function Pointer

- Function pointer points to the code instead of the data so there are some restrictions on the function pointers as compared to other pointers. Following are some important properties of function pointer:
- Points to the memory address of a function in the code segment.
- Requires the exact function signature (return type and parameter list).
- Can point to different functions with matching signatures.
- Cannot perform arithmetic operations like increment or decrement.
- Supports array-like functionality for tables of function pointers.

# Double Pointer / Pointer to Pointer

- double pointers are those pointers which stores the address of another pointer. The first pointer is used to store the address of the variable, and the second pointer is used to store the address of the first pointer. That is why they are also known as a pointer to pointer.

# Example :

```
#include <stdio.h>

int main() {

    // A variable
    int var = 10;

    // Pointer to int
    int *ptr1 = &var;

    // Pointer to pointer (double pointer)
    int **ptr2 = &ptr1;

    printf("var: %d\n", var);
    printf("*ptr1: %d\n", *ptr1);
    printf("**ptr2: %d", **ptr2);

    return 0;
}
```

## Explanation:

In this code, **ptr1** is a pointer that stores the address of the integer variable **var**. **ptr2** is a double pointer that stores the address of the pointer **ptr1**. **\*\*ptr2** dereferences **ptr2** to get the value of **ptr1** (which is the address of **var**) and then dereferences that address to get the value of **var** itself.

## Output

```
var: 10
*ptr1: 10
**ptr2: 10
```

# Let's Breakdown.... STEP BY STEP!

- Declaration
- A double pointer can be declared similar to a single pointer. The difference is we have to place an additional '\*' before the name of the pointer.
- type \*\*name;

# Initialization

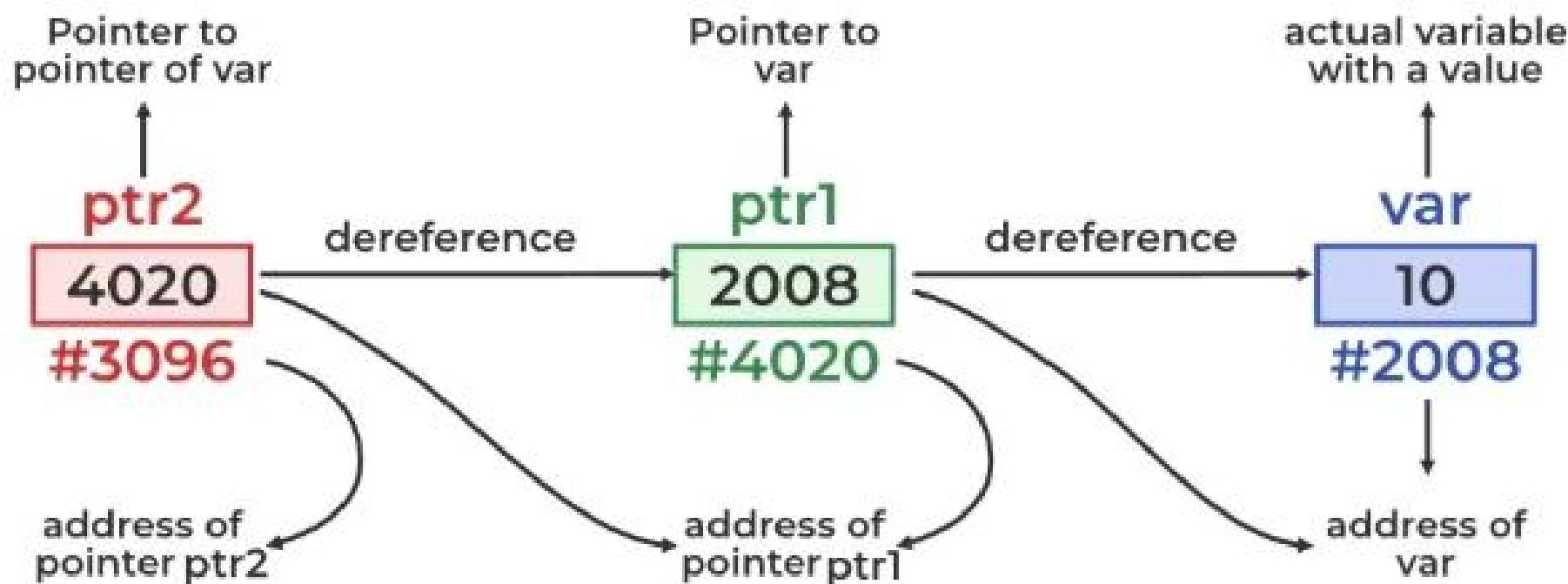
- The double pointer stores the address of another pointer to the same type.
- `name = &single_ptr;` // After declaration
- `type **name = &single_ptr;` // With declaration

# Deferencing

- `*name;` // Gives you the address of the single pointer
- `**name;` // Gives you the value of the variable it points to

# Understanding by Daigram

## Double Pointer



# Application of Double Pointers in C

- They are used in the dynamic memory allocation of multidimensional arrays.
- They can be used to store multilevel data such as the text document paragraph, sentences, and word semantics.
- They are used in data structures to directly manipulate the address of the nodes without copying.
- They can be used as function arguments to manipulate the address stored in the local pointer.
- [More](#)

# Multilevel Pointers / chain pointer

- In C, we can create multi-level pointers with any number of levels such as – \*\*\*ptr3, \*\*\*\*ptr4, \*\*\*\*\*ptr5 and so on.
- Most popular of them is double pointer (pointer to pointer). It stores the memory address of another pointer.
- Instead of pointing to a data value, they point to another pointer.

# Prerequisite: Pointers in C, Double Pointer (Pointer to Pointer) in C

- A pointer is used to point to a memory location of a variable. A pointer stores the address of a variable.
- Similarly, a chain of pointers is when there are multiple levels of pointers. Simplifying, a pointer points to address of a variable, double-pointer points to a variable and so on. This is called multiple indirections.

# Syntax - Declaration

Syntax:

```
// level-1 pointer declaration  
datatype *pointer;  
  
// level-2 pointer declaration  
datatype **pointer;  
  
// level-3 pointer declaration  
datatype ***pointer;  
. . .  
and so on
```

Declaration:

```
int *pointer_1;  
int **pointer_2;  
int ***pointer_3;
```

.

.

and so on

# Uses of Pointers in C

- Pass Arguments by Pointers
- Accessing Array Elements
- Return Multiple Values from Function
- Dynamic Memory Allocation
- Implementing Data Structures
- In System-Level Programming where memory addresses are useful.
- To use in Control Tables.

# Pass Argument using Pointers

```
// C program to swap two values
// without passing pointer to
// swap function.
#include <stdio.h>

void swap(int* a, int* b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

// Driver code
int main()
{
    int a = 10, b = 20;
    printf("Values before swap function are: %d, %d\n",
           a, b);
    swap(&a, &b);
    printf("Values after swap function are: %d, %d",
           a, b);
    return 0;
}
```

# Advantages of Pointers

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

# Issues with Pointers

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for memory leaks in C.
- Accessing using pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

# Concept : Dynamic Memory Allocation

- a variable defined in a function is stored in the stack memory. The requirement of this memory is that it needs to know the size of the data to memory at compile time (before the program runs). Also, once defined, we can neither change the size nor completely delete the memory.
- C provides a feature called Dynamic Memory Allocation. It allows you to allocate memory at runtime, giving your program the ability to handle data of varying sizes. Dynamic resources are stored in the heap memory instead of the stack.

# Need of DMA

- the size of an array in C is fixed and should be known at compile time.  
There can be two problems:
- The size of the array is not sufficient to store all the elements. To resolve this, one might set the size to store the maximum theoretically possible elements. This creates another problem.

# DMA

**Array Size: 5**

Array[0]   Array[1]   Array[2]   Array[3]   Array[4]

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

**Remaining: [6, 7, 8]**

# DMA

Array Size: 8

Array[0]	Array[1]	Array[2]	Array[3]	Array[4]	Array[5]	Array[6]	Array[7]
1	2	3	4	5	Unused	Unused	Unused

unused memory: 3 slots

# DMA

- This size of the array is much more than what is required to store the elements. This leads to the wastage of memory.
- This is where the dynamic memory allocation comes in. The size of the array can be increased if more elements are to be inserted and decreased if less elements are inserted. Moreover, there is no need to estimate the max possible size. The size can be decided at runtime according to the requirement.

# malloc()

- The malloc() (stands for memory allocation) function is used to allocate a single block of contiguous memory on the heap at runtime. The memory allocated by malloc() is uninitialized, meaning it contains garbage values.
- Syntax : malloc(size);
- where size is the number of bytes to allocate.
- This function returns a void pointer to the allocated memory that needs to be converted to the pointer of required type to be usable. If allocation fails, it returns NULL pointer.

# Example : malloc()

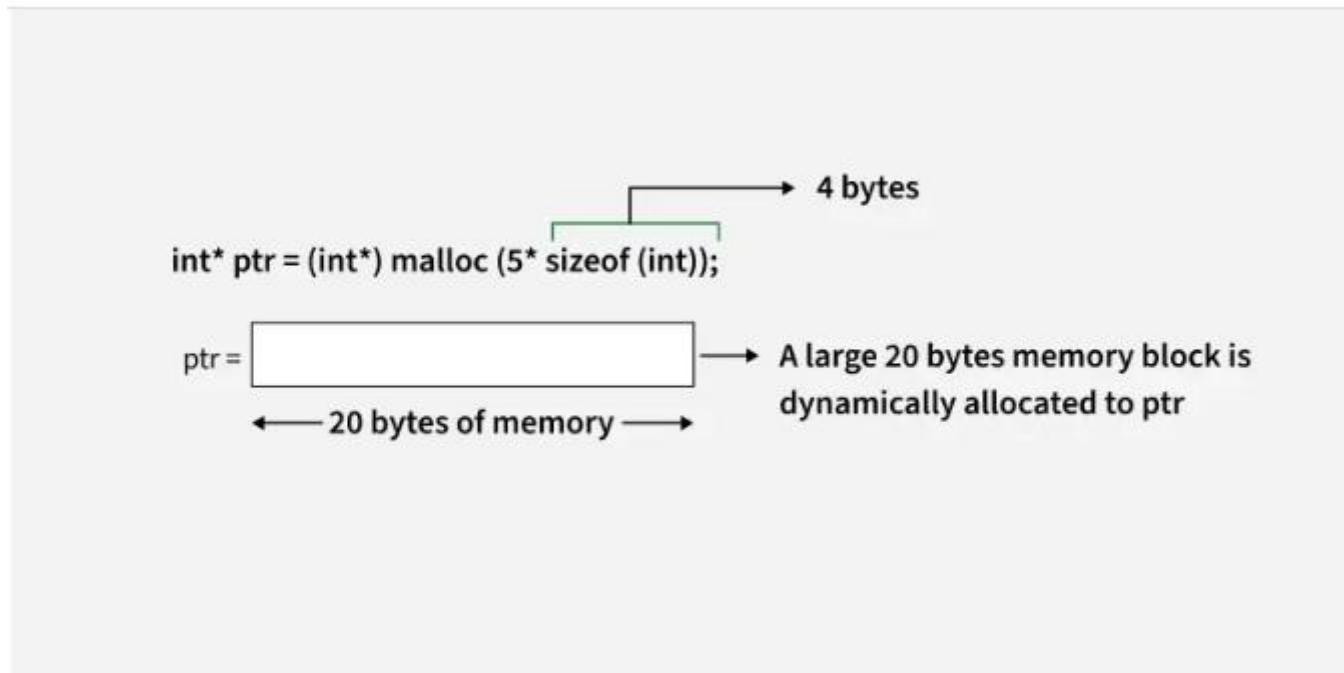
```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int *)malloc(20);

    // Populate the array
    for (int i = 0; i < 5; i++)
        ptr[i] = i + 1;

    // Print the array
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]);
    return 0;
}
```

# malloc() - Diagram



# calloc()

- The calloc() (stands for contiguous allocation) function is similar to malloc(), but it initializes the allocated memory to zero. It is used when you need memory with default zero values.
- Syntax
- `calloc(n, size);`
- where n is the number of elements and size is the size of each element in bytes.
- This function also returns a void pointer to the allocated memory that is converted to the pointer of required type to be usable. If allocation fails, it returns NULL pointer.

# Example : calloc()

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

int main() {
    int *ptr = (int *)calloc(5, sizeof(int));

    // Checking if failed or pass
    if (ptr == NULL) {
        printf("Allocation Failed");
        exit(0);
    }

    // No need to populate as already
    // initialized to 0

    // Print the array
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]);
    return 0;
}
```

# Diagram : calloc()

```
int* ptr = (int*) calloc (5, sizeof (int));
```

ptr =  → 5 blocks of 4 bytes each is  
dynamically allocated to ptr

← 20 bytes of memory →

# free()

- The memory allocated using functions malloc() and calloc() is not de-allocated on their own. The free() function is used to release dynamically allocated memory back to the operating system. It is essential to free memory that is no longer needed to avoid memory leaks.
- Syntax
- free(ptr);
- where ptr is the pointer to the allocated memory.
- After freeing a memory block, the pointer becomes invalid, and it is no longer pointing to a valid memory location.

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

int main() {
    int *ptr = (int *)calloc(sizeof(int), 5);

    // Do some operations.....
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]);

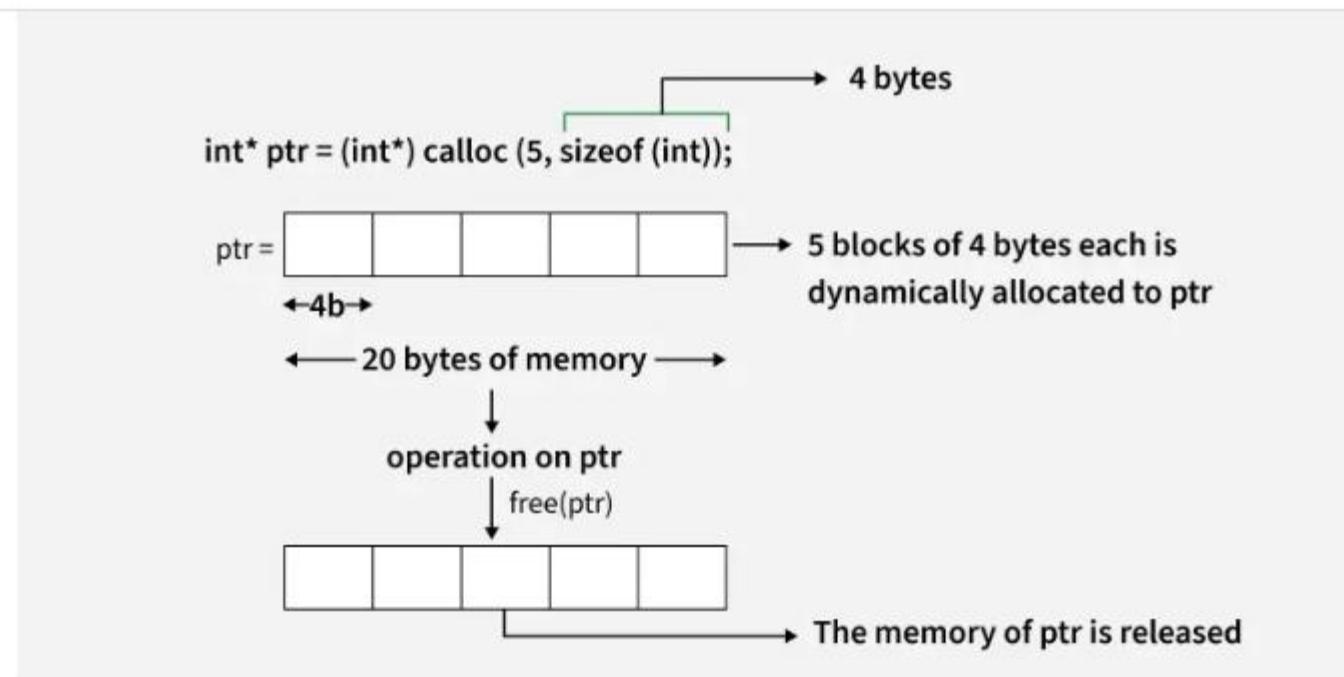
    // Free the memory after completing
    // operations
    free(ptr);

    return 0;
}
```

After calling `free()`, it is a good practice to set the pointer to `NULL` to avoid using a "dangling pointer," which points to a memory location that has been deallocated.

```
ptr = NULL;
```

# free()



# realloc()

,realloc() function is used to resize a previously allocated memory block. It allows you to change the size of an existing memory allocation without needing to free the old memory and allocate a new block.

- Syntax
- `realloc(ptr, new_size);`
- where, `ptr` is the pointer to the previously allocated memory block and `new_size` is the reallocated size that the memory block should have in bytes.
- This function returns a pointer to the newly allocated memory, or `NULL` if the reallocation fails. If it fails, the original memory block remains unchanged.

# Example:

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

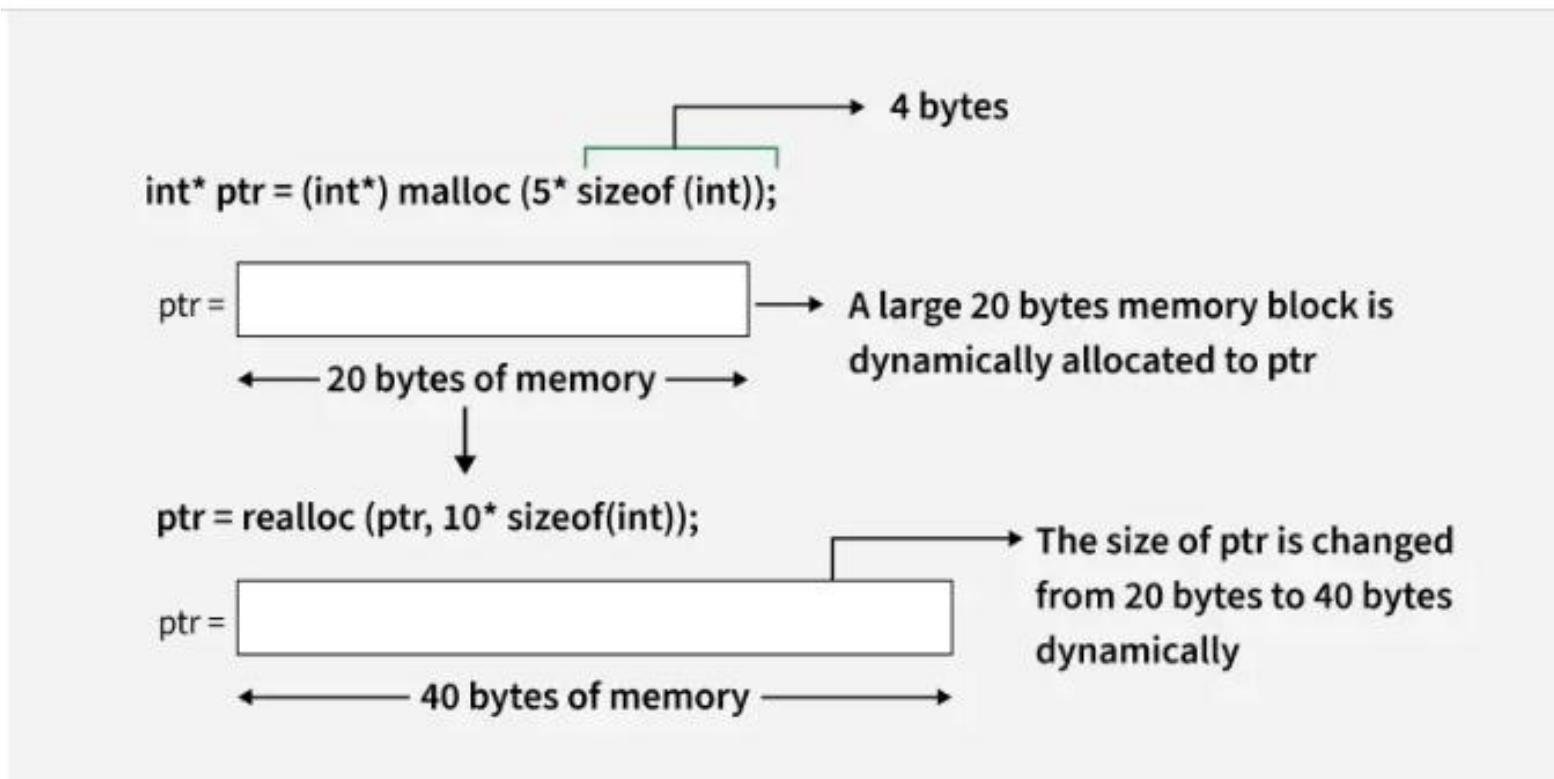
int main() {
    int *ptr = (int *)malloc(5 * sizeof(int));

    // Resize the memory block to hold 10 integers
    ptr = (int *)realloc(ptr, 10 * sizeof(int));

    // Check for allocation failure
    if (ptr == NULL) {
        printf("Memory Reallocation Failed");
        exit(0);
    }

    return 0;
}
```

# Diagram :



# malloc vs calloc

S.No.	malloc()	calloc()
1.	malloc() is a function that creates one block of memory of a fixed size.	calloc() is a function that assigns a specified number of blocks of memory to a single variable.
2.	malloc() only takes one argument	calloc() takes two arguments.
3.	malloc() is faster than calloc.	calloc() is slower than malloc()
4.	malloc() has high time efficiency	calloc() has low time efficiency
5.	malloc() is used to indicate memory allocation	calloc() is used to indicate contiguous memory allocation
6.	Syntax : void* malloc(size_t size);	Syntax : void* calloc(size_t num, size_t size);
7.	malloc() does not initialize the memory to zero	calloc() initializes the memory to zero
8.	malloc() does not add any extra memory overhead	calloc() adds some extra memory overhead

# Issues Associated with Dynamic Memory Allocation

- Memory Leaks: Failing to free dynamically allocated memory leads to memory leaks, exhausting system resources.
- Dangling Pointers: Using a pointer after freeing its memory can cause undefined behavior or crashes.
- Fragmentation: Repeated allocations and deallocations can fragment memory, causing inefficient use of heap space.
- Allocation Failures: If memory allocation fails, the program may crash unless the error is handled properly

# C Programming

## file management

# What is file in c?

- a file is used to store data permanently, meaning it remains available even after the program ends.
- Files allow reading, writing, and modifying data efficiently instead of using temporary variables in memory.
- Types of Files in C
  - 1. Text Files (.txt) – Stores data in a human-readable format.
  - 2. Binary Files (.bin) – Stores data in a machine-readable format.

# File Handling in C

- C provides functions from the stdio.h library to work with files.
- Basic Operations:
  - - Creating & Opening a File (`fopen`)
  - - Writing to a File (`fprintf`, `fputc`, `fputs`)
  - - Reading a File (`fscanf`, `fgetc`, `fgets`)
  - - Closing a File (`fclose`)

# file handling

- explanation : File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as fopen(), fwrite(), fread(), fseek(), fprintf(), etc. to perform input, output, and many different C file operations in our program.

# Need of File Handling in C

- So far, the operations in C program are done on a prompt/terminal in which the data is only stored in the temporary memory (RAM). This data is deleted when the program is closed. But in the software industry, most programs are written to store the information fetched from the program. The use of file handling is exactly what the situation calls for.
- File handling allows us to read and write data on files stored in the secondary memory such as hard disk from our C program.

# C File Operation

- 1. Creating a new file.
- 2. Opening an existing file.
- 3. Reading from file.
- 4. Writing to a file.
- 5. Moving to a specific location in a file.
- 6. Closing a file.

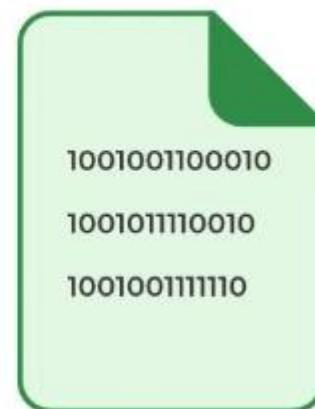
# Components in C File Handling

- 1. File file is a container of data. It can be classified into two types based on the way the file stores the data. They are as follows:
- types of files in c

## Types of Files in C



file.txt



file.bin

## i) Text File

- A text file contains data in the form of ASCII characters and is generally used to store a stream of characters.
- Each line in a text file ends with a new line character ('\n').
- It can be read or written by any text editor.
- They are generally stored with .txt file extension.
- Text files can also be used to store the source code.

## ii) Bin file

- A binary file contains data in binary form (i.e. 0's and 1's) instead of ASCII characters. They contain data that is stored in a similar manner to how it is stored in the main memory.
- The binary files can be created only from within a program and their contents can only be read by a program.
- More secure as they are not easily readable.
- They are generally stored with .bin file extension.

# File Pointer

- A file pointer is a reference to a particular position in the opened file. It is used in file handling to perform all file operations such as read, write, close, etc. We use the FILE macro to declare the file pointer variable. The FILE macro is defined inside <stdio.h> header file.
- FILE\* pointer\_name;

# File Operations

File operation	Declaration & Description
<b>fopen() - To open a file</b>	<p>Declaration: FILE *fopen (const char *filename, const char *mode)</p> <p>fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist.</p> <pre>FILE *fp; fp=fopen ("filename", "mode"); Where, fp - file pointer to the data type "FILE". filename - the actual file name with full path of the file. mode - refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.</pre>
<b>fclose() - To close a file</b>	<p>Declaration: int fclose(FILE *fp);</p> <p>fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below.</p> <pre>fclose (fp);</pre>
<b>fgets() - To read a file</b>	<p>Declaration: char *fgets(char *string, int n, FILE *fp)</p> <p>fgets function is used to read a file line by line. In a C program, we use fgets function as below.</p> <pre>fgets (buffer, size, fp); where, buffer - buffer to put the data in. size - size of the buffer fp - file pointer</pre>
<b>fprintf() - To write into a file</b>	<p>Declaration:</p> <pre>int fprintf(FILE *fp, const char *format, ...);</pre> <p>fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below.</p> <pre>fprintf (fp, "some data"); or fprintf (fp, "text %d", variable_name);</pre>

# 1. Opening The File

- `FILE* fopen(*file_name, *access_mode);`
- Parameters
  - `file_name`: name of the file when present in the same directory as the source file. Otherwise, full path.
  - `access_mode`: Specifies for what operation the file is being opened.
- Return Value
  - If the file is opened successfully, returns a file pointer to it.
  - If the file is not opened, then returns NULL.

# Opening Mode

# Opening Mode

<b>ab</b>	Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
<b>r+</b>	Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file.
<b>rb+</b>	Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
<b>w+</b>	Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.
<b>wb+</b>	Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
<b>a+</b>	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens the file in both reading and append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
<b>ab+</b>	Open for both reading and appending in binary mode. If the file does not exist, it will be created.

# Example : Opening a file

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

int main() {

    // File pointer to store the
    // value returned by fopen
    FILE* fptr;

    // Opening the file in read mode
    fptr = fopen("filename.txt", "r");

    // checking if the file is
    // opened successfully
    if (fptr == NULL) {
        printf("The file is not opened.");
    }
    return 0;
}
```

The file is not opened because it does not exist in the source directory. But the `fopen()` function is also capable of creating a file if it does not exist.

**Note:** It is essential to check for `NULL` values that might be returned by the `fopen()` function to avoid any errors.

## Output

The file is not opened.

# Create a File in C

- The fopen() function can not only open a file but also can create a file if it does not exist already. For that, we have to use the modes that allow the creation of a file if not found such as w, w+, wb, wb+, a, a+, ab, and ab+.

# Create a file in C

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

int main() {

    // File pointer
    FILE* fptr;

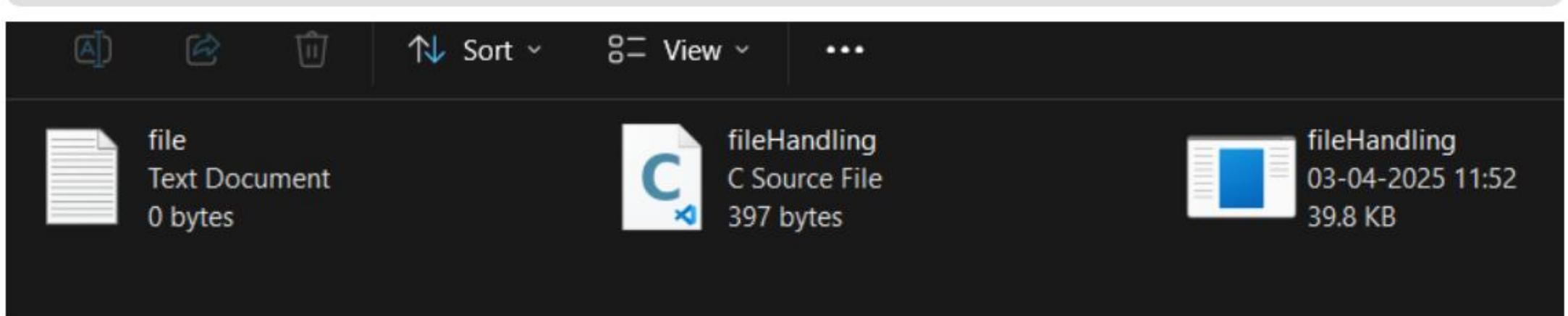
    // Creating file using fopen()
    // with access mode "w"
    fptr = fopen("file.txt", "w");

    // checking if the file is created
    if (fptr == NULL)
        printf("The file is not opened.");
    else
        printf("The file is created Successfully.");
    return 0;
}
```

# Creating a file

## Output

The file is created successfully.



# Write a file

The file write operations can be performed by the functions `fprintf()` and `fputs()`. C programming also provides some other functions that can be used to write data to a file such as:

Function	Description
<code>fprintf()</code>	Similar to <code>printf()</code> , this function uses formatted string and variable arguments list to print output to the file.
<code>fputs()</code>	Prints the whole line in the file and a newline at the end.
<code>fputc()</code>	Prints a single character into the file.
<code>fputw()</code>	Prints a number to the file.
<code>fwrite()</code>	This function writes the specified number of bytes to the binary file.

```
// Creating file using fopen()
// with access mode "w"
fptr = fopen("file.txt", "w");

// Checking if the file is created
if (fptr == NULL)
    printf("The file is not opened.");
else{
    printf("The file is now opened.\n");
    fputs(data, fptr);
    fputs("\n", fptr);

    // Closing the file using fclose()
    fclose(fptr);
    printf("Data successfully written in file "
        "file.txt\n");
    printf("The file is now closed.");
}

return 0;
}
```

# Example

# Reading From a File

---

The file read operation in C can be performed using functions **fscanf()** or **fgets()**. Both the functions performed the same operations as that of **scanf()** and **gets** but with an additional parameter, the file pointer. There are also other functions we can use to read from a file. Such functions are listed below:

Function	Description
<a href="#"><b>fscanf()</b></a>	Use formatted string and variable arguments list to take input from a file.
<a href="#"><b>fgets()</b></a>	Input the whole line from the file.
<a href="#"><b>fgetc()</b></a>	Reads a single character from the file.
<a href="#"><b>fgetw()</b></a>	Reads a number from a file.
<a href="#"><b>fread()</b></a>	Reads the specified bytes of data from a binary file.

# Example :

[Input and Output](#) [Control Flow](#) [Functions](#) [Arrays](#) [Strings](#) [Pointers](#) [Preprocessors](#) [File Handling](#) [Programs](#)

```
int main() {
    FILE* fptr;

    // Declare the character array
    // for the data to be read from file
    char data[50];
    fptr = fopen("file.txt", "r");

    if (fptr == NULL) {
        printf("file.txt file failed to open.");
    }
    else {

        printf("The file is now opened.\n");

        // Read the data from the file
        // using fgets() method
        while (fgets(data, 50, fptr)
              != NULL) {

            // Print the data
            printf("%s", data);
        }

        // Closing the file using fclose()
        fclose(fptr);
    }
    return 0;
}
```

# Closing a file

The **fclose()** function is used to close the file. After successful file operations, you must always close a file to remove it from the memory.

Syntax:

```
fclose(file_pointer);
```



[More](#)

# C Programming

## CLI- Command Line Interface

Basics

# Basics

- A Command Line Interface (CLI) in C allows users to interact with a program via text-based commands instead of a graphical user interface (GUI). CLI programs take input, process data, and display output directly in a terminal.

# Example :

## Example: Using Command-Line Arguments (`argc`, `argv`)

C

 Copy

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <your_name>\n", argv[0]);
        return 1;
    }

    printf("Hello, %s!\n", argv[1]);
    return 0;
}
```

### Explanation:

- `argc` : Number of arguments passed.
- `argv[]` : Array containing the arguments (`argv[0]` is the program name).
- Running `./program Vraj` prints `Hello, Vraj!`.

# Additional Topics : Storage classes in c

- Refer from this website : <https://www.geeksforgeeks.org/storage-classes-in-c/>

# Thank You

want to learn about C++?