C Programming

MD: Abu Bokor Siddik.

Rajshahi Polytechnic Institute.

Date: 07-12-2024

*** TABLE OF CONTENTS ***

Chapter-4: Input-Output Operation -

- © Type of I/O operation.
- © Formatted Unformatted input/output operation.
- © Practice set-04.

Chapter-5: Branching and Looping Statements -

- © Statements & type of statements.
- © Conditional & unconditional program control flow.
- © Practice set – 5.

>> Project-1= Number guessing game.

Chapter-6: Function -

- © Definition of function.
- © Types of functions.
- © Create and call/declaration function & value return.
- © Call by value, reference.
- © Function argument & prototype.
- © Recursive & iterative function.
- © Static, automatic, register & external storage variable.
- © Practice set-6.

Chapter-7: Arrays -

- © Definition of arrays & syntax.
- © Necessity of using array & Applications.
- © Types of arrays.
- © Array declaration and initialization.
- © Accessing elements & arrays in memory.
- © Program using arrays.
- © Passing arrays to functions.
- © Practice set-7.

Chapter-8: Pointer-

- © Definition of pointer.
- © Advantage of pointer.
- © Pointer declaration & initialization.
- © Pointer expression.
- © Practice set-8.

Chapter-9: String-

- © What is String?
- © Declaration & initialization.
- © Taka a string from the user.
- © Gets () and puts ().
- © Declaring a sting using pointers.
- © String manipulation.
- © Standard library functions for strings.
- © Practice set-9.

Chapter-10: Structures-

- © What is structure?
- © Declaration and initialization.
- © Structure in memory.
- © Pointer to Structure.
- © Arrow Operator.
- © Passing structure to a function.
- © Typedef keyword.
- © Practice set:10.

Chapter-11: Union-

- © What is union?
- © Declaration & initialization.
- © Structure & Union.
- © Enumeration-
- © Practice set-11.

Chapter-12: File Operation-

- © What is File operation?
- © File open & closing.
- © File read write operation.
- © Binary file operation.
- © Practice set-12.

Chapter-13: Pre-processor directive in C-

- © Definition.
- © Pre-processor & their functions.
- © Definition of deader file and list standard header files.
- © Process of including header file in routine.
- © Constant directive.

- © Conditional compilation directive.
- © Pragma directive.
- © Special directive.
- © Uses of macro.
- © Advantage of macros over function in programming.
- © Practice set-13.

Chapter-14: Dynamic Memory Allocation.

- © Dynamic memory allocation.
- © Function for DMA in C.
- © Practice set-14.

>> Project-2: Snake gun game.

All code link in here:
https://github.com/AbuBokorSiddik67/C_programming/tree/master/

Chapter-0: Introduction to C programming -

What is programming?

> Computer programming is a medium for us to communicate with computers. Just like we use ‘Bangla’ or ‘English’ to communicate with each other, programming is a way for us to deliver our instructions to the computer.

History of C programming-

The history of the C programming language is fascinating, as it evolved through several stages and influenced many modern programming languages.

Origins of C

C was developed in the early 1970s at Bell Labs by Dennis Ritchie and Ken Thompson. It originated as a byproduct of efforts to improve an earlier language called B, which was developed by Thompson. B itself was based on the BCPL (Basic Combined Programming Language) created by Martin Richards in the 1960s. B, however, had limitations in terms of efficiency and lacked support for data types, which made it less suited for system programming.

Development of C

To address these limitations, Dennis Ritchie began working on C in 1971. The goal was to create a programming language that combined the power and efficiency needed to write an operating system. This led to the development of C, which provided better low-level access to memory, a variety of data types, and direct operations on hardware. By 1973, C was mature enough to be used for rewriting the UNIX operating system, a major milestone that showcased C’s capabilities and portability.

Key Features Introduced

C introduced several concepts that set it apart from other languages of its time, such as:

Low-level memory access: Allowed efficient manipulation of data.

Structured programming: Features like functions, loops, and conditional statements.

Data types and operators: Made it more versatile and easier to write efficient code.

Standardization and Spread

In the late 1970s, C started to gain popularity beyond Bell Labs, primarily due to its association with UNIX, which was also spreading in academia and industry. To ensure compatibility across systems, ANSI (American National Standards Institute) established

a standard for C, known as ANSI C, in 1989. This standardization increased its adoption further as it provided a unified specification.

Influence on Modern Programming Languages

The success of C influenced many other programming languages that followed, such as C++, C#, Java, and Objective-C. These languages borrowed syntax, data structures, and concepts from C, making it foundational in computer science education and system programming.

Modern Uses of C -

Even today, C remains highly relevant in various domains, especially in operating systems, embedded systems, and high-performance applications, due to its efficiency and control over hardware.

Installation -

We will use VS Code as our code editor to write our code and install MinGW gcc

compiler to compile our C program.

Compilation is the process of translating high-level source code written in programming languages like C into machine code, which is the low-level code that a computer's CPU can execute directly. Machine code consists of binary instructions specific to a computer's architecture.

We can install VS Code and MinGW from their respective websites.

Recourse link1: https://en.wikipedia.org/wiki/C_(programming_language)

Recourse link1: https://code.visualstudio.com/download

Chapter-1: Basic variables, constants & Keywords -

Variables -

In C programming, a variable is a named location in memory used to store data. Variables act as storage containers for values that can change during the execution of a program. Each variable in C has a specific data type, which defines the type of data it can hold (e.g., integers, floating-point numbers, characters). Variables are essential for performing calculations, storing input from users, and managing data throughout the program.

Key Characteristics of Variables in C

1. **Data Type**: Each variable in C must have a data type, such as int (for integers), float (for floating-point numbers), char (for characters), etc. The data type defines the size of memory allocated for the variable and the type of data it can hold.
2. **Name**: A variable's name (or identifier) is given by the programmer. It must start with a letter or underscore (_) and can contain letters, digits, or underscores (e.g., total, age, _score).
3. **Memory Allocation**: When a variable is declared, the compiler allocates a specific amount of memory based on its data type. For instance, an int variable might take 4 bytes of memory, while a char variable usually takes 1 byte.
4. **Value**: The actual data stored in a variable. The value of a variable can change throughout the program, hence the name "variable."
5. **Scope**: The scope of a variable is the part of the program where the variable can be accessed. For example, variables declared inside a function can only be used within that function (local scope).
6. **Lifetime**: The lifetime of a variable refers to how long the variable remains in memory. Variables declared within a function only exist while that function is running, whereas global variables persist throughout the program's runtime.

Variables initialization:

A=10; a=20; b ,c ,d=10; etc.

Constants -

An entity whose value does not change is called as a constant. A variable is an entity whose value can be changed.

Types of constants-

Primarily, there are three types of constants:

1. Integer Constant → 1,6,7,9

2. Real Constant → 322.1, 2.5 ,7.0

3. Character Constant → 'a', '$', '@' (must be enclosed within single quotes)

Keywords -

These are reserved words, whose meaning is already known to the compiler. There are 32 keywords available in C.

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

Scape sequence -

In C programming, **e**scape sequences are special character sequences starting with a backslash (\) that represent non-printable or special characters. They allow programmers to insert special characters, control characters, or perform specific formatting tasks in a way that is easy to read and write within a string or character literal.

Escape Sequences	Meaning
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null
\a	Bell
\b	Backspace
\f	form Feed
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

Comments -

In C programming, comments are lines or parts of code that are ignored by the compiler. Comments are added by programmers to explain code logic, make the code more readable, or temporarily disable certain code sections during testing or

debugging. They are essential for documenting code and making it easier for others (or yourself in the future) to understand the purpose and function of the code.

Types of Comments in C:

C supports two types of comments:

1. **Single-line Comments**: Begin with // and extend to the end of the line.
2. **Multi-line Comments**: Begin with /* and end with */. They can span multiple lines.
   Shortcut- *Ctrl+?* > comment all line. *Alt + Seclect* > selection line.

Compilation and execution -

Terminal code:

> gcc *filename.c*

> *./a.exc*

Programming structure & boiler plat-

#include<stdio.h>
Int main()
{
  Return 0;
}
	Output:

Explanation of Each Line

#include <stdio.h>
This line includes the standard input-output library, which provides functions like printf and scanf for handling input and output in C programs.

int main()
This is the main function, the entry point of every C program. The int before main() specifies that this function will return an integer value. When the program runs, it starts executing from main().

return 0;
This line ends the main function and returns 0 to the operating system. A return value of 0 typically indicates that the program finished successfully.

Run our first program -

#include<stdio.h>
Int main()
{
Printf(“Hello world”);
Return 0;
}
	Output:
	
	Hello world

Practice set –1:

1. Write a program in c language print you Name, Roll, Collage Name etc in the box.
2. Show alarm output.

Chapter-2: Datatype in C program

Data type - In C programming, data types define the type of data a variable can hold. Data types specify the size, storage, and range of values. C language has several basic and derived data types:

Types of datatypes-

1. Basic Data Types

- **Integer (int)**: Holds integer values without decimal points, typically occupying 2 or 4 bytes depending on the system architecture.
  Example: int num = 5;
- **Character (char)**: Stores a single character, such as 'A' or 'z', using 1 byte. Character data is represented using ASCII values.
  Example: char letter = 'A';
- **Floating Point (float)**: Holds decimal numbers with single precision, typically 4 bytes.
  Example: float price = 19.99;
- **Double (double)**: Similar to float but with double precision (higher accuracy), using 8 bytes.
  Example: double distance = 123.456;

2. Derived Data Types

- **Array**: Collection of elements of the same data type, stored sequentially in memory.
  Example: int numbers[5] = {1, 2, 3, 4, 5};
- **Pointer**: Stores memory addresses. A pointer to an integer,
  for example, is defined as int *ptr;.
- **Structure (struct)**: Allows grouping of variables of different data types into a single unit. Example:

```
Code:-
struct Person
{
  char name[50];
  int age;
};
```

Union: Similar to structures, but all members share the same memory location. Example:

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

3. Void Data Type

- **Void (void)**: Represents an absence of data type. Often used with functions that do not return a value (e.g., void main() {}) or pointers that don't have a defined data type.

4. Enumeration (enum)

- Used to define named integer constants, improving code readability. Example:

Code-
enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};

Modifiers

- **Signed and Unsigned**: Can be applied to integer and character types. Unsigned types allow only positive values but double the maximum value.
- **Short and Long**: Modify the size of integer types. For example, short int is typically 2 bytes, whereas long int is 4 or 8 bytes.

Each data type has a specific role and range, and they're selected based on the requirements of the program, such as memory efficiency or accuracy.

Format specifier-

Format specifiers in C are used to take inputs and print the output of a type. The symbol we use in every format specifier is %. Format specifiers tell the compiler about the type of data that must be given or input and the type of data that must be printed on the screen.

Data type		Format specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

Type casting/conversion.

Type casting- Converting one type of data to another type.

Int a=3.21;

Float B;

B= (float) a;

Printf(B);

Output: 3

An Arithmetic operation between

• int and int → int

• int and float → float

• float and float → float

Example: o 5/2 becomes 2 as both the operands are int o 5.0/2 becomes 2.5 as one of the operands is float o 2/5 becomes 0 as both the operands are int

NOTE: In programming, type compatibility is crucial. For int a = 3.5;, the float 3.5 is demoted to 3, losing the fractional part because a is an integer. Conversely, for float a = 8;, the integer 8 is promoted to 8.0, matching the float type of a and retaining precision.

Basic input & output -

In C programming, input and output operations are essential for interacting with the user. We use standard library functions like printf for output and scanf for input. Here's an overview:

Output using printf:

a. printf is used to display text or values on the screen.
b. Syntax: printf("format string", variables);
c. Example:

```
Code-
#include <stdio.h>
int main()
{
   printf("Hello, World!\n");
   return 0;
}
```

d. Here, printf displays the string "Hello, World!" followed by a newline (\n).
e.

Input using scanf:

f. scanf is used to take input from the user.
g. Syntax: scanf("format string", &variable);
h. The & symbol (address-of operator) is used with variables to store the input directly at the variable's memory address.
i. Example:

```
Code
#include <stdio.h>
int main()
{
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);
   printf("Your age is: %d\n", age);
   return 0;
}
```

j. Here, the program prompts the user to enter their age, stores the input in the age variable, and then displays it.

Practice_set-2:

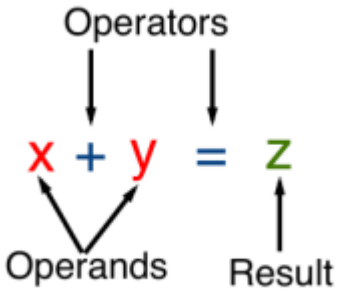
1. Write a C program to calculate area of a rectangle:
   a. Using hard coded inputs.
   b. Using inputs supplied by the user.
2. Calculate the area of circle and modify the same program to calculate the volume of a cylinder given its radius and height.
3. Write a program to convert Celsius to Fahrenheit.
4. Write a program to calculate simple interest for set of values representing principal, number of years and rate of interest.

Chapter-3: Operator in C

What is operator & operands -

Operator: In C language, Operator is a special symbol or sign of word which provide instruction to compiler for execute a particular task.

Operand: Operator works with which data or value or constant or variable or expression it's called operand.
Example:



Types of operators -

Six types of operators in C program-

1. Arithmetic Operator-
   It's two type-
   a. Binary.> Operation with two operand.
      - Addition (+).
      - Subtraction (-).
      - Multiplication (*).
      - Division (/).
      - Modulus (%).

      Example:
      A=5,B=10,C;
      C=A+B;
      Result C= 10

   b. Unary.> Operation with one operand.
      - Increment (++).> add or subtracts one.
        1. (++) Pre-fix increment.
        2. Post-fix increment (++).
           Ex. A=2;
           ++A = 3;
           A++ = 2; > After one add.
      - Decrement.
        3. (--) Pre-fix decrement.
        4. Post-fix decrement (--).
           Ex. A=2;
           --A = 1;
           A-- =2; > After one subtracts

2. Relational Operator -

<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal	A<=B
>=	Greater than or equal	A>=B
==	Equal	A==B
!=	Not equal	A!=B

3. Logical Operator – Which operator works with expression is called logical operator. Its return 0 or 1.

And	&&	1&&1=1	1	1 = 1
OR	||	1||0=0	1	0 = 0
NOT	!	!1=0	0	0 = 0

Explanation-
Here > a=2, b=2, c=4;
And- when all input is high then output is high else low.
Ex. (a==b) && (b==c).> value is false (0).
OR- when any input is high then output is high.
Ex. (a==b) || (b==c).> value is true (1).
NOT- Its operator works like compliment that means 1 or 0 and 0 or 1.
Ex. !(b==c).> value is 1.

4. Bitwise Operator – Bitwise operator works on single bit that means bit by bit . Its operator increase the efficiency of program. Its six types-
   I. Bitwise AND &.
   II. BITWISE OR |.
   III. BITWISE NOT ~.
   IV. BITWISE EX-OR ^.
   V. BIWISE LEFT-SHIFT <<.
   VI. BITWISE RIGHT-SHIFT >>.

5. Assignment Operator - (=) operator is assignment operator.
   Its two types -
   - Short hand operator.

Addition & assignment	Y=Y+X	Y+=X
Subtraction & assignment	Y=Y-X	Y-=X
Multiplication & assignment	Y=Y*X	Y*=X
Division and assignment	Y=Y/X	Y/=X

Modulus and assignment	Y=Y%X	Y%=X

- Multiple assignment operator.
  X=Y=10;

6. Conditional operator -
   Result= Expression1? Expression 2: Expression3;
   If expression1 is true result expression 2, else result expression 3.

Special operator -

Comma (,), size of operator, pointer operator etc.

** Write a program maximum number of two numbers.

Operator precedence & associativity -

The following table lists the operator priority in C:

Operator	Description	Precedence level	Associativity
( ) [ ] . ->> ++ --	Parentheses: grouping or function call Brackets (array subscript) Dot operator (Member selection via object name) Arrow operator(Member selection via pointer) Postfix increment/decrement	1	Left to Right
+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Prefix increment/decrement Logical NOT One's complement Indirection Address (of operand) Type cast Determine size in bytes on this implementation	2	Right to Left
* / %	Multiplication Division Modulus	3	Left to Right
+ -	Addition Subtraction	4	Left to Right
<< >>	Left shift Right shift	5	Left to Right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left to Right
== !=	Equal to Not equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
|	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
||	Logical OR	12	Left to Right
? :	Conditional operator	13	Right to Left
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

Mathematic expression & Value determination -

Expression: It consists with operator, operand or constant.

Ex-1:

>a=5; b=6; c=3;

> (a+b)*(a+c)-90 = -2 answer.

> a+b*c/10 = 6.8 answer.

Practice set-3:

1. Which of the following is invalid in C?

    a. int a=1; int b = a;

    b. int v = 3*3;

    c. char dt = '21 dec 2020';

2. What data type will 3.0/8 – 2 return?

3. Write a program to check whether a number is divisible by 97 or not.

4. Explain step by step evaluation of 3*x/y – z+k, where x=2, y=3, z=3, k=15. 30 + 1 will be:

    a. Integer.

    b. Floating point number.

    c. Character.

5. if a=10, b=5 than write a program determine the value a+b, a-b, a*b, a/b, a%b.

6. if a=10, b=5, c=4 than write a program determine the value (a+b)>c && (b+c)>a.

7. if x=2, y=3, A=x++, B=y-- than write a program determine the value A and B.

Chapter-4: Input-Output Operation -

Type of input output Operation:

Two types of function in C language>

1. Formatted input/output function.
2. Unformatted input/output function.

Formatted unformatted input/output operation:

Formatted I/O Functions

Formatted I/O functions are used to take various inputs from the user and display multiple outputs to the user. These types of I/O functions can help to display the output

to the user in different formats using the format specifiers. These I/O supports all data types like int, float, char, and many more.

Why they are called formatted I/O?

These functions are called formatted I/O functions because we can use format specifiers in these functions and hence, we can format these functions according to our needs.

The following formatted I/O functions will be discussed in this section-

printf()

scanf()

sprintf()

sscanf()

printf():

printf() function is used in a C program to display any value like float, integer, character, string, etc on the console screen. It is a pre-defined function that is already declared in the stdio.h(header file).

Syntax 1:

To display any variable value.

printf("Format Specifier", var1, var2, ..., varn);

Example:

// C program to implement

// printf() function

```
#include <stdio.h>
 //printf() function
// Driver code
int main()
{
// Declaring an int type variable
int a;

// Assigning a value in a variable
a = 20;
```

```
// Printing the value of a variable
printf("%d", a);

return 0;
}
```

```
// C program to implement
// printf() function
#include <stdio.h>

// Driver code
int main()
{
// Displays the string written
// inside the double quotes
printf("This is a string");
return 0;
}
```

```
// C program to implement
// scanf() function
#include <stdio.h>

// Driver code
int main()
{
int num1;

// Printing a message on
// the output screen
printf("Enter a integer number: ");

// Taking an integer value
// from keyboard
scanf("%d", &num1);

// Displaying the entered value
printf("You have entered %d", num1);

return 0;
}
```

```
// C program to implement
// the sprintf() function
#include <stdio.h>

// Driver code
int main()
{
char str[50];
int a = 2, b = 8;

// The string "2 and 8 are even number"
// is now stored into str
sprintf(str, "%d and %d are even number",
a, b);

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

```
// C program to implement
// sscanf() function
#include <stdio.h>

// Driver code
int main()
{
char str[50];
int a = 2, b = 8, c, d;

// The string "a = 2 and b = 8"
// is now stored into str
// character array
sprintf(str, "a = %d and b = %d",
a, b);

// The value of a and b is now in
// c and d
sscanf(str, "a = %d and b = %d",
&c, &d);

// Displays the value of c and d
printf("c = %d and d = %d", c, d);
return 0; }
```

Unformatted Input/Output functions

Unformatted I/O functions are used only for character data type or character array/string and cannot be used for any other datatype. These functions are used to read single input from the user at the console and it allows to display the value at the console.

Why they are called unformatted I/O?

These functions are called unformatted I/O functions because we cannot use format specifiers in these functions and hence, cannot format these functions according to our needs.

The following unformatted I/O functions will be discussed in this section-

getch()

getche()

getchar()

putchar()

gets()

puts()

putch()

getch():

getch() function reads a single character from the keyboard by the user but doesn't display that character on the console screen and immediately returned without pressing enter key. This function is declared in conio.h(header file). getch() is also used for hold the screen.

Syntax:

getch();

or

variable-name = getch();

```
// C program to implement
// the getche() function
#include <conio.h>
#include <stdio.h>
```

Example:

```
// C program to implement
// getch() function
#include <conio.h>
#include <stdio.h>
// Driver code
int main()
{
printf("Enter any character: ");
// Reads a character but
// not displays
getch();
return 0;
}
```

```
// Driver code
int main()
{
printf("Enter any character: ");

// Reads a character and
// displays immediately
getche();
return 0;
}
```

```
// C program to implement
// the getchar() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
// Declaring a char type variable
char ch;

printf("Enter the character: ");

// Taking a character from keyboard
ch = getchar();

// Displays the value of ch
printf("%c", ch);
return 0;
}
```

```
// C program to implement
// the putchar() function
#include <conio.h>
#include <stdio.h>
```

```
// Driver code
int main()
{
char ch;
printf("Enter any character: ");

// Reads a character
ch = getchar();

// Displays that character
putchar(ch);
return 0;
}
```

```
// C program to implement
// the gets() function
#include <conio.h>
#include <stdio.h>

// Driver code
int main()
{
// Declaring a char type array
// of length 50 characters
char name[50];

printf("Please enter some texts: ");

// Reading a line of character or
// a string
gets(name);

// Displaying this line of character
// or a string
printf("You have entered: %s",
name);
return 0;
}
```

```
// C program to implement
// the puts() function
#include <stdio.h>

// Driver code
int main()
{
char name[50];
printf("Enter your text: ");

// Reads string from user
gets(name);

printf("Your text is: ");

// Displays string
puts(name);

return 0;
}
```

```
// C program to implement
// the putch() functions
#include <conio.h>
```

```
#include <stdio.h>

// Driver code
int main()
{
char ch;
printf("Enter any character:\n ");

// Reads a character from the keyboard
ch = getch();

printf("\nEntered character is: ");

// Displays that character on the
console
putch(ch);
return 0;
}
```

Practice set-04 -

1. Input two float type variables and add this number.
2. Input character for user and display.
3. Write a program use all function on formatted & unformatted operation.
4. An input string and determine value of length.
5. Input a string and convert a lower case.
6. Input a sting and convert upper case.
7. Input two string and comparison its.
8. Input two string and concatenate its.
9. Program to print numbers from one to one hundred divisible by 7.
10. Write a program area of tringle from user input.
11. Write a program as your wise.

Chapter-5: Branching and Looping Statements

Statements & type of statements-

A statement in C language is a single instruction that directs the program to perform a specific task. It usually ends with a semicolon (;).

The statement are two type-

- Simple statements= A simple statement in C language is a single instruction or command that performs a specific task. It typically ends with semicolon (;).
  Example:

```
Int a=6; //A variable declaration and assignment.
Printf("Hello world"); // A print statement.
```

- Compound statements= A compound statement in C is a block of multiple simple statements enclosed within {and} (curly braces). Its instructions as a single unit.
  Example:

```
{
   Int a=10;
   Printf("value is :%d", a);
}
```

Conditional & unconditional program control flow-

Program control flow: Control flow in C determines the order in which statements are executed during a program's runtime. Control flow can be classified into conditional and unconditional categories.

Its two types:

- Branching statements.
- Looping statements.

Branching statements: -

1. Conditional Control Flow-

Conditional control flow is based on the evaluation of a condition. The program makes decisions and executes specific blocks of code based on whether a condition is **true** or **false**.

Key Conditional Constructs

1. **if Statement**
   a. Executes a block of code if the condition is true.

```
Code-
if (x > 0) {
   printf("Positive number\n");
}
```

2. **if-else Statement**
    a. Executes one block if the condition is true; otherwise, executes another block.

```
Code
if (x > 0)
{
   printf("Positive number\n");
}
else
 {
   printf("Non-positive number\n");
}
```

3. **Else if statement.**

```
Code
If( x>0 )
{
 Printf(“Hello world”);
}
Else if ( x > 10)
{
Printf(“RPI”);
}
Else if ( x > 10)
{
Printf(“RPI”);
}
else
 {
   printf("Wrong number\n");
}
```

Nested Statement- A nested statement is simply one statement placed inside another. It is frequently used in situations where hierarchical logic or repetitive processes need

to be implemented. Proper use of indentation and comments is important to maintain code readability.

4. **Nested if**

```
Code
if (x > 0)
{
   if (x % 2 == 0)
}
```

5. **Nested if-else**
    a. An if-else statement inside another if-else.

```
Code
if (x > 0)
{
   if (x % 2 == 0)
{
      printf("Positive even number\n");
 }
else
{
      printf("Positive odd number\n");
 }
else {
   printf("Non-positive number\n");
}
```

6. **else-if Ladder**
    a. Tests multiple conditions sequentially.

```
Code
if (x > 0) {
   printf("Positive\n");
} else if (x == 0) {
   printf("Zero\n");
}
// here you can use many condition.
else {
   printf("Negative\n");
}
```

7. **switch Statement**

a. Selects a block of code to execute based on the value of a variable or expression.

```
switch (x) {
  case 1:
    printf("One\n");                    // Here you can use any
Character.
    break;
  case 2:
    printf("Two\n");
    break;
  default:
    printf("Other\n");
}
```

// nested switch your work

Characteristics of Conditional Control Flow

- **Based on Conditions:** Execution depends on logical or relational conditions.
- **Selective Execution:** Only the block corresponding to the true condition is executed.

2. Unconditional Control Flow

Unconditional control flow does not depend on conditions. It allows the program to jump to specific points without any evaluation.

Key Unconditional Constructs

1. **goto Statement**
    a. Transfers control to a labeled statement unconditionally.

```
Code
int x = 5;
if (x == 5)
{
  goto label;
}
printf("This will be skipped.\n");
label:
  printf("Jumped to label.\n");
```

2. **break Statement**
    a. Exits the nearest enclosing loop or switch statement.

```
Code
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break;
   }
   printf("%d\n", i);
}
```

3. **continue Statement**
    a. Skips the rest of the current loop iteration and proceeds with the next iteration.

```
Code
for (int i = 0; i < 5; i++)
{
   if (i == 2)
{
      continue;
 }
   printf("%d\n", i);
}
```

4. **return Statement**
    a. Exits from a function and optionally returns a value.

```
Code
int add(int a, int b) {
   return a + b;
}
```

Characteristics of Unconditional Control Flow

- **No Conditions:** Flow is redirected without any checks or conditions.
- **Direct Execution:** Forces control to jump to another part of the program.

Looping statements: -

Loop: A loop in C programming is a control structure that allows you to repeatedly execute a block of code as long as a specified condition is true. Loops are fundamental to programming, enabling automation of repetitive tasks efficiently.

Loops are two types in C language:

- Conditional looping.
- Unconditional looping.

Explain:

Conditional Looping-

- For loop.
- While loop.
- Do loop.

>>

1. **for loop**:
    a. Used when the number of repetitions is known in advance.
    b. Syntax:

```
Code
for (initialization; condition; increment/decrement)
{
   // Code block to execute
}
```

Example:

```
Code
for(int i = 1; i <= 5; i++) {
   printf("%d\n", i);
}
```

This prints numbers from 1 to 5.

2. **while loop**:
    a. Used when the number of repetitions is not known beforehand, and the condition is checked before executing the code block.
    b. Syntax:

```
Code
while(condition) {
   // Code block to execute
}
```

    c. Example:

```
Code
int i = 1;
while(i <= 5) {
   printf("%d\n", i);
   i++;
}
```

This also prints numbers from 1 to 5.

3. **do-while loop**:
    a. Similar to the while loop, but ensures that the code block runs at least once because the condition is checked after execution.
    b. Syntax:

```
Code
do {
  // Code block to execute
} while(condition);
```

    c. Example:

```
Code
int i = 1;
do {
  printf("%d\n", i);
  i++;
} while(i <= 5);
```

This also prints numbers from 1 to 5.

Key Concepts:

- **Initialization**: Setting up the starting point of the loop.
- **Condition**: Determines whether the loop should continue or stop.
- **Increment/Decrement**: Updates the loop control variable to eventually terminate the loop.

Unconditional looping-

An unconditional loop in C refers to a loop that does not rely on an explicit condition to terminate. These loops typically continue indefinitely unless a specific instruction, such as a break or exit () statement, interrupts them. Unconditional loops are also known as infinite loops.

Types of Unconditional Loops:

1. **Using while(1) or while(true)**:
    a. The condition 1 (true) ensures the loop runs indefinitely.
    b. Example:

```
Code
#include <stdio.h>

int main() {
  while(1) { // Infinite loop
    printf("This is an infinite loop.\n");
    break; // Ends the loop
```

```
    }
    return 0;
}
```

2. **Using for(;;)**:
    a. The syntax omits all three components (initialization, condition, increment/decrement) of the for loop, resulting in an infinite loop.
    b. Example:

```
Code
#include <stdio.h>

int main() {
    for(;;) { // Infinite loop
        printf("This is another infinite loop.\n");
        break; // Ends the loop
    }
    return 0;
}
```

3. **Using do-while(1)**:
    a. Similar to while(1), but the code block executes at least once before the condition is checked.
    b. Example:

```
Code
#include <stdio.h>

int main() {
    do {
        printf("This is a do-while infinite loop.\n");
        break; // Ends the loop
    } while(1);
    return 0;
}
```

Controlling Unconditional Loops: -

1. **break/continue Statement**:
    a. Immediately exits the loop when encountered.
    b. Example:

```
Code
int count = 0;
while(1) {
    if(count == 5) {
        break; // Exit loop when count is 5
```

```
    }
    printf("Count: %d\n", count);
    count++;
}
```

2. **return Statement**:
    a. Exits the entire function (or program) when encountered.
    b. Example:

```
Code
int main() {
    while(1) {
        printf("Exiting...\n");
        return 0; // Exit the program
    }
}
```

3. **exit() Function**:
    a. Terminates the program directly, available through the <stdlib.h> library.
    b. Example:

```
Code
#include <stdlib.h>

int main() {
    while(1) {
        printf("Exiting...\n");
        exit(0); // Ends the program
    }
}
```

Use Cases:

Unconditional loops are commonly used for:

- Event-driven programs (e.g., listening for user input).
- Embedded systems where the program runs indefinitely.
- Simulations or real-time systems.

Note: While unconditional loops can be useful, always ensure there is a mechanism to break the loop to prevent your program from hanging or crashing.

Practice set – 5:

>>Use branching

1. What will be the output of this program int
   a = 10;
   if (a = 11)
   printf("I am 11");
   else printf("I am not 11");
2. Write a program to determine whether a student has passed or failed. To pass, a student requires a total of 40% and at least 33% in each subject. Assume there are three subjects and take the marks as input from the user.
3. Calculate income tax paid by an employee to the government as per the slabs mentioned below:

   Income Slab

   Tax 2.5 – 5.0L 5%

   5.0L - 10.0L 20%

   Above 10.0L 30%

   Note: that there is no tax below 2.5L. Take income amount as an input from the user.

5. Write a program to find whether a year entered by the user is a leap year or not. Take year as an input from the user.

6. Write a program to find greatest of four numbers entered by the user.

>> use loop

1. Write a program to print multiplication table of a given number n.
2. Write a program to print multiplication table of 10 in reversed order.
3. A do while loop is executed:
   - At least once.
   - At least twice.
   - At most once.
4. What can be done using one type of loop can also be done using the other two types of loops-true or false?
5. Write a program to sum first ten natural numbers using while loop.
6. Write a program to implement program 5 using ‘for and do-while' loop.
7. Write a program to calculate the sum of the numbers occurring in the multiplication table of 8. (consider 8*1 to 8*10).
8. Write a program to calculate the factorial of a given number using a for loop.
9. Repeat 8 using while loop.
10. Write a program to check whether a given number is prime or not using loops.

11. Implement 10 using other types of loops.

Project-1: Number guessing game.

We write a program that generate a random number and asks the player to guess it. If the player's guess is higher than the actual number, the program displays "Lower number please". Similarly, if the user's guess is too low, the program prints" Higher number please".

When the user guesses the correct number, the program displays the number of guesses the player used to arrive at the number.

Hints: Use loop and use random number generator.

Project1 link- https://github.com/AbuBokorSiddik67/C_programming/tree/master/

Chapter-5: Function-

Definition: In C programming, a function is a block of code that performs a specific task. Functions help to divide a program into smaller, manageable, and reusable sections. It takes inputs (optional), performs operations, and may return a value. Functions are used to avoid redundancy and improve the modularity and readability of code.

Key Components of a Function:

1. **Function Declaration (Prototype):** Specifies the function's name, return type, and parameters.
2. Example:

```
Code
int add(int a, int b);
```

3. **Function Definition:** The actual implementation of the function, where the operations are written.

Example:

```
Code
```

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

4. **Function Call:** Invokes the function to execute.
5. Example:

```
Code
int result = add(5, 3); // Calls the 'add' function with arguments 5 and 3.
```

Syntax of a Function:

Example:

```
Code
return_type function_name(parameter_list) {
   // Body of the function
   // Statements
   return value; // Optional
}
```

```
Code
#include <stdio.h>

// Function declaration
int multiply(int x, int y);

int main() {
   int result = multiply(4, 5); // Function call
   printf("The product is: %d\n", result);
   return 0;
}

// Function definition
int multiply(int x, int y) {
   return x * y;
}
```

Types of functions:

In C, functions are primarily categorized into **two main types**:

1. Library Functions

- **Definition:** Predefined functions provided by C's standard library.
- **Purpose:** Perform common tasks like input/output, string manipulation, and mathematical calculations.
- **Examples:**
  - **Input/Output Functions:** printf(), scanf() (in stdio.h)
  - **String Functions:** strlen(), strcpy() (in string.h)
  - **Math Functions:** sqrt(), pow() (in math.h)

Note: These functions are ready-to-use and don't require the programmer to define them.

2. User-Defined Functions

- **Definition:** Functions created by the programmer for specific tasks.
- **Purpose:** Improve modularity, reusability, and readability of code by breaking it into manageable blocks.
- **Example:**

```
Code
int add(int a, int b) {
   return a + b;
}
```

- **Subcategories:**

  >No Return, No Parameters

```
Code
void greet() {
   printf("Hello, World!");
}
```

- **No Return, With Parameters**

```
Code
void display(int n) {
   printf("The number is: %d", n);
}
```

- **With Return, No Parameters**

```
Code
int getNumber() {
   return 10;
}
```

- **With Return, With Parameters**

```
Code
```

```
int multiply(int x, int y) {
  return x * y;  }
```

Note:

- **Library Functions:** Predefined by C, such as printf().
- **User-Defined Functions:** Custom functions written by the programmer.

Create, Declare, and Call a Function in C (with Value Return):

A **function in C** is created by declaring it, defining it, and calling it. When a function **returns a value**, it sends a result back to the caller after execution.

Steps to Create and Call a Function:

1. **Declare the Function (Prototype):**
    a. Tells the compiler the function's name, return type, and parameters (optional).
    b. Example:

```
Code
int add(int, int);
```

2. **Define the Function:**
    a. Contains the actual logic (function body).
    b. Example:

```
Code
int add(int a, int b) {
  return a + b; // Returns the sum of a and b
}
```

3. **Call the Function:**
    a. Execute the function by passing the required arguments.
    b. Example:

```
Code
int result = add(5, 10); // Calls the add function
```

Complete Example:

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

```
// Function declaration (prototype)
int add(int a, int b);

int main() {
   int num1 = 5, num2 = 10;

   // Function call
   int result = add(num1, num2);

   // Print the returned value
   printf("The sum is: %d\n", result);

   return 0;
}

// Function definition
int add(int a, int b) {
   return a + b; // Returns the sum
}
```

Output:

The sum is: 15

Function argument & prototype:

Prototype: A **function prototype** is a declaration of a function that specifies its return type, name, and parameter list without defining its body. It serves as a contract to let the compiler know what arguments and return type the function will have.

Syntax of a Function Prototype:

```
Code
return_type function_name(parameter_list);
```

Example of a Function Prototype:

```
Code
int add(int, int); // The function 'add' takes two integers and returns an integer.
```

Arguments: Function arguments are the values passed to a function during its call. These arguments are used to perform operations inside the function. There are four main types of arguments:

1. Formal Arguments

- **Definition:** Variables defined in the function **definition** that receive values from actual arguments during a function call.
- **Characteristics:**
  - Acts as placeholders for actual values.
  - Allocated memory only when the function is called.
- **Example:**

```
Code
void greet(char name[]) { // 'name' is the formal argument
  printf("Hello, %s!\n", name);
}
int main() {
  greet("Alice");
  return 0;
}
```

2. Actual Arguments

- **Definition:** Values or variables passed to a function during the **function call**.
- **Characteristics:**
  - These values are copied to the formal arguments.
  - Can be constants, variables, or expressions.
- **Example:**

```
Code
void greet(char name[]) {
  printf("Hello, %s!\n", name);
}
int main() {
  greet("Alice"); // "Alice" is the actual argument
  return 0;
}
```

3. Default Arguments

- **Definition:** Arguments with a predefined value used when no value is provided during the function call.
- **C-Specific Note: C does not support default arguments directly**, but this behavior can be mimicked using overloading in C++ or by providing multiple function definitions in C.
- **Example in C (using multiple function calls):**

```
Code
```

```
void display(int x, int y) {
   printf("x = %d, y = %d\n", x, y);
}
```

```
int main() {
   display(10, 20); // Passes both arguments
   display(10, 0);  // Mimics default argument for 'y'
   return 0;
}
```

4. Command-Line Arguments

- **Definition:** Arguments passed to the main() function when a program is executed from the command line.
- **Purpose:** Useful for passing runtime parameters.
- **Characteristics:**
    - Handled by the main() function using argc (argument count) and argv (argument vector).
    - argc: Holds the number of command-line arguments.
    - argv: Array of strings holding the arguments.

Syntax:

```
Code
int main(int argc, char *argv[]) {
   // Code here
}
```

Example:

Run the program using the command:

bash

```
Code
./program_name arg1 arg2 arg3
```

Code Example:

Code:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   printf("Number of arguments: %d\n", argc);
   for (int i = 0; i < argc; i++) {
      printf("Argument %d: %s\n", i, argv[i]);
   }
   return 0;
}
```

Output (if executed as ./a.out hello world):

```
Code
Number of arguments: 3
Argument 0: ./a.out
Argument 1: hello
Argument 2: world
```

Comparison of Argument Types

Argument Type	Definition	Example
Formal-Argument	Variables in the function definition that accept actual arguments during a function call.	void add(int a, int b) { ... }
Actual-Argument	Values/variables passed to the function during the call.	add(10, 20);
Default-Argument	Predefined argument values (not natively supported in C but can be mimicked).	display(10, 0);
Command-Line Argument	Input provided at runtime from the command line, handled by argc and argv in main().	./program_name arg1 arg2

Recursive & iterative function:

Recursive

Recursive: A **recursive function** is a function that calls itself during its execution, either directly or indirectly. Recursion helps in solving problems that can be broken down into smaller sub-problems of the same type.

Key Concepts of Recursion

1. **Base Case:**
    a. The condition where the recursion stops.
    b. Prevents infinite recursion.
2. **Recursive Case:**
    a. The function calls itself with modified arguments, aiming to reach the base case.

Syntax of Recursive Function

```
Code
return_type function_name(parameters) {
   if (base_condition) {
      // Base case
      return value;
   } else {
      // Recursive case
      return function_name(modified_parameters);
   }
}
```

Example 1: Factorial Using Recursion

***Problem: Calculate the factorial of a number using recursion*.**

```
Code
#include <stdio.h>

// Recursive function to calculate factorial
int factorial(int n) {
   if (n == 0 || n == 1) { // Base case
      return 1;
   } else {
      return n * factorial(n - 1); // Recursive case
   }
}

int main() {
   int num = 5;
   printf("Factorial of %d is %d\n", num, factorial(num));
```

```
    return 0;
}
```

Output:

Code
Factorial of 5 is 120

Explanation:

1. factorial(5) calls factorial(4), which calls factorial(3)... until factorial(0) (base case).
2. Results are multiplied as the recursion unwinds:

5×4×3×2×1=120>5times \times 4 \times 3 \times 2 \times 1 = 120>5×4×3×2×1=120.

Example 2: Fibonacci Sequence Using Recursion

Problem: Generate the nth Fibonacci number using recursion.

```
Code
#include <stdio.h>

// Recursive function to calculate Fibonacci number
int fibonacci(int n) {
    if (n == 0) { // Base case
        return 0;
    } else if (n == 1) { // Base case
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
    }
}

int main() {
    int n = 6;
    printf("Fibonacci number at position %d is %d\n", n, fibonacci(n));
    return 0;
}
```

Output:

Code
Fibonacci number at position 6 is 8

Explanation:

1. Fibonacci series starts as: 0, 1, 1, 2, 3, 5, 8, ...
2. Recursive calls:

fibonacci(6) = fibonacci(5) + fibonacci(4)
=(fibonacci(4)+fibonacci(3))+(fibonacci(3)+fibonacci(2))= (fibonacci(4) + fibonacci(3)) + (fibonacci(3) + fibonacci(2))=(fibonacci(4)+fibonacci(3))+(fibonacci(3)+fibonacci(2)).

Advantages of Recursion

1. **Simpler Code:** Recursion makes the code more concise and easier to understand for problems with repetitive sub-structures.
2. **Solves Complex Problems:** Ideal for problems like tree traversal, backtracking, etc.

Disadvantages of Recursion

1. **High Memory Usage:** Each recursive call consumes stack memory, which can lead to a stack overflow for large input.
2. **Performance Overhead:** Recursion can be slower than iterative solutions due to repeated function calls.

Example of Recursion vs Iteration

Recursion Example:

```
Code
int sum_recursive(int n) {
   if (n == 0) { // Base case
      return 0;
   } else {
      return n + sum_recursive(n - 1); // Recursive case
   }
}
```

Iteration Example:

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

```
    return sum;
}
```

Iterative

Iterative: An **iterative process** in programming refers to solving a problem using loops to repeat a set of instructions until a specific condition is met. It contrasts with **recursion**, where a function calls itself to repeat the process.

Key Features of Iteration-

1. **Uses Loops:**
    a. Common loops include for, while, and do-while.
2. **Step-by-Step Execution:**
    a. The process moves step-by-step, repeating the block of code until the condition is satisfied.
3. **Explicit Termination:**
    a. The loop exits when a stopping condition is met, preventing infinite loops.
4. **Memory Efficiency:**
    a. Iteration uses less memory compared to recursion as it doesn't involve function call stacks.

Syntax of Iteration-

Using a for Loop:

```
Code
for (initialization; condition; increment) {
    // Code block to repeat
}
```

Using a while Loop:

```
Code
while (condition) {
    // Code block to repeat
}
```

Using a do-while Loop:

```
Code
```

```
do {
   // Code block to repeat
} while (condition);
```

Example of Iteration

Example 1: Factorial Using Iteration

```
Code
#include <stdio.h>

int factorial(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i; // Multiply result by i in each iteration
   }
   return result;
}

int main() {
   int num = 5;
   printf("Factorial of %d is %d\n", num, factorial(num));
   return 0;
}


Output:

Code
Factorial of 5 is 120
```

Example 2: Fibonacci Sequence Using Iteration

```
Code
#include <stdio.h>

void fibonacci(int n) {
   int a = 0, b = 1, next;
   printf("Fibonacci Sequence: %d %d ", a, b);
   for (int i = 3; i <= n; i++) {
      next = a + b; // Calculate next Fibonacci number
      printf("%d ", next);
      a = b; // Update a and b
      b = next;
   }
```

```
}

int main() {
   int n = 6;
   fibonacci(n);
   return 0;
}
```

Output:

mathematica
Code
Fibonacci Sequence: 0 1 1 2 3 5

Advantages of Iteration-

1. **Memory Efficient:**
    a. Iteration does not consume stack memory as recursion does.
2. **Easier Debugging:**
    a. Loops are straightforward to debug compared to recursive function calls.
3. **Faster Execution:**
    a. Avoids the overhead of recursive function calls.

Disadvantages of Iteration

1. **Complex Code for Certain Problems:**
    a. Problems like tree traversal are more complex to solve iteratively compared to recursion.
2. **May Require Additional Data Structures:**
    a. For tasks like backtracking, iterative solutions may need explicit stacks or queues.

Comparison: Iteration vs Recursion-

Aspect	Iteration	Recursion
Definition	Repeats a block of code using loops.	A function that calls itself.
Termination	Controlled by loop conditions.	Controlled by the base case.

Memory Usage	Uses minimal memory.	Uses stack memory for each function call.
Readability	Often less elegant for complex problems.	Easier to read for problems like tree traversal.
Performance	Generally faster due to no function call overhead.	Slower due to repeated function calls.

Static, automatic, register & external storage variable:

>> **Storage classes in C** determine the **scope, lifetime, and linkage** of a variable. Let's explore each in detail:

1. Automatic Storage Class (auto)

- **Default behavior:** Variables declared inside a function or block without a specific storage class are automatically classified as auto.
- **Scope:** Local to the block or function in which it is declared.
- **Lifetime:** Exists only during the execution of the block; memory is deallocated after the block is exited.
- **Keyword:** auto (though rarely used explicitly as it is the default).
- **Storage location:** RAM.

Example:

Code

```
#include <stdio.h>
void example() {
   auto int x = 10; // Implicitly auto
   printf("%d\n", x);
}
```

Output: 10

2. Static Storage Class (static)

- **Scope:** If declared within a function, it is local to that function but retains its value between function calls. If declared globally, its scope is limited to the file in which it is declared.
- **Lifetime:** Entire duration of the program.
- **Keyword:** static

- **Use case:** To preserve a variable's value or restrict global variable access to the current file.

Example:

```
Code
#include <stdio.h>
void example() {
   static int count = 0; // Retains value
between calls
   count++;
   printf("%d\n", count);
}

int main() {
   example(); // Output: 1
   example(); // Output: 2
   example(); // Output: 3
   return 0;
}
```

3. Register Storage Class (register)

- **Scope:** Local to the block or function where declared.
- **Lifetime:** Exists until the block is exited.
- **Storage location:** CPU registers (if available), which are faster than RAM.
- **Keyword:** register
- **Use case:** For variables that need quick access, like loop counters. The compiler may ignore the request if registers are unavailable.

Example:

```
Code
#include <stdio.h>
void example() {
   register int i;
   for (i = 0; i < 5; i++) {
      printf("%d ", i);
   }
}
Output: 0 1 2 3 4
```

4. External Storage Class (extern)

- **Scope:** Global, but accessible across multiple files.
- **Lifetime:** Entire duration of the program.
- **Keyword:** extern
- **Use case:** Used to declare variables shared between files.
- **Storage location:** RAM.

Example (Two Files):

File 1 (file1.c):

```
Code
#include <stdio.h>
int globalVar = 10; // Declared and initialized
```

File 2 (file2.c):

```
Code
#include <stdio.h>
extern int globalVar; // Access globalVar from file1.c
void example() {
   printf("%d\n", globalVar);
}


To compile:

Code
gcc file1.c file2.c -o output

Output: 10
```

Summary Table:

Storage Class	Keyword	Scope	Lifetime	Storage Location	Default Value
Automatic	auto	Local	Block execution	RAM	Garbage
Static	static	Local (file scope)	Entire program	RAM	Zero

Register	register	Local	Block execution	CPU Register	Garbage
External	extern	Global (multi-file)	Entire program	RAM	Zero

Practice set-6:

1. Write a program using function to find average of three numbers.

2. Write a function to convert Celsius temperature into Fahrenheit.

3. Write a function to calculate force of attraction on a body of mass ‘m’ exerted by earth. Consider g = 9.8m/s2.

4. Write a program using recursion to calculate nth element of Fibonacci series.

5. What will the following line produce in a C program:

```
int a = 4;
printf("%d %d %d \n", a, ++a, a++);
```

6. Write a recursive function to calculate the sum of first ‘n’ natural numbers.

7. Write a program using function to print the following pattern (first n lines)

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

Chapter-7: Arrays

Definition of arrays & syntax:

Array: An array is a collection of variables some of data types.

An **array** is a data structure that stores a fixed-size, ordered collection of elements of the same data type. Arrays allow efficient access to their elements using an index. They are commonly used for organizing data and performing operations on a collection of similar items.

Key properties:

1. **Fixed Size**: The size of an array is defined when it is created and cannot be changed.

2. **Homogeneous Elements**: All elements in an array must be of the same type (e.g., integers, floating-point numbers, or characters).
3. **Indexed Access**: Each element can be accessed using an index, starting from 0 in most programming languages.

Syntax:

data_type array_name[size];

Example:

```
Code

int numbers[5];  // Array of 5 integers
```

Necessity of using array & Applications:

Necessity of Using Arrays

Arrays are essential in programming and computer science because they:

1. **Efficient Data Organization**:
   Arrays provide a structured way to store and organize data of the same type, making it easier to manage and manipulate.
2. **Direct Access to Elements**:
   Arrays allow constant-time access to any element using its index, making operations like retrieval fast and efficient.
3. **Memory Optimization**:
   Arrays use contiguous memory locations, reducing overhead compared to storing individual variables.
4. **Simplicity in Code**:
   Arrays simplify the handling of a large number of variables by allowing them to be grouped together under a single name.
5. **Basis for Advanced Data Structures**:
   Arrays serve as the foundation for more complex data structures like stacks, queues, matrices, and hash tables.

Applications of Arrays

1. **Data Storage**:

Arrays are widely used to store and process lists of items such as numbers, names, or objects.

2. **Mathematical Computations**:
   Arrays are essential for implementing matrices and performing operations like addition, multiplication, or solving systems of equations.
3. **Searching and Sorting Algorithms**:
   Arrays are used in algorithms like binary search, merge sort, and quick sort to efficiently search or sort data.
4. **Representation of Multi-dimensional Data**:
   Arrays can represent data in multiple dimensions (e.g., 2D arrays for images or grids, 3D arrays for volumetric data).
5. **Simulation and Modeling**:
   In simulations or modeling, arrays store states or parameters, such as in game development or scientific computations.
6. **Database Management**:
   Arrays are used to store records or entries for temporary processing in databases.
7. **String Manipulations**:
   Arrays of characters are used in string operations, such as in C programming, where strings are represented as character arrays.
8. **Graphics and Image Processing**:
   Arrays are used to store pixel data for images or to represent geometric shapes in 2D or 3D space.
9. **Networking and Communication**:
   Arrays are used to handle data packets or store frames in protocols.

Types of Arrays:

Arrays can be classified based on their dimensions and the way they store data. Below are the main types:

1. One-Dimensional Array (1D Array)

- **Definition**: A one-dimensional array is a linear list of elements, where each element is accessed by a single index.
- **Example**: Storing marks of students in a single subject.
- **Syntax (in C)**:
-

```
Code
data_type array_name[size];
```

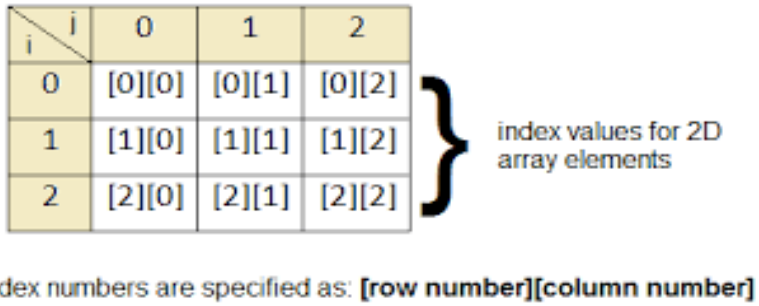
- **Example**:

```
int marks[5] = {90, 85, 78, 92, 88};  // Array of 5 integers
```

2. Two-Dimensional Array (2D Array)

- **Definition**: A two-dimensional array is a collection of elements arranged in rows and columns, like a table or matrix.
- **Example**: Storing data like a chessboard or student marks in multiple subjects.
- **Syntax (in C)**:
  Code
  data_type array_name[rows][columns];

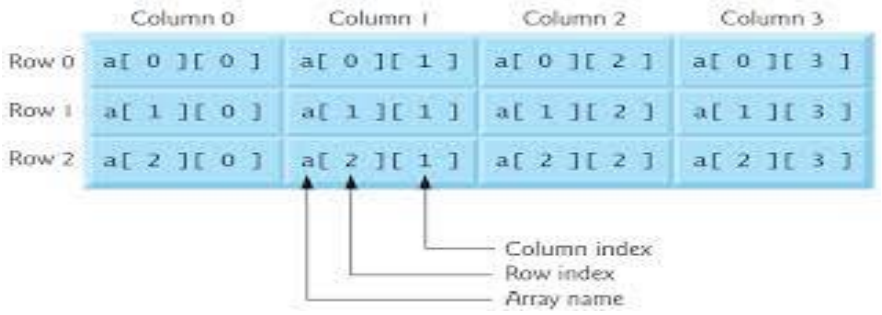
Index in array:



Index numbers are specified as: **[row number][column number]**

- **Example**:
  Code

```
int matrix[3][3] = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};  // 3x3 matrix
```



3. Multidimensional Array

- **Definition**: Arrays with more than three dimensions, though rarely used due to increased complexity. They can represent higher-dimensional data.
- **Example**: Scientific simulations or tensor representations in machine learning.

```
Code
int tensor[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
printf("%d", tensor[1][1][1]); // Outputs 8
```

```
Code
#include <stdio.h>

int main() {
  // Declare a 3D array with 2 layers, 3 rows, and 3 columns
  int arr[2][3][3] = {
    {
      {1, 2, 3},  // First layer, first row
      {4, 5, 6},  // First layer, second row
      {7, 8, 9}   // First layer, third row
    },
    {
      {10, 11, 12}, // Second layer, first row
      {13, 14, 15}, // Second layer, second row
      {16, 17, 18}  // Second layer, third row
    }
  };

  // Output the elements of the 3D array
  printf("Elements of the 3D array:\n");
  for (int i = 0; i < 2; i++) { // Loop over layers (depth)
    for (int j = 0; j < 3; j++) { // Loop over rows
      for (int k = 0; k < 3; k++) { // Loop over columns
        printf("arr[%d][%d][%d] = %d\n", i, j, k, arr[i][j][k]);
      }
    }
  }

  return 0;
}
```

- **Syntax**: Similar to 2D or 3D arrays, but with more dimensions.

4. Dynamic Arrays

- **Definition**: Arrays whose size is determined during runtime, typically using pointers and dynamic memory allocation.
- **Declaration and Example**:

```
int *arr;
int size = 5;
arr = (int *)malloc(size * sizeof(int)); // Allocates memory dynamically
```

5. Character Array (String)

- **Definition**: An array of characters used to store strings.
- **Declaration**:

```
char str[10];
```

- **Example:**

```
char str[] = "Hello";
printf("%s", str); // Outputs Hello
```

6. Array of Pointers

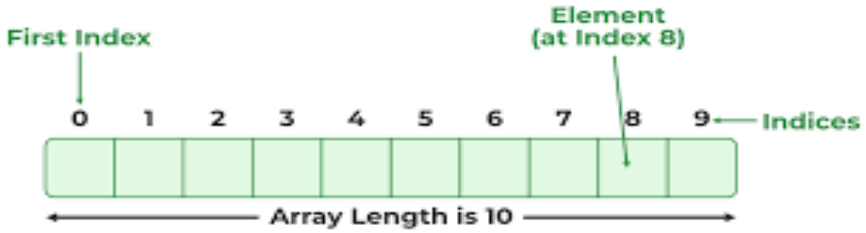
- **Definition**: An array where each element is a pointer, often used to store strings or dynamic data.
- **Declaration and Example**:

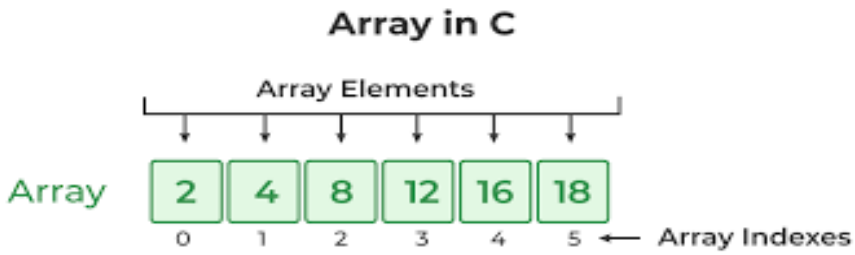
```
char *arr[] = {"apple", "banana", "cherry"};
printf("%s", arr[1]); // Outputs banana
```

Summary Table:

Type	Shape	Example
One-Dimensional Array	Linear	int arr[5] = {1, 2, 3, 4, 5};
Two-Dimensional Array	Table	int matrix[2][2] = {{1,2},{3,4}};
Multi-Dimensional Array	More than 2 indices	int tensor[2][2][2];
Dynamic Array	Varies at runtime	int *arr = malloc(size * sizeof(int));
Character Array	String storage	char str[] = "Hello";
Array of Pointers	Array of addresses	char *arr[] = {"one", "two"};

Array index:





Array Declaration and Initialization:

Arrays in C are declared and initialized to store multiple elements of the same type in a contiguous block of memory. Below are detailed explanations of their **declaration** and **initialization**:

Array Declaration-

- **Syntax**:

```
Code
data_type array_name[size];
```

- **data_type**: Type of elements stored (e.g., int, float, char).
- **array_name**: The name of the array.
- **size**: The total number of elements in the array.

Example:

```
Code
int numbers[5]; // Declares an array named "numbers" that can store 5 integers.
char letters[10]; // Declares an array for 10 characters.
```

Array Initialization-

- Assigning values to the array at the time of declaration or later.

1. Compile-Time Initialization

- Values are assigned during declaration.
- **Syntax**:

```
Code
data_type array_name[size] = {value1, value2, ..., valueN};
```

- **Example**:

```
Code
int numbers[5] = {10, 20, 30, 40, 50};
char vowels[5] = {'a', 'e', 'i', 'o', 'u'};
```

- If fewer values are provided, the remaining elements are set to **0** (for numeric types).

```
Code
int numbers[5] = {10, 20}; // Remaining elements are initialized to 0
```

2. Run-Time Initialization

- Values are assigned dynamically during program execution using loops.
- **Example**:

```
Code
int numbers[5];
for (int i = 0; i < 5; i++) {
   numbers[i] = i * 10; // Assigns values: 0, 10, 20, 30, 40
}
```

3. Omitting the Size

- If the size is omitted during declaration, the compiler determines it based on the number of values.
- **Example**:

```
Code
```

```
int numbers[] = {10, 20, 30, 40, 50}; // Size automatically set to 5.
```

String (Character Array) Initialization-

Special rules apply for character arrays used to store strings.

Example:

```
Code
char str1[] = "Hello"; // Automatically appends '\0' at the end (size = 6).
char str2[6] = "Hello"; // Manually specifying size.
```

Note:

If size is smaller than the string length (excluding \0), a compilation error occurs:

```
Code
char str[4] = "Hello"; // Error: size too small.
```

Partial Initialization-

- You can initialize some elements, leaving others to be **0**.

```
Code
int arr[5] = {1, 2}; // Result: {1, 2, 0, 0, 0}
```

Accessing Array Elements

Accessing array elements requires using their index (0-based).

Example:

```
Code
int arr[3] = {10, 20, 30};
printf("%d", arr[1]); // Outputs 20 (second element).
```

Accessing Elements & Arrays in Memory:

Arrays in C are stored in contiguous blocks of memory, meaning each element is placed sequentially in memory, one after another. The way elements are accessed and represented in memory depends on their type and the number of dimensions. Let's break this down:

1. Accessing Elements in One-dimensional Arrays

- **Memory Layout**: A one-dimensional array's elements are stored sequentially.
- **Syntax**: array[index]
- **Formula**: Address of array[index] = Base Address + (index * Size of each element)
- **Example**:

```
Code
int arr[5] = {10, 20, 30, 40, 50};
printf("%d", arr[2]); // Outputs: 30
```

  - **Memory Calculation**: Suppose arr starts at memory address 0x1000, and each integer is 4 bytes:
    - Address of arr[0]: 0x1000
    - Address of arr[1]: 0x1004
    - Address of arr[2]: 0x1008 (calculated as 0x1000 + (2 * 4)).

2. Accessing Elements in Two-dimensional Arrays

- **Memory Layout**: Stored row-by-row (row-major order) in memory.
- **Syntax**: array[row][column]
- **Formula**: Address of array[row][col] = Base Address + (row * Number of Columns + col) * Size of each element
- **Example**:

```
Code
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
printf("%d", matrix[1][2]); // Outputs: 6
```

  - **Memory Calculation**: Suppose matrix starts at 0x2000, and each integer is 4 bytes:
    - Address of matrix[0][0]: 0x2000

- Address of matrix[1][2]: 0x2000 + (1 * 3 + 2) * 4 = 0x2014

3. Accessing Elements in Multi-dimensional Arrays

- **Memory Layout**: Stored in row-major order for all dimensions.
- **Syntax**: array[d1][d2][d3]
- **Formula**: Address of array[d1][d2][d3] =

Base Address + ((d1 * D2 * D3) + (d2 * D3) + d3) * Size of each element

- **Example**:

```
Code
int cube[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
printf("%d", cube[1][0][1]); // Outputs: 6
```

- **Memory Calculation**: Suppose cube starts at 0x3000, and each integer is 4 bytes:
  - Address of cube[1][0][1]:

0x3000 + ((1 * 2 * 2) + (0 * 2) + 1) * 4 = 0x3014

4. Pointers and Arrays

- **Relationship**: The name of an array acts as a pointer to its first element.
- **Access via Pointer Arithmetic**:

```
Code
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr; // Pointer to the first element
printf("%d", *(ptr + 2)); // Outputs: 30
```

- *(ptr + 2) accesses the value at the memory address (Base Address + 2 * Size).

5. Accessing Elements in Dynamic Arrays

Dynamic arrays are created using memory allocation functions such as malloc or calloc.

- **Example**:

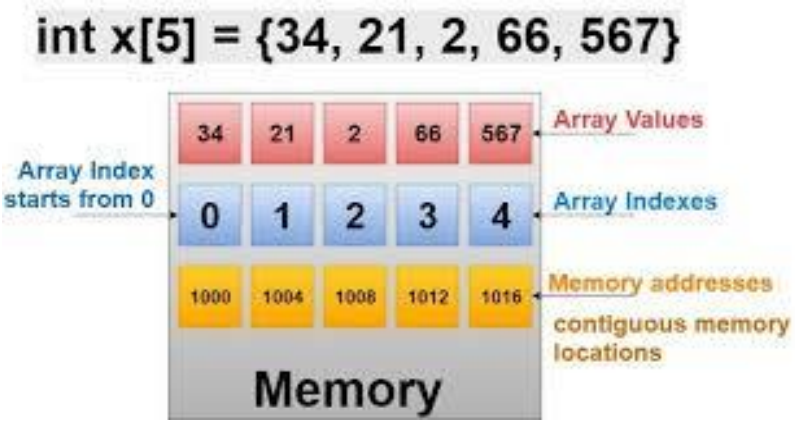
```
Code
int *arr = (int *)malloc(5 * sizeof(int));
for (int i = 0; i < 5; i++) {
   arr[i] = i + 1; // Assign values
}
printf("%d", arr[3]); // Outputs: 4
```

```
free(arr); // Release memory
```

6. Arrays and Memory Alignment

Each element of an array is aligned according to its data type:

- **Integers**: Usually aligned to 4 bytes.
- **Floats/Doubles**: Aligned to 4 or 8 bytes.
- **Characters**: Aligned to 1 byte.



Memory alignment ensures efficient data access by aligning addresses with the system's word size.

Program Using Arrays:

//Write a program determine the maximum number with three number use array.

```
#include <stdio.h>

int main() {
   int n;

   // Input: Number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int arr[n]; // Declare an array of size n

   // Input: Array elements
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      printf("Element %d: ", i + 1);
```

```
    scanf("%d", &arr[i]);
  }

  // Compute the sum of elements
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += arr[i];
  }

  // Calculate the average
  float average = (float)sum / n;

  // Output: Sum and Average
  printf("Sum = %d\n", sum);
  printf("Average = %.2f\n", average);

  return 0;
}
```

Explanation-

1. **Input**:
    a. The program asks the user to enter the number of elements.
    b. The user enters the values for the array using a loop.
2. **Processing**:
    a. A for loop iterates over the array to calculate the sum of its elements.
    b. The average is calculated by dividing the sum by the number of elements. Typecasting to float ensures decimal precision.
3. **Output**:
    a. The program displays the sum and the average of the elements.

Example Run

Input:

```
Enter the number of elements: 5
Enter 5 elements:
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
_____
Output:
Sum = 150
```

```
Average = 30.00
```

Passing arrays to functions:

In C, arrays can be passed to functions to allow modular programming and code reuse. Since arrays are passed by reference, changes made to the array inside the function are reflected in the original array.

Key Points-

1. **Passing an Array**:
    a. When passing an array to a function, only the pointer to the first element is passed, not the entire array.
    b. The function does not receive a copy of the array; it accesses the same memory as the original array.
2. **Syntax**:
    a. When declaring the function, use datatype arrayName[] or datatype *arrayName as the parameter to indicate that the function accepts an array.
3. **Modifications**:
    a. Since arrays are passed by reference, any changes made inside the function affect the original array.

Example: Passing an Array to a Function:

Program to Calculate the Sum of Array Elements-

Code

```
#include <stdio.h>

// Function declaration
int calculateSum(int arr[], int size);

int main() {
  int n;

  // Input: Number of elements
  printf("Enter the number of elements: ");
  scanf("%d", &n);

  int arr[n];
```

```
   // Input: Array elements
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Call the function and display the result
   int sum = calculateSum(arr, n);
   printf("Sum of elements = %d\n", sum);

   return 0;
}

// Function definition
int calculateSum(int arr[], int size) {
   int sum = 0;
   for (int i = 0; i < size; i++) {
      sum += arr[i];
   }
   return sum;
}
```

Explanation

1. **Function Declaration**:

```
Code
int calculateSum(int arr[], int size);
```

   a. int arr[]: Specifies that the function accepts an array of integers.
   b. int size: The size of the array is passed separately because C does not track array sizes.
2. **Passing the Array**:
   a. The call calculateSum(arr, n) passes the starting address of the array (arr) and its size (n) to the function.
3. **Inside the Function**:
   a. The array is accessed using indices (arr[i]).
   b. The function calculates the sum and returns it.
4. **Output**:
   a. The sum is displayed in the main function.

Modifying Arrays in Functions

If the function modifies the array, the changes will persist in the original array. For example:

Code

```
#include <stdio.h>

// Function to double each element
void doubleElements(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      arr[i] *= 2;
   }
}

int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int size = sizeof(arr) / sizeof(arr[0]);

   printf("Original array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");

   // Call the function
   doubleElements(arr, size);

   printf("Modified array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");

   return 0;
}
```

Output:

```
Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10
```

Practice set-7:

1. Write a program to create an array of 10 integers and store multiplication table of 5 in it.
2. Repeat problem 3 for a general input provided by the user using scanf.

3. Write a program containing a function which reverses the array passed to it.
4. Write a program containing functions which counts the number of positive integers in an array.
5. Create an array of size 3 X 10 containing multiplication tables of the numbers 2,7 and 9 respectively.
6. Repeat problem 7 for a custom input given by the user.
7. Create a three-dimensional array and print the address of its elements in increasing order.

Chapter-8: Pointer-

Definition of a Pointer-

A **pointer** in C is a variable that stores the **memory address** of another variable. Instead of holding a value directly, a pointer holds the address where the value is stored. Pointers are powerful because they allow programs to access and manipulate data stored in memory, enabling efficient memory management and dynamic memory allocation.

Key Features of a Pointer:

1. **Address Storage:** A pointer contains the memory address of a variable.
2. **Pointer Type:** The type of a pointer depends on the type of data it points to (e.g., int*, float*, char*).
3. **Dereferencing:** Using the * operator, you can access or modify the value at the memory location the pointer points to.
4. **NULL Pointer:** A special type of pointer that points to nothing and is useful for error handling.
5. **Pointer Arithmetic:** Pointers support arithmetic operations, such as incrementing or decrementing to traverse memory.

Syntax:

```
Code
datatype *pointerName;
```

Here, datatype specifies the type of the variable the pointer will point to, and pointerName is the name of the pointer variable.

Example:

```
#include <stdio.h>
int main() {
   int a = 10;        // Declare an integer variable
   int *ptr;          // Declare a pointer to an integer
   ptr = &a;          // Assign the address of 'a' to the pointer

   printf("Value of a: %d\n", a);        // Print the value of 'a'
   printf("Address of a: %p\n", &a);      // Print the address of 'a'
   printf("Pointer ptr points to: %p\n", ptr);  // Print the address stored in 'ptr'
   printf("Value at address ptr points to: %d\n", *ptr); // Dereference and print value
   return 0;
}
Output:
Value of a: 10
Address of a: 0x7ffeefa4a678
Pointer ptr points to: 0x7ffeefa4a678
Value at address ptr points to: 10
```

Advantages of Pointers-

Pointers offer several advantages in C programming by providing efficient and flexible access to memory. Below are some key advantages:

1. Dynamic Memory Management

- Pointers enable dynamic allocation and deallocation of memory during runtime using functions like malloc(), calloc(), realloc(), and free().
- This helps optimize memory usage and is essential for applications where memory requirements are not fixed.

2. Efficient Array Handling

- Pointers allow direct manipulation of arrays by accessing and iterating through their memory addresses, making array operations more efficient.
- Example:

```
int arr[] = {1, 2, 3};
int *ptr = arr;
for (int i = 0; i < 3; i++) {
```

```
    printf("%d ", *(ptr + i));  // Access array elements using pointers
}
```

3. Faster Execution

- By passing memory addresses instead of copying entire data structures, pointers speed up function calls and reduce memory overhead.

4. Passing Large Data Structures

- Instead of passing large structures or arrays by value, pointers allow passing them by reference, which avoids duplicating data and saves memory.
- Example:

```
void updateValue(int *ptr) {
   *ptr = 100;  // Modify value at the address
}
```

5. Building Complex Data Structures

- Pointers are essential for creating and managing complex data structures like linked lists, trees, graphs, etc.
- Example: A node in a linked list is implemented using pointers:

```
struct Node {
   int data;
   struct Node *next;
};
```

6. Pointer Arithmetic

- Pointers support arithmetic operations (e.g., incrementing, decrementing) to traverse contiguous memory efficiently, such as in arrays or buffers.

7. Hardware-Level Programming

- Pointers allow low-level access to memory and hardware resources, making them useful in embedded systems and operating system programming.

8. Shared Access

- Pointers allow multiple functions or modules to work with the same memory location, facilitating data sharing without duplication.

Pointer declaration & initialization-

In C programming, a pointer must first be declared before it is used. After declaration, it can be initialized by assigning it the memory address of another variable.

1. Pointer Declaration

A pointer is declared using the * operator along with the data type of the variable it will point to.

Syntax:

```
datatype *pointerName;
```

- *datatype: Specifies the type of variable the pointer will point to (e.g., int, float, char).*
- pointerName: The name of the pointer variable.
- *: Indicates that the variable is a pointer.

Example:

```
int *ptr;    // Declares a pointer to an integer
char *cptr;  // Declares a pointer to a character
float *fptr; // Declares a pointer to a floating-point variable
```

2. Pointer Initialization

After declaring a pointer, it can be initialized by assigning it the **address** of a variable using the address-of operator (&).

Syntax:

```
pointerName = &variableName;
```

- &variableName: Fetches the memory address of the variable.
- pointerName: The pointer that will store the memory address.

Example:

```
pointerName = &variableName;
int a = 10;    // Declare an integer variable
int *ptr;      // Declare a pointer to an integer
ptr = &a;      // Initialize the pointer with the address of 'a'
```

Here, ptr now holds the address of a, and *ptr (dereferencing) can access the value of a.

Combined Declaration and Initialization

You can declare and initialize a pointer in a single step.

Syntax:

```
datatype *pointerName = &variableName;
```

Example:

```
int a = 10;
int *ptr = &a;  // Declare and initialize the pointer
```

3. Null Pointer Initialization

Pointers can be initialized to NULL to indicate they do not point to any valid memory location.

Syntax:

```
datatype *pointerName = NULL;
```

Example:

```
int *ptr = NULL;  // Declare a null pointer
```

4. Example Program

```
#include <stdio.h>

int main() {
   int a = 20;       // Declare an integer variable
   int *ptr = &a;    // Declare and initialize a pointer to 'a'

   printf("Value of a: %d\n", a);       // Print value of 'a'
   printf("Address of a: %p\n", &a);    // Print address of 'a'
   printf("Pointer ptr points to: %p\n", ptr);  // Print address stored in 'ptr'
```

```
   printf("Value at ptr: %d\n", *ptr);   // Dereference pointer to get value of 'a'

   return 0;
}
```

Output:

```
Value of a: 20
Address of a: 0x7ffc1234abcd
Pointer ptr points to: 0x7ffc1234abcd
Value at ptr: 20
```

Pointer Expressions-

Pointer expressions involve operations that use pointers to manipulate memory locations or access values stored in memory. These expressions are essential for efficiently handling data structures, arrays, and memory-related tasks.

1. Basic Pointer Expressions

Pointers can be used in expressions to access or manipulate memory and values.

Example:

```
int a = 10;
int *p = &a;    // p points to the address of a
int b = *p + 5;  // Access value of a through p and perform arithmetic
```

2. Pointer Arithmetic

C allows arithmetic operations on pointers, such as incrementing, decrementing, or adding/subtracting integers. These operations consider the size of the data type to which the pointer points.

Key Pointer Arithmetic Operations:

1. **Increment (ptr++)**: Moves the pointer to the next memory location.
2. **Decrement (ptr--)**: Moves the pointer to the previous memory location.
3. **Addition (ptr + n)**: Moves the pointer forward by n locations.
4. **Subtraction (ptr - n)**: Moves the pointer backward by n locations.

Example:

```
int arr[] = {10, 20, 30, 40};
int *p = arr;     // Points to the first element of the array
printf("%d\n", *p);      // 10
p++;            // Points to the next element
printf("%d\n", *p);      // 20
p += 2;          // Points to the fourth element
printf("%d\n", *p);      // 40
```

3. Dereferencing a Pointer

Using the dereference operator (*), a pointer can access or modify the value stored at the memory address it points to.

Example:

```
int a = 10;
int *p = &a;  // p points to the address of a
printf("%d\n", *p);  // Access the value of a through p (output: 10)
*p = 20;          // Modify the value of a through p
printf("%d\n", a);   // Output: 20
```

4. Pointer Comparison

Pointers can be compared using relational operators (==, !=, <, >, etc.) to check their relative positions in memory. Comparisons are meaningful only if the pointers point to elements of the same array.

Example:

```
int arr[] = {10, 20, 30};
int *p1 = &arr[0];
int *p2 = &arr[2];

if (p1 < p2) {
   printf("p1 points to an earlier position.\n");
}
```

5. Expression Involving NULL Pointer

A NULL pointer can be used in expressions to check if a pointer points to a valid memory location.

Example:

```
int *p = NULL;
if (p == NULL) {
   printf("Pointer is NULL.\n");
}
```

6. Complex Pointer Expressions

Pointer expressions can combine arithmetic and dereferencing to access or manipulate array elements, structure members, or dynamically allocated memory.

Example:

```
int arr[] = {10, 20, 30, 40};
int *p = arr;
printf("%d\n", *(p + 2));  // Access the 3rd element (output: 30)
```

7. Expression with Function Pointers

Function pointers allow functions to be stored and called dynamically.

Example:

```
#include <stdio.h>

void printHello() {
   printf("Hello, World!\n");
}

int main() {
   void (*funcPtr)() = printHello;  // Function pointer
   funcPtr();                 // Call function using the pointer
   return 0;
}
```

8. Example Program

```
#include <stdio.h>

int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int *ptr = arr;
```

```
    printf("First element: %d\n", *ptr);    // Dereference pointer
    ptr++;                          // Increment pointer
    printf("Second element: %d\n", *ptr);  // Dereference incremented pointer
    printf("Difference between pointers: %ld\n", ptr - arr); // Pointer subtraction

    return 0;
}
```

Output:

```
First element: 1
Second element: 2
Difference between pointers: 1
```

Practice set-8:

1. Write a program to print the address of a variable. Use this address to get the value of the variable.
2. Write a program having a variable 'i'. Print the address of 'i'. Pass this variable to a function and print its address. Are these addresses same? Why?
3. Write a program to change the value of a variable to ten times of its current value. 4. Write a function and pass the value by reference.
5. Write a program using a function which calculates the sum and average of two numbers. Use pointers and print the values of sum and average in main().
6. Write a program to print the value of a variable i by using "pointer to pointer" type of variable.
7. Try problem 3 using call by value and verify that it does not change the value of the said variable.

Chapter-9: String-

What is a String?

In **C programming**, a **string** is essentially an **array of characters** that ends with a special character called the **null character (\0)**. This null character indicates the end of the string. Strings in C are not a separate data type but are represented as arrays of characters.

Definition:

A **string** is a sequence of characters stored in contiguous memory locations and terminated by the null character (\0).

How Strings Work in C:

- Strings are stored as arrays of characters.
- The last character of the array is always \0.
- Strings can be manipulated using library functions provided in <string.h>.

Declaration of Strings:

1. **Static String Declaration:**

```
char str[20] = "Hello, World!";
```

   a. Here, str is a character array with a maximum capacity of 20 characters.
   b. The string "Hello, World!" occupies 13 characters, including the \0.

2. **Dynamic String Assignment:**

```
char str[] = "Hello!";
```

   a. Here, the array size is automatically determined by the compiler.

3. **Character-by-Character Assignment:**

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Key Properties:

- Strings in C are mutable, meaning you can change their content.
- The sizeof operator calculates the size of the array, including \0.
- Strings can be manipulated using functions like:
  - ♦ strlen() – Gets the string length.
  - ♦ strcpy() – Copies one string to another.
  - ♦ strcat() – Concatenates two strings.
  - ♦ strcmp() – Compares two strings.

Example:

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

int main() {
   char str1[20] = "Hello";
   char str2[] = ", World!";

   strcat(str1, str2);  // Concatenate str2 to str1
```

```
    printf("Concatenated String: %s\n", str1);  // Output: Hello, World!
    printf("String Length: %lu\n", strlen(str1));  // Output: 13

    return 0;
}
```

Declaration & initialization-

In C programming, strings are **arrays of characters** terminated by a null character (\0). Declaring and initializing strings in C is an important concept because strings are not a built-in data type but are represented using character arrays.

1. String Declaration

String declaration reserves space in memory to store a sequence of characters.

Syntax:

```
char string_name[size];
```

Examples:

1. Declaring an empty string:

```
char str[10];  // Reserves space for 10 characters
```

    a. No value is assigned, so the content is uninitialized (garbage value).
2. Declaring without specifying the size:

```
char str[] = "Hello";
```

    a. The size is determined automatically (6 characters, including the \0).

2. String Initialization

String initialization assigns a value to the string during or after its declaration.

Ways to Initialize a String:

1. **During Declaration:**
    a. Using double quotes (preferred):

```
char str[6] = "Hello";
```

    b. Character-by-character:

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

2. **After Declaration:**

```
char str[10];
strcpy(str, "World");  // Requires <string.h>
```

Key Points to Remember:

- The null character (\0) is essential to mark the end of the string.
- If the size of the array is smaller than the string, it will cause memory issues.
- Strings can be initialized with functions like strcpy() for dynamic assignments.

Examples of String Declaration and Initialization:

1. **Static Initialization:**

```
char name[20] = "Alice";
printf("Name: %s\n", name);
```

2. **Dynamic Initialization (Using strcpy()):**

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

int main() {
   char greeting[20];
   strcpy(greeting, "Hello, World!");
```

```
   printf("Greeting: %s\n", greeting);
   return 0;
}
```

3. **Array of Strings:**

```
char names[3][10] = {"Alice", "Bob", "Charlie"};
printf("First Name: %s\n", names[0]);
printf("Second Name: %s\n", names[1]);
```

Declaration vs Initialization of Strings

Feature	Declaration	Initialization
Purpose	Allocates memory for strings.	Assigns a value to the string.
When Performed	Before using the string.	At declaration or later.
Example	char str[10];	char str[] = "Hello";

Taking a String from the User -

In **C programming**, we can take input for a string from the user using various methods. Strings in C are represented as character arrays, and the input functions handle strings carefully to manage the terminating null character (\0).

1. Using scanf() Function

- The scanf() function can be used to read a string, but it stops reading at the first whitespace (space, tab, or newline).
- Syntax:

```
scanf("%s", string_name);
```

- Example:

```
#include <stdio.h>

int main() {
   char str[50];  // Declare a character array
   printf("Enter a string: ");
```

```
    scanf("%s", str);  // Input string (no spaces)
    printf("You entered: %s\n", str);
    return 0;
}
```

Limitations:

- Cannot read strings with spaces (e.g., "Hello World").

2. Using gets() Function

- The gets() function allows input of strings with spaces but is considered unsafe as it can cause buffer overflows.
- Syntax:

```
gets(string_name);
```

- Example:

```
#include <stdio.h>

int main() {
    char str[50];
    printf("Enter a string: ");
    gets(str);  // Input string (supports spaces)
    printf("You entered: %s\n", str);
    return 0;
}
```

Note: The gets() function is deprecated and should be avoided.

3. Using fgets() Function (Recommended)

- The fgets() function is the safest way to take string input as it prevents buffer overflows.
- It reads until a newline or the specified limit is reached.
- Syntax:

```
fgets(string_name, size, stdin);
```

- Example:

```
#include <stdio.h>

int main() {
```

```
    char str[50];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);  // Input string safely
    printf("You entered: %s", str);
    return 0;
}
```

Key Points:

- Includes the newline character \n if the user presses Enter.
- Safer than gets() and handles spaces.

Comparison of Methods

Method	Supports Spaces	Handles Buffer Overflow	Recommended
scanf()	No	No	No
gets()	Yes	No	No (Deprecated)
fgets()	Yes	Yes	Yes

Complete Example Program

```
#include <stdio.h>

int main() {
    char str[50];

    // Taking input using scanf
    printf("Enter a single word (no spaces): ");
    scanf("%s", str);
    printf("Using scanf: %s\n", str);

    // Clearing input buffer
    while (getchar() != '\n');

    // Taking input using fgets
    printf("Enter a sentence (with spaces): ");
    fgets(str, sizeof(str), stdin);
    printf("Using fgets: %s", str);

    return 0;
}
```

Gets and puts function-

In C, the gets() and puts() functions are used for input and output operations, respectively. However, gets() is considered unsafe and is not recommended to be used due to potential security risks. Below is an explanation of both functions:

1. gets()

- **Purpose**: Reads a line of input from the standard input (keyboard) and stores it as a string (a sequence of characters) in a variable.
- **Syntax**:

```
char *gets(char *str);
```

- **Parameters**:
  - str: A pointer to the character array (string) where the input will be stored.
- **Return Value**:
  - Returns the input string on success.
  - Returns NULL on error (e.g., end of file or input error).
- **Warning**: gets() does not perform any bounds checking, which can lead to buffer overflow issues. This is why it has been deprecated in modern C standards (C99 onwards) and is considered unsafe.

Example (Unsafe, avoid usage):

```
#include <stdio.h>
int main() {
   char str[100];
   printf("Enter a string: ");
   gets(str);  // Unsafe, can cause buffer overflow
   printf("You entered: %s\n", str);
   return 0;
}
```

2. puts()

- **Purpose**: Outputs a string to the standard output (usually the screen) followed by a newline.
- **Syntax**:

```
int puts(const char *str);
```

- **Parameters**:
  - str: A pointer to the string to be printed.
- **Return Value**:

  - Returns a non-negative value on success.
  - Returns EOF on error.
- **Behavior**: puts() automatically appends a newline (\n) after printing the string.

Example:

```
#include <stdio.h>
int main() {
   char str[] = "Hello, World!";
   puts(str);  // Prints the string followed by a newline
   return 0;
}
```

Safe Alternatives:

- Instead of gets(), use fgets() for safer input handling:

```
fgets(str, sizeof(str), stdin);
```

fgets() allows you to specify the size of the buffer, preventing buffer overflow.

Summary:

- **gets()**: Unsafe, deprecated, reads a line of input but doesn't check the buffer size.
- **puts()**: Safe, prints a string followed by a newline.

It is highly recommended to use fgets() for input and puts() or printf() for output in modern C programming.

Declaring a sting using pointers-

In C, you can declare a string using pointers in a few different ways. A string in C is essentially an array of characters terminated by a null character (\0), but when using pointers, you're creating a reference to that array. Below are some methods of declaring and using strings with pointers.

1. Declaring a String Using a Pointer (Static Allocation)

You can declare a string by creating a pointer to a character and then assigning it a string literal. This method points to a fixed string literal in memory.

Example:

```
#include <stdio.h>
int main() {
   // Pointer to a string literal
   char *str = "Hello, World!";
```

```
  // Print the string using the pointer
  printf("%s\n", str);

  return 0;
}
```

- **Explanation**: The pointer str is initialized to point to the string literal "Hello, World!". In this case, the string is stored in read-only memory, and you cannot modify the string content using this pointer.

2. Declaring a String Using a Pointer (Dynamic Allocation)

You can also declare a string using a pointer and allocate memory dynamically for the string. This allows you to modify the contents of the string.

Example:

```
#include <stdio.h>
#include <stdlib.h> // For malloc()

int main() {
  // Dynamically allocate memory for a string of 20 characters
  char *str = (char *)malloc(20 * sizeof(char));

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

  // Assign a string to the allocated memory
  strcpy(str, "Hello, World!");

  // Print the string using the pointer
  printf("%s\n", str);

  // Free the allocated memory
  free(str);

  return 0;
}
```

- **Explanation**:
  - malloc() allocates memory for 20 characters.

- strcpy() copies the string "Hello, World!" into the dynamically allocated memory.
- After use, it's essential to call free() to release the memory allocated by malloc().

3. Declaring a String Using a Pointer to a Character Array

Another way to declare a string using pointers is to create a pointer to an array of characters. This is similar to declaring a string in the form of a character array, but with a pointer.

Example:

```
#include <stdio.h>

int main() {
   // Pointer to an array of characters (string)
   char *str = "Hello, World!";

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

Summary:

- **Static String Declaration**: You can use a pointer to point to a string literal, e.g., char *str = "Hello";.
- **Dynamic Memory Allocation**: You can allocate memory dynamically using malloc() for strings, e.g., char *str = malloc(size * sizeof(char));.
- **Pointer to Array**: A pointer can also be used to refer to an array of characters, such as char str[] = "Hello";.

When using strings with pointers, remember that:

- **String literals** are immutable (cannot be changed) when pointed to by a pointer.
- **Dynamic memory allocation** allows modification of the string, but you must handle memory management (allocation and freeing) carefully.

String manipulation-

String Manipulation in C

In C, strings are essentially arrays of characters terminated by a null character ('\0'). C does not have a built-in string type like higher-level languages, so string manipulation involves working with character arrays or pointers to characters.

Here are some key points and examples of string manipulation in C:

1. Basic String Declaration

In C, a string is simply an array of characters. It is always null-terminated, meaning it ends with a special character '\0' that indicates the end of the string.

```
char str[] = "Hello, World!";  // Declaring a string
```

Here, str holds an array of characters and the compiler automatically adds the '\0' at the end.

2. String Input and Output

Using scanf() to input a string:

scanf() can be used to take input from the user.

```
#include <stdio.h>

int main() {
   char str[100];  // Declare a character array to store the string
   printf("Enter a string: ");
   scanf("%99[^\n]", str);  // Read input until a newline
   printf("You entered: %s\n", str);  // Output the string
   return 0;
}
```

- The format "%99[^\n]" allows us to read the entire line, including spaces, until a newline is encountered.

Using gets() (Not recommended):

Although gets() can be used to read strings, it is **unsafe** and deprecated because it does not check the size of the input, which can lead to buffer overflow issues.

3. String Functions in C

The C Standard Library (<string.h>) provides several functions for string manipulation. Some commonly used string functions are:

a. strlen(): Returns the length of the string (excluding the null-terminator).

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

int main() {
   char str[] = "Hello";
   printf("Length of string: %lu\n", strlen(str));  // Output: 5
   return 0;
```

```
}
```

b. strcpy(): Copies a string to another.

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

int main() {
   char src[] = "Hello";
   char dest[20];
   strcpy(dest, src);  // Copy content of src into dest
   printf("Copied string: %s\n", dest);  // Output: Hello
   return 0;
}
```

c. strcat(): Concatenates (joins) two strings.

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

int main() {
   char str1[20] = "Hello";
   char str2[] = " World!";
   strcat(str1, str2);  // Concatenate str2 to str1
   printf("Concatenated string: %s\n", str1);  // Output: Hello
World!
   return 0;
}
```

d. strcmp(): Compares two strings.

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

int main() {
   char str1[] = "Apple";
   char str2[] = "Banana";
   int result = strcmp(str1, str2);  // Compare str1 and str2
   if (result < 0)
      printf("str1 is less than str2\n");
```

```
    else if (result > 0)
        printf("str1 is greater than str2\n");
    else
        printf("Both strings are equal\n");
    return 0;
}
```

e. strchr(): Searches for the first occurrence of a character in a string.

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

int main() {
    char str[] = "Hello, World!";
    char *ptr = strchr(str, 'o');  // Find first occurrence of 'o'
    if (ptr != NULL)
        printf("Found 'o' at position: %ld\n", ptr - str);  // Output the position
    else
        printf("'o' not found\n");
    return 0;
}
```

f. strstr(): Finds the first occurrence of a substring within a string.

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

int main() {
    char str[] = "Hello, World!";
    char *substr = strstr(str, "World");  // Search for the substring "World"
    if (substr != NULL)
        printf("Substring found: %s\n", substr);  // Output: World!
    else
        printf("Substring not found\n");
    return 0;
}
```

g. sprintf(): Formats and stores a string in a variable.

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

```
int main() {
   char buffer[100];
   int num = 123;
   sprintf(buffer, "The number is %d", num);  // Store formatted string in
buffer
   printf("%s\n", buffer);  // Output: The number is 123
   return 0;
}
```

4. String Manipulation without Standard Functions

You can also manipulate strings manually by iterating over the characters of a string.

Example: Reversing a String

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

void reverseString(char str[]) {
   int start = 0;
   int end = strlen(str) - 1;
   while (start < end) {
      // Swap characters at start and end positions
      char temp = str[start];
      str[start] = str[end];
      str[end] = temp;

      start++;
      end--;
   }
}

int main() {
   char str[] = "Hello";
   reverseString(str);
   printf("Reversed string: %s\n", str);  // Output: olleH
   return 0;
}
```

5. Memory Management with Strings

When working with strings, especially when dynamically allocating memory, it's important to free any allocated memory to prevent memory leaks.

Example: Dynamically Allocating and Freeing Memory

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

int main() {
   char *str = (char *)malloc(20 * sizeof(char));  // Dynamically allocate memory for
a string
   if (str == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }
   strcpy(str, "Dynamic String");
   printf("Allocated string: %s\n", str);  // Output: Dynamic String

   free(str);  // Free the allocated memory
   return 0;
}
```

Conclusion

String manipulation in C requires an understanding of pointers, character arrays, and the standard library functions available in <string.h>. Common tasks include:

- **Input/Output** using functions like scanf(), printf(), and gets() (though unsafe).
- **String operations** like copying, concatenating, comparing, and searching.
- **Memory management** when uing dynamic memory allocation (malloc(), free()).

Carefully managing memory and being aware of string bounds is essential for safe and efficient string manipulation in C.

Standard library functions for strings-

Standard Library Functions for Strings in C

In C, strings are represented as arrays of characters, and the C Standard Library provides several functions in the <string.h> header to manipulate strings. These functions allow operations such as copying, concatenating, comparing, and searching within strings. Below is an overview of the most commonly used string functions in C.

1. strlen() - String Length

Purpose: Returns the length of the string, excluding the null terminator (\0).

Syntax:

size_t strlen(const char *str);

- **Parameters**:
  - str: The string whose length is to be calculated.
- **Return Value**:
  - The number of characters in the string (excluding the null terminator).

Example:

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

int main() {
   char str[] = "Hello";
   printf("Length of the string: %lu\n", strlen(str));  // Output: 5
   return 0;
}
```

2. strcpy() - String Copy

Purpose: Copies the content of one string to another.

Syntax:

char *strcpy(char *dest, const char *src);

- **Parameters**:
  - dest: The destination string where src will be copied.
  - src: The source string to be copied.
- **Return Value**:
  - Returns the destination string (dest).

Example:

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

int main() {
   char src[] = "Hello, World!";
   char dest[50];
   strcpy(dest, src);
   printf("Destination string: %s\n", dest);  // Output: Hello, World!
   return 0;
}
```

3. strncpy() - String Copy (with length)

Purpose: Copies a specified number of characters from one string to another. If the length of the source string is less than the specified number, it will pad the destination with \0.

Syntax:

char *strncpy(char *dest, const char *src, size_t n);

- **Parameters**:
  - dest: The destination string where src will be copied.
  - src: The source string to be copied.
  - n: The maximum number of characters to copy.
- **Return Value**:
  - Returns the destination string (dest).

Example:

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

int main() {
   char src[] = "Hello";
   char dest[10];
   strncpy(dest, src, 3);  // Copy first 3 characters
   dest[3] = '\0';  // Manually null-terminate the string
   printf("Destination string: %s\n", dest);  // Output: Hel
   return 0;
}
```

4. strcat() - String Concatenation

Purpose: Concatenates (joins) two strings, appending the second string (src) to the first string (dest).

Syntax:

char *strcat(char *dest, const char *src);

- **Parameters**:
  - dest: The destination string to which src will be appended.
  - src: The source string to be appended to dest.
- **Return Value**:
  - Returns the destination string (dest).

Example:

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

int main() {
   char str1[50] = "Hello";
   char str2[] = " World!";
```

```
    strcat(str1, str2);  // Append str2 to str1
    printf("Concatenated string: %s\n", str1);  // Output: Hello World!
    return 0;
}
```

---

5. strncat() - String Concatenation (with length)

Purpose: Concatenates a specified number of characters from one string to another.

Syntax:

char *strncat(char *dest, const char *src, size_t n);

- **Parameters**:
  - dest: The destination string to which src will be appended.
  - src: The source string to be appended to dest.
  - n: The maximum number of characters to append from src.
- **Return Value**:
  - Returns the destination string (dest).

Example:

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

int main() {
    char str1[50] = "Hello";
    char str2[] = " World!";
    strncat(str1, str2, 3);  // Append first 3 characters of str2 to str1
    printf("Concatenated string: %s\n", str1);  // Output: Hello Wor
    return 0;
}
```

6. strcmp() - String Comparison

Purpose: Compares two strings lexicographically. It returns an integer that indicates the relationship between the strings.

Syntax:

int strcmp(const char *str1, const char *str2);

- **Parameters**:
  - str1: The first string to compare.
  - str2: The second string to compare.
- **Return Value**:
  - A value less than 0 if str1 is lexicographically less than str2.
  - 0 if str1 is equal to str2.
  - A value greater than 0 if str1 is lexicographically greater than str2.

Example:

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

int main() {
   char str1[] = "Apple";
   char str2[] = "Banana";
   int result = strcmp(str1, str2);

   if (result < 0)
      printf("'%s' is less than '%s'\n", str1, str2);
   else if (result > 0)
      printf("'%s' is greater than '%s'\n", str1, str2);
   else
      printf("'%s' is equal to '%s'\n", str1, str2);
   return 0;
}
```

7. strncmp() - String Comparison (with length)

Purpose: Compares the first n characters of two strings lexicographically.

Syntax:

int strncmp(const char *str1, const char *str2, size_t n);

- **Parameters**:
  - str1: The first string to compare.
  - str2: The second string to compare.
  - n: The number of characters to compare.
- **Return Value**: Same as strcmp().

Example:

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

int main() {
   char str1[] = "Apple";
   char str2[] = "Apples";
   int result = strncmp(str1, str2, 5);  // Compare first 5 characters

   if (result == 0)
      printf("First 5 characters are equal\n");
```

```
  else
    printf("First 5 characters are not equal\n");

  return 0;
}
```

---

8. strchr() - Search for a Character

Purpose: Finds the first occurrence of a character in a string.

Syntax:

char *strchr(const char *str, int c);

- **Parameters**:
  - str: The string to search within.
  - c: The character to find.
- **Return Value**:
  - A pointer to the first occurrence of the character c in str.
  - If the character is not found, returns NULL.

Example:

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

int main() {
  char str[] = "Hello, World!";
  char *ptr = strchr(str, 'o');  // Find first occurrence of 'o'
  if (ptr != NULL)
    printf("Found 'o' at position: %ld\n", ptr - str);  // Output: 4
  return 0;
}
```

9. strstr() - Search for a Substring

Purpose: Finds the first occurrence of a substring in a string.

Syntax:

char *strstr(const char *haystack, const char *needle);

- **Parameters**:
  - haystack: The string to search within.
  - needle: The substring to find.
- **Return Value**:
  - A pointer to the first occurrence of the substring needle in haystack.
  - If the substring is not found, returns NULL.

Example:

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

int main() {
   char str[] = "Hello, World!";
   char *substr = strstr(str, "World");  // Find substring "World"
   if (substr != NULL)
      printf("Found substring: %s\n", substr);  // Output: World!
   return 0;
}
```

---

10. strtok() - Tokenize a String

Purpose: Breaks a string into tokens based on delimiters (usually spaces, commas, etc.).

Syntax:

char *strtok(char *str, const char *delim);

- **Parameters**:
 - `str`: The string to be tokenized (on the first call).
 - `delim`: The delimiter characters used to split the string.
- **Return Value**:
 - A pointer to the first token found. On subsequent calls, it returns the next token.

Example:

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

int main() {
   char str[] = "Hello, World, C, Programming";
   char *token = strtok(str, ", ");  // Tokenize by space and
comma
   while (token != NULL) {
      printf("Token: %s\n", token);
      token = strtok(NULL, ", ");
   }
   return 0;
}
```

Conclusion

These string functions in C provide a variety of utilities to perform essential string operations such as copying, concatenating, comparing, searching, and tokenizing strings. Most of these functions are part of the C Standard Library and are defined in the <string.h> header file. Mastering these functions is key to effective string manipulation in C.

Practice set-9-

1. Which of the following is used to appropriately read a multi-word string.

- ✓ gets()
- ✓ puts()
- ✓ printf()
- ✓ scanf()

2. Write a program to take string as an input from the user using %c and %s confirm that the strings are equal.

3. Write your own version of strlen function from <string.h>

4. Write a function slice() to slice a string. It should change the original string such that it is now the sliced string. Take 'm' and 'n' as the start and ending position for slice.

5. Write your own version of strcpy function from <string.h>

6. Write a program to encrypt a string by adding 1 to the ascii value of its characters.

7. Write a program to decrypt the string encrypted using encrypt function in problem 6.

8. Write a program to count the occurrence of a given character in a string.

9. Write a program to check whether a given character is present in a string or not.

Chapter-10: Structures-

What is Structure in c language?

In C programming, a **structure** is a user-defined data type that allows you to group different types of variables under a single name. These variables, called **members** or **fields**, can be of different data types. Structures are useful when you need to represent

an entity that has several properties or characteristics, but these properties may differ in type (e.g., an employee has a name, age, and salary).

Syntax of a Structure

The syntax for defining a structure is:

```
struct StructureName {
   dataType member1;
   dataType member2;
   // more members
};
```

Example

Here's an example of a structure in C:

```
#include <stdio.h>

// Define a structure for an Employee
struct Employee {
   char name[50];
   int age;
   float salary;
};

int main() {
   // Declare a structure variable
   struct Employee emp1;

   // Assign values to the members
   strcpy(emp1.name, "John Doe");
   emp1.age = 30;
   emp1.salary = 55000.50;

   // Access and print the structure members
   printf("Name: %s\n", emp1.name);
   printf("Age: %d\n", emp1.age);
   printf("Salary: %.2f\n", emp1.salary);

   return 0;
}
```

Explanation:

1. **Defining the structure**:
    - struct Employee defines a structure named Employee.
    - The structure has three members: name (a character array), age (an integer), and salary (a float).
2. **Using the structure**:
    - A variable emp1 of type struct Employee is declared.
    - The name member is assigned using strcpy (since name is a string).
    - Other members (age and salary) are assigned directly.
3. **Accessing members**:
    - Structure members are accessed using the dot (.) operator.

Structure Benefits

- **Encapsulation**: It allows you to bundle related data together, making your code more organized.
- **Data Integrity**: Since a structure can contain multiple types of data, it helps in representing real-world entities in a program.

Nested Structures

Structures can also contain other structures as members, known as **nested structures**.

```
struct Address {
   char city[50];
   char state[50];
   int zip;
};

struct Person {
   char name[50];
   int age;
   struct Address address; // Nested structure
};
```

Structure Size

The size of a structure depends on the sum of the sizes of its members, but there may also be padding for alignment reasons (to make the structure's memory access more efficient).

Conclusion

A **structure** in C provides a way to store multiple variables of different data types together, making it useful for representing complex data. It is a fundamental feature for building larger and more maintainable programs.

Declaration and initialization-

In C, structures are declared to define a template for data that can hold multiple members of different types. After declaring a structure, you can initialize it either at the time of declaration or later when creating an instance of the structure.

1. Structure Declaration

To declare a structure, you use the struct keyword, followed by the structure name and the members within curly braces {}. Here is the syntax for declaring a structure:

```
struct StructureName {
   dataType member1;
   dataType member2;
   // more members
};
```

2. Structure Initialization

There are two common ways to initialize a structure in C:

1. **At the time of declaration**
2. **After declaration**

1. Declaration with Initialization

You can initialize a structure at the time of declaring an instance. This method is often used to provide initial values for each member of the structure.

Example:

```
#include <stdio.h>

struct Employee {
   char name[50];
   int age;
   float salary;
};

int main() {
   // Initialize the structure at the time of declaration
   struct Employee emp1 = {"John Doe", 30, 50000.75};

   // Print the structure members
   printf("Name: %s\n", emp1.name);
   printf("Age: %d\n", emp1.age);
   printf("Salary: %.2f\n", emp1.salary);

   return 0;
```

```
}
```

Explanation:

- emp1 is a structure of type Employee, initialized with:
    - name as "John Doe"
    - age as 30
    - salary as 50000.75

The structure members are initialized in the same order they are declared.

2. Initialization After Declaration

You can also declare a structure first and then initialize its members separately. This is often done in the program when you want to initialize the structure later.

Example:

```
#include <stdio.h>

struct Employee {
   char name[50];
   int age;
   float salary;
};

int main() {
   // Declare the structure variable
   struct Employee emp1;

   // Initialize the members individually
   strcpy(emp1.name, "Alice Johnson");
   emp1.age = 25;
   emp1.salary = 60000.50;

   // Print the structure members
   printf("Name: %s\n", emp1.name);
   printf("Age: %d\n", emp1.age);
   printf("Salary: %.2f\n", emp1.salary);

   return 0;
}
```

Explanation:

- First, the structure emp1 is declared.
- Then, each member of the structure is initialized separately:

- name is initialized using strcpy (since name is a string).
- age and salary are initialized directly.

3. Designated Initialization (C99 and later)

In C99 and later standards, you can initialize structure members using designated initializers. This allows you to initialize specific members of the structure in any order.

Example:

```
#include <stdio.h>

struct Employee {
   char name[50];
   int age;
   float salary;
};

int main() {
   // Designated initialization (C99 and later)
   struct Employee emp1 = {.salary = 45000.50, .name = "Bob Smith", .age = 40};

   // Print the structure members
   printf("Name: %s\n", emp1.name);
   printf("Age: %d\n", emp1.age);
   printf("Salary: %.2f\n", emp1.salary);

   return 0;
}
```

Explanation:

- In this example, members of emp1 are initialized using the member names directly.
- This method is especially useful when initializing large structures or when you don't want to maintain the order of the members.

4. Initializing Arrays in Structures

When a structure contains an array, you can also initialize it in a similar way.

Example:

```
#include <stdio.h>

struct Student {
   char name[50];
```

```
    int marks[5];  // Array of 5 integers
};

int main() {
    // Declare and initialize the structure
    struct Student student1 = {"John", {85, 90, 88, 92, 87}};

    // Print the structure members
    printf("Name: %s\n", student1.name);
    printf("Marks: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", student1.marks[i]);
    }
    printf("\n");

    return 0;
}
```

Explanation:

- The marks array inside the structure is initialized with {85, 90, 88, 92, 87}.
- The name is initialized with "John".

Summary of Structure Initialization:

1. **At the time of declaration**:
    - You can initialize the members in the order of their declaration in the structure.
2. **After declaration**:
    - You can assign values to the structure members individually.
3. **Designated Initialization (C99)**:
    - Allows you to initialize specific members of a structure in any order using the member names.

By using structures, you can group different types of data together, making your programs more organized and manageable.

Structure in memory-

In C programming, a **structure** is a collection of variables, possibly of different types, grouped together under a single name. The way structures are stored in memory depends on various factors such as the size of the data members and padding for memory alignment. Here's a detailed explanation of how structures are stored in memory:

1. Memory Layout of Structures

When a structure is declared in C, the compiler allocates memory for its members. The memory layout is sequential, meaning the members are laid out in the order they are defined in the structure. However, the compiler may introduce padding between members to satisfy the machine's alignment constraints.

Example:

```
#include <stdio.h>

struct MyStruct {
   char a;   // 1 byte
   int b;    // 4 bytes
   char c;   // 1 byte
};

int main() {
   struct MyStruct s;
   printf("Size of struct: %zu\n", sizeof(s));
   return 0;
}
```

Memory Layout of MyStruct:

- a: 1 byte (char)
- b: 4 bytes (int)
- c: 1 byte (char)

However, the total size of the structure is **12 bytes**, not 6 bytes, due to **padding**. The reason for this is **alignment**: modern computers typically access memory in chunks (e.g., 4-byte chunks for int values). To ensure efficient memory access, the compiler adds padding after a and c to align b to a memory address that's divisible by 4.

So, the memory layout would look something like this:

| a (1 byte) | padding (3 bytes) | b (4 bytes) | c (1 byte) | padding (3 bytes) |

Why Padding is Used:

- **Alignment Constraints**: Some data types have alignment restrictions. For example, an int might require 4-byte alignment, meaning it needs to start at an address that is a multiple of 4.
- **Performance**: Misaligned data accesses can result in slower memory operations on some architectures, as the CPU may need to perform additional operations to access misaligned data.

2. Size of a Structure

The size of a structure is influenced by:

1. **The size of the individual members**: The size of the structure is at least the sum of the sizes of its members.
2. **Padding and alignment**: The compiler may add padding between structure members or at the end of the structure to ensure that each member is properly aligned. As a result, the size of a structure may be larger than the sum of its member sizes.

Example (calculating structure size):

```
#include <stdio.h>

struct Example {
   char a;   // 1 byte
   int b;    // 4 bytes
   char c;   // 1 byte
};

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

Expected Output:

Size of structure: 12 bytes

Here, even though the sum of the sizes of individual members (1 byte + 4 bytes + 1 byte = 6 bytes), the structure's total size is **12 bytes** because of padding for alignment.

3. Accessing Structure Members in Memory

Each structure member can be accessed using the **dot (.) operator**, and the compiler calculates the correct memory offset for each member. When accessing a structure member, the memory location is computed based on the offsets of each member relative to the beginning of the structure.

Example:

```
#include <stdio.h>

struct Student {
   int rollNumber;  // 4 bytes
   char grade;     // 1 byte
   float marks;    // 4 bytes
```

```
};

int main() {
   struct Student student1 = {101, 'A', 85.5};

   printf("Memory address of rollNumber: %p\n", (void*)&student1.rollNumber);
   printf("Memory address of grade: %p\n", (void*)&student1.grade);
   printf("Memory address of marks: %p\n", (void*)&student1.marks);

   return 0;
}
```

Output Example (addresses will vary):
Memory address of rollNumber: 0x7ffeea11e9fc
Memory address of grade: 0x7ffeea11e9f0
Memory address of marks: 0x7ffeea11e9f4
The addresses show how the members of the structure are stored in memory. The members are usually stored in the order they are defined, but the exact memory locations may vary depending on padding and alignment.

4. Structure Padding and Alignment Rules

- **Alignment of Members**: The alignment of a structure member is typically determined by the largest alignment requirement among its members. For example, if a structure has an int (4-byte alignment) and a char (1-byte alignment), the structure as a whole may be aligned on a 4-byte boundary.
- **Padding Between Members**: To satisfy alignment, padding may be inserted between structure members. The compiler tries to ensure that each member is aligned to its natural boundary.
- **Padding After the Structure**: The compiler may also add padding at the end of the structure to ensure that arrays of structures are aligned properly. For example, if a structure has a size of 10 bytes and needs to be followed by another structure, the compiler might add 2 bytes of padding to align the next structure correctly in memory.

5. Structure and Memory Access
Memory access to structures can vary in performance depending on how they are aligned. Misaligned access can cause inefficiencies on some architectures, but modern compilers typically handle structure alignment to optimize memory access speed. Here's how different members can affect memory access:

- **Proper Alignment**: Members aligned on their natural boundaries (e.g., 4-byte int at an address divisible by 4) typically result in faster access.

- **Misalignment**: Accessing members that are not aligned properly may incur performance penalties, especially on certain architectures like ARM or older processors.

6. Structure Pointers and Memory

When working with pointers to structures, the structure is not directly copied into memory but rather accessed through the pointer.

Example:

```
#include <stdio.h>

struct Student {
   int rollNumber;
   char grade;
   float marks;
};

int main() {
   struct Student student1 = {101, 'A', 85.5};
   struct Student *ptr = &student1;

   // Access structure members using pointer
   printf("Roll number: %d\n", ptr->rollNumber);
   printf("Grade: %c\n", ptr->grade);
   printf("Marks: %.2f\n", ptr->marks);

   return 0;
}
```

Here, ptr is a pointer to the structure student1. The -> operator is used to access members of the structure through the pointer.

Conclusion

- **Structure Memory Layout**: The memory layout of a structure is influenced by member sizes, padding, and alignment constraints.
- **Padding**: Compilers insert padding between members or at the end of the structure to ensure proper memory alignment and efficient access.
- **Size of Structure**: The size of a structure can be larger than the sum of the sizes of its individual members due to padding and alignment.

When working with structures, especially large ones, understanding their memory layout and alignment is important for optimizing both memory usage and performance.

Pointer to Structure-

In C programming, a **pointer to a structure** is a pointer that points to the memory location where a structure is stored. Just like pointers to other data types, a pointer to a structure holds the address of the structure's variable, allowing you to access and manipulate the structure's members indirectly.

Why Use a Pointer to a Structure?

- **Dynamic Memory Allocation**: You can dynamically allocate memory for a structure using pointers, which is useful for situations where the number of structures is not known beforehand (e.g., in linked lists, trees, etc.).
- **Efficiency**: Passing a pointer to a structure is more efficient than passing the entire structure by value, especially when the structure is large.

Declaring a Pointer to a Structure

A pointer to a structure is declared using the struct keyword, followed by the pointer variable name with an asterisk (*).

Syntax:

```
struct StructureName *pointerName;
```

Example:

```
#include <stdio.h>

// Define a structure
struct Student {
   char name[50];
   int age;
   float marks;
};

int main() {
   // Declare a structure variable
   struct Student student1 = {"John Doe", 20, 90.5};

   // Declare a pointer to the structure
   struct Student *ptr;

   // Assign the address of student1 to ptr
   ptr = &student1;
```

```
    // Access structure members using the pointer
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Marks: %.2f\n", ptr->marks);

    return 0;
}
```

Explanation:

1. **Declaring the structure**: We define a Student structure with three members: name, age, and marks.
2. **Declaring the pointer**: We declare a pointer ptr of type struct Student *.
3. **Assigning address**: We assign the address of student1 to ptr using the & operator.
4. **Accessing structure members**: We use the -> operator to access the structure members via the pointer ptr.

Pointer to Structure and the -> Operator

When using a pointer to access a structure's members, we use the -> operator, which is shorthand for dereferencing the pointer and then accessing the member. This is equivalent to (*ptr).member but is more commonly used.

Example of -> Operator:

```
ptr->age = 21;  // Equivalent to (*ptr).age = 21;
```

Pointer to Structure in Function Calls

Pointers to structures are often used when passing structures to functions, especially when large structures need to be passed, or you want to modify the structure in the function.

Example:

```
#include <stdio.h>

struct Student {
    char name[50];
    int age;
    float marks;
};

// Function to modify structure using a pointer
void modifyStudent(struct Student *ptr) {
    ptr->age = 22; // Modify the age of the student
    ptr->marks = 95.5; // Modify the marks of the student
```

```
}

int main() {
   struct Student student1 = {"Alice", 20, 85.0};

   // Call the function with a pointer to the structure
   modifyStudent(&student1);

   // Print modified values
   printf("Name: %s\n", student1.name);
   printf("Age: %d\n", student1.age);
   printf("Marks: %.2f\n", student1.marks);

   return 0;
}
```

Explanation:

- **Function modifyStudent**: This function accepts a pointer to a structure (struct Student *ptr). Inside the function, the structure members are modified via the pointer.
- **Passing the structure**: We pass the address of student1 to the function using the & operator.

Dynamic Memory Allocation with Pointers to Structures

You can dynamically allocate memory for structures using malloc() or calloc() functions. This is particularly useful when the number of structures is not known at compile time.

Example:

```
#include <stdio.h>
#include <stdlib.h>  // For malloc()

struct Student {
   char name[50];
   int age;
   float marks;
};

int main() {
   // Dynamically allocate memory for one Student structure
   struct Student *ptr = (struct Student *)malloc(sizeof(struct Student));
```

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

    // Assign values to the dynamically allocated structure
    strcpy(ptr->name, "Bob");
    ptr->age = 21;
    ptr->marks = 90.5;

    // Access and print the structure members
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Marks: %.2f\n", ptr->marks);

    // Free the dynamically allocated memory
    free(ptr);

    return 0;
}
```

Explanation:

1. **Dynamic Memory Allocation**: We use malloc() to dynamically allocate memory for a Student structure. The size of memory allocated is the size of the structure.
2. **Assigning values**: The members of the structure are accessed using the pointer ptr and the -> operator.
3. **Freeing memory**: After using the dynamically allocated memory, we call free() to release it back to the system.

Important Considerations When Using Pointers to Structures

1. **Dereferencing**: Always ensure that the pointer points to valid memory before dereferencing it. Dereferencing a NULL pointer or an uninitialized pointer can lead to undefined behavior (crashes, memory corruption, etc.).
2. **Memory Allocation**: When dynamically allocating memory for a structure (or an array of structures), always check if the memory allocation is successful by verifying if the pointer is NULL.
3. **Pointer Arithmetic**: You can use pointer arithmetic with pointers to structures, but it's generally less common than accessing structure members via the -> operator.

Summary of Key Points

- A **pointer to a structure** holds the memory address of a structure and allows you to access its members indirectly.
- To access structure members through a pointer, use the -> operator (shorthand for dereferencing the pointer and accessing the member).
- **Pointers to structures** are useful when passing large structures to functions or when dynamically allocating memory for structures.
- When using dynamic memory allocation (malloc or calloc), make sure to check if the memory allocation was successful and always free the memory after use.

Using pointers to structures is a common technique in C programming that helps improve the efficiency of memory management and function calls.

Arrow Operator-

In the C programming language, the **arrow operator (->)** is used to access members of a structure (or union) via a pointer to the structure. It is particularly useful when dealing with dynamically allocated structures or structures passed as pointers.

Syntax:

```
pointer_to_structure->member
This is equivalent to:
(*pointer_to_structure).member
```

Explanation:

- **pointer_to_structure**: A pointer to a structure.
- **member**: A field (or member) within the structure.

The arrow operator simplifies accessing members without the need for explicit dereferencing.

Example:

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

struct Point {
   int x;
   int y;
};

int main() {
   struct Point *p = (struct Point *)malloc(sizeof(struct Point)); //
Allocate memory for structure
```

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

    // Access structure members using the arrow operator
    p->x = 10;
    p->y = 20;

    printf("Point coordinates: x = %d, y = %d\n", p->x, p->y);

    free(p); // Free the allocated memory
    return 0;
}
```

Output:

```
Point coordinates: x = 10, y = 20
```

Key Points:

1. **Usage Context**: The arrow operator is used when dealing with **pointers to structures**.
2. **Simplifies Syntax**: Without the arrow operator, you'd need to write (*p).x instead of p->x.
3. **Common in Dynamic Memory**: Frequently used when working with dynamically allocated structures or passing structures as arguments by reference.
4. **Error Prevention**: Using the arrow operator reduces the chances of errors in parentheses placement compared to (*pointer).member.

The arrow operator is fundamental in C when working with complex data structures like linked lists, trees, and other pointer-heavy constructs.

Passing structure to a function-

In C, structures can be passed to functions in three main ways:

1. Pass by Value

- A copy of the structure is passed to the function.
- Any changes made to the structure within the function do not affect the original structure.

Syntax Example:

```
#include <stdio.h>

struct Point {
```

```
    int x;
    int y;
};

void printPoint(struct Point p) {
    printf("x = %d, y = %d\n", p.x, p.y);
}

int main() {
    struct Point p1 = {10, 20};
    printPoint(p1); // Passing structure by value
    return 0;
}
```

Output:

x = 10, y = 20

Key Notes:

- Safe, as changes inside the function won't affect the original structure.
- Can be inefficient for large structures due to copying overhead.

2. Pass by Reference (Using Pointers)

- A pointer to the structure is passed to the function.
- This allows the function to directly modify the original structure.

Syntax Example:

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void updatePoint(struct Point *p) {
    p->x = 100;  // Modifying original structure
    p->y = 200;
}

int main() {
    struct Point p1 = {10, 20};
    updatePoint(&p1); // Passing structure by reference
```

```
    printf("x = %d, y = %d\n", p1.x, p1.y);
    return 0;
}
```

Output:

x = 100, y = 200

Key Notes:

- Efficient, as no copying of the structure is needed.
- The original structure can be modified inside the function.

3. Pass by Pointer for Read-Only Access

- You can pass a pointer to a const structure if you want to prevent modifications within the function.

Syntax Example:

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void printPoint(const struct Point *p) {
    printf("x = %d, y = %d\n", p->x, p->y);
    // p->x = 50; // Error: Cannot modify a const structure
}

int main() {
    struct Point p1 = {10, 20};
    printPoint(&p1); // Passing pointer to const structure
    return 0;
}
```

Output:

x = 10, y = 20

Key Notes:

- Prevents accidental modification of the structure.
- Useful for functions that only need to read structure data.

Comparison Table

Method	Advantages	Disadvantages
Pass by Value	Simple and safe (no risk of modifying data).	Inefficient for large structures (copying).
Pass by Reference	Efficient (no copying); can modify the data.	Risk of accidental modification.
Pass Pointer (const)	Efficient and safe for read-only access.	Can't modify the structure even if needed.

Practical Considerations:

- Use **pass by value** for small structures and when modification is not required.
- Use **pass by reference** for large structures or when modification is necessary.
- Use **const pointers** for functions that only read the structure's data.

Typedef keyword-

The typedef keyword in C is used to create an alias (alternative name) for an existing type. It makes code more readable, maintainable, and portable by allowing you to define descriptive names for complex data types.

Syntax:

```
typedef existing_type new_type_name;
```

Common Use Cases:

1. Creating Shorter Aliases for Data Types

Simplifies working with long or complex data type declarations.

```
typedef unsigned int uint;
uint age = 25;  // 'uint' is now an alias for 'unsigned int'
```

2. Defining New Names for Structures

Makes code more readable and avoids repeated use of the struct keyword.

```
#include <stdio.h>

typedef struct {
   int x;
   int y;
} Point;

int main() {
```

```
    Point p1 = {10, 20};  // No need to write 'struct Point'
    printf("x = %d, y = %d\n", p1.x, p1.y);
    return 0;
}
```

3. Enhancing Code Portability

Allows easy redefinition of types for different platforms.

```
typedef int BOOL;
#define TRUE 1
#define FALSE 0

BOOL isReady = TRUE;
```

4. Pointer Type Aliases

Simplifies pointer syntax for readability.

```
typedef int* IntPtr;

IntPtr p1, p2;  // Both p1 and p2 are pointers to int
```

5. Function Pointer Typedefs

Makes function pointer declarations more understandable.

```
typedef void (*Callback)(int);

void myFunction(int x) {
    printf("Value: %d\n", x);
}

int main() {
    Callback cb = myFunction;  // Assign function to the callback
    cb(10);  // Call the function through the pointer
    return 0;
}
```

Advantages of typedef:

1. **Improves Code Readability**: Abstracts complex or verbose type definitions.
2. **Reduces Repetition**: Avoids repeatedly specifying the same type declaration.
3. **Improves Portability**: Makes it easier to adapt code for different environments or platforms.
4. **Encapsulates Implementation Details**: Hides underlying data type complexities.

Common Mistakes:

- **Confusion with Macros**: Unlike #define, typedef is interpreted by the compiler and respects scope.
- **Pointer Ambiguity**: Declaring multiple variables with one typedef alias may confuse pointer usage. Always declare pointers explicitly:
- typedef int* IntPtr;
- IntPtr p1, p2; // p1 is a pointer, but p2 is NOT a pointer (common mistake)

Comparison with #define:

Feature	typedef	#define
Scope	Limited to the block	Global (text replacement)
Type Checking	Enforced by the compiler	No type checking
Usability	Works with complex types	Limited to simple cases

Example Combining Uses:

```
#include <stdio.h>

typedef struct {
   int x;
   int y;
} Point;

typedef Point* PointPtr;

void printPoint(PointPtr p) {
   printf("x = %d, y = %d\n", p->x, p->y);
}

int main() {
   Point p = {10, 20};
   PointPtr ptr = &p;
   printPoint(ptr);  // Access using typedef aliases
   return 0;
}
```

Output:

x = 10, y = 20

By using typedef, the code becomes cleaner and easier to understand.

Practice set-10:

1. Create a two-dimensional vector using structures in C.
2. Write a function 'sumVector' which returns the sum of two vectors passed to it. The vectors must be two–dimensional.
3. Twenty integers are to be stored in memory. What will you prefer- Array or structure?
4. Write a program to illustrate the use of arrow operator → in C.
5. Write a program with a structure representing a complex number.
6. Create an array of 5 complex numbers created in Problem 5 and display them with the help of a display function. The values must be taken as an input from the user.
7. Write problem 5's structure using 'typedef' keywords.
8. Create a structure representing a bank account of a customer. What fields did you use and why?
9. Write a structure capable of storing date. Write a function to compare those dates.
10. Solve problem 9 for time using 'typedef' keyword.

Chater-11: Union-

What is Union?

In C, a **union** is a special data type that allows you to store different data types in the same memory location. The key feature of a union is that **all its members share the same memory space**, so only one member can hold a value at any given time. This is in contrast to a structure (struct), where each member has its own memory.

Syntax

```
union UnionName {
   DataType1 member1;
   DataType2 member2;
   ...
};
```

Key Points

1. **Shared Memory:** All members of a union share the same memory location. The size of the union is determined by the size of its largest member.
2. **Single Active Member:** At any given time, only one member of a union can hold a valid value.

3. **Overwriting Values:** Assigning a value to one member of a union will overwrite the value of other members.

Example

```
#include <stdio.h>

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

int main() {
   union Data data;

   data.i = 10;
   printf("data.i: %d\n", data.i);

   data.f = 220.5;
   printf("data.f: %.2f\n", data.f);

   snprintf(data.str, sizeof(data.str), "Hello");
   printf("data.str: %s\n", data.str);

   // Notice how only the most recent value is valid
   printf("After modifying str:\n");
   printf("data.i: %d\n", data.i); // Overwritten
   printf("data.f: %.2f\n", data.f); // Overwritten

   return 0;
}
```

Output

data.i: 10
data.f: 220.50
data.str: Hello
After modifying str:
data.i: 1819043176
data.f: 302.000000

In the example above:

- data.i is initially set to 10, but assigning a value to data.f overwrites the memory.

- Assigning a value to data.str overwrites the memory again, invalidating data.i and data.f.

When to Use Unions

Unions are useful when:

1. **Memory Conservation is Critical:** If you want to store one of several types of values but never more than one at the same time.
2. **Variant Data Types:** For implementing polymorphism or variant data structures, like in embedded systems or device drivers.

Practical Use Case: Variant

```
#include <stdio.h>

typedef union Variant {
   int i;
   float f;
   char c;
} Variant;

typedef struct {
   char type; // 'i', 'f', or 'c'
   Variant value;
} Data;

int main() {
   Data data;

   data.type = 'i';
   data.value.i = 42;
   printf("Integer: %d\n", data.value.i);

   data.type = 'f';
   data.value.f = 3.14;
   printf("Float: %.2f\n", data.value.f);

   data.type = 'c';
   data.value.c = 'A';
   printf("Char: %c\n", data.value.c);

   return 0;
}
```

This approach can simulate a "variant" type that holds one of several possible types.

Declaration and initialization-

Union Declaration

A union is declared using the union keyword. You define a union similar to a structure (struct), but remember, **all members share the same memory**.

Syntax:

```
union UnionName {
   DataType1 member1;
   DataType2 member2;
   ...
};
```

Example:

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

This defines a union named Data with:

- An integer i,
- A float f,
- A character array str.

Union Initialization

You can initialize a union at the time of declaration or later in your code.

1. Static Initialization at Declaration

You can initialize **only one member** of the union at a time since they share the same memory.

Syntax:

```
union UnionName variableName = {initialValue};
```

Example:

```
union Data data1 = {10};       // Initializes `i` to 10
union Data data2 = {220.5f};   // Initializes `f` to 220.5
union Data data3 = {"Hello"};  // Initializes `str` to "Hello"
```

2. Dynamic Initialization (After Declaration)

You can assign values to union members later, but **only the last assigned member will hold a valid value**.

Example:

```
union Data data;

data.i = 42;           // Assign an integer
data.f = 3.14f;        // Overwrites memory, `i` becomes invalid
snprintf(data.str, sizeof(data.str), "Hi!"); // `f` becomes invalid
```

Accessing Union Members

- Use the dot operator (.) to access union members.
- Only access the member that was last assigned a value.

Example:

```
#include <stdio.h>

union Data {
   int i;
   float f;
   char str[20];
};
int main() {
   union Data data = {10};  // Initialize `i` to 10

   printf("data.i: %d\n", data.i);

   data.f = 22.5;  // Assign to `f`, overwriting `i`
   printf("data.f: %.2f\n", data.f);

   snprintf(data.str, sizeof(data.str), "Hello");
   printf("data.str: %s\n", data.str);

   return 0;
}
```

Notes

1. **Size:** The size of a union is the size of its largest member.
2. **Initialization Limitations:** Only one member can be initialized at a time.

3. **Overwrite Behavior:** Writing to one member invalidates the previous value of other members.

Structure & Union-

Structure in C

A structure groups different types of variables under a single name. Each member has its **own memory location**, and all members can store values **simultaneously**.

Syntax:

```
struct StructName {
   DataType1 member1;
   DataType2 member2;
   ...
};
```

Example:

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

Key Points:

1. **Memory Allocation:** Each member has its own memory, so the total memory of a structure equals the sum of the sizes of all members.
2. **Simultaneous Access:** All members can hold values independently and simultaneously.
3. **Usage:** Suitable for grouping related data.

Example Usage:

```
struct Data data = {42, 3.14, "Hello"};
printf("Integer: %d, Float: %.2f, String: %s\n", data.i, data.f, data.str);
```

Union in C

A union is similar to a structure but allows different types of data to share the **same memory** location. Only one member can hold a value at any time.

Syntax:

```
union UnionName {
   DataType1 member1;
   DataType2 member2;
   ...
};
```

Example:

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

Key Points:

1. **Memory Allocation:** All members share the **same memory space**, so the size of a union equals the size of its largest member.
2. **Single Active Member:** At any given time, only one member holds a valid value.
3. **Usage:** Suitable for memory-constrained applications or where only one value is active at a time.

Example Usage:

```
union Data data;
data.i = 42;
printf("Integer: %d\n", data.i);

data.f = 3.14;
printf("Float: %.2f\n", data.f); // Overwrites `i`

snprintf(data.str, sizeof(data.str), "Hello");
printf("String: %s\n", data.str); // Overwrites `f`
```

Differences Between Structure and Union

Feature	Structure	Union
Memory	Separate memory for each member.	Shared memory for all members.
Size	Sum of sizes of all members.	Size of the largest member.
Access	All members can be accessed at once.	Only one member holds a value at a time.

Feature	Structure	Union
Usage	Used to group related data.	Used to save memory in constrained environments.
Overwriting	Members do not overwrite each other.	Writing to one member overwrites others.

Example to Illustrate the Difference

```
#include <stdio.h>

struct StructData {
   int i;
   float f;
   char str[20];
};

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

int main() {
   struct StructData s = {42, 3.14, "Hello"};
   union UnionData u;

   // Structure: All members are independent
   printf("Structure:\n");
   printf("Integer: %d, Float: %.2f, String: %s\n", s.i, s.f, s.str);

   // Union: Members share memory
   printf("\nUnion:\n");
   u.i = 42;
   printf("Integer: %d\n", u.i);

   u.f = 3.14;
   printf("Float: %.2f\n", u.f);  // Overwrites `i`

   snprintf(u.str, sizeof(u.str), "Hello");
   printf("String: %s\n", u.str); // Overwrites `f`
```

```
    return 0;
}
```

Output:
Structure:
Integer: 42, Float: 3.14, String: Hello

Union:
Integer: 42
Float: 3.14
String: Hello
This shows how structures preserve data in all members, while unions allow only one value at a time.

Enumeration-

In C programming, **enumeration (enum)** is a user-defined data type that allows you to assign names to a set of integral constants. This makes the code more readable and easier to maintain. Enums are often used when a variable can take only a limited set of specific values.

Syntax

```
enum EnumName {
  value1,
  value2,
  value3,
  // ...
};
```

Key Points

1. **Underlying Type**: By default, the values in an enum are integers starting from 0 and incrementing by 1 for each subsequent name.
2. **Custom Values**: You can explicitly assign values to some or all members.
3. **Scoped Constants**: The values defined in an enum are globally scoped (or locally scoped if the enum is defined inside a function).

Example

Basic Enum

```
#include <stdio.h>

enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main() {
  enum Days today = Monday;
```

```
  if (today == Monday) {
    printf("It's Monday!\n");
  }

  return 0;
}
```

Output:

```
It's Monday!
```

Enum with Custom Values

```
#include <stdio.h>

enum Status {
  SUCCESS = 1,
  FAILURE = 0,
  PENDING = -1
};

int main() {
  enum Status currentStatus = PENDING;

  if (currentStatus == PENDING) {
    printf("Status: Pending\n");
  }

  return 0;
}
```

Output:

```
Status: Pending
```

Why Use Enums?

- **Improved Readability**: Replace magic numbers with meaningful names.
- **Ease of Debugging**: Easier to understand code logic.
- **Type Safety**: Helps prevent assigning invalid values.

Enums are especially useful for scenarios like representing states, days of the week, error codes, or configuration options.

Practice set-11:

Here's a **practice set for unions in C** to help you strengthen your understanding and skills.

Basic Questions

1. Declaration and Initialization

- Write a program to declare a union Person with members:
  - int id
  - float salary
  - char name[30]
- Initialize and display the values for each member separately. Observe and explain the results.

2. Memory Sharing

- Write a program to:
  - Declare a union with an integer and a float.
  - Assign values to both members and print their values.
  - Explain how memory sharing impacts the output.

Intermediate Questions

3. Size of Union

- Write a program to:
  - Declare a union with three members: int, double, and char.
  - Use sizeof() to determine the size of the union and each member.
  - Compare the size of the union with the size of its largest member and explain the result.

4. Reading Input into Union

- Create a union SensorData with the following members:
  - int temperature
  - float pressure
  - char status[10]
- Write a program that:
  - Allows the user to input and display data for one member at a time.
  - Explain the behavior when switching between members.

Advanced Questions

5. Variant Data Handling

- Create a union Variant with members:
  - int i
  - float f
  - char str[20]
- Use a struct to associate the union with a char type field (e.g., 'i', 'f', 's').
- Write a program to:
  - Assign and print values based on the type field.

- Example: If type = 'i', display the integer value.

6. Union for Memory Optimization

- Implement a program that uses a union to store sensor readings for:
  - Temperature (int in Celsius)
  - Pressure (float in Pascal)
  - Status (char[20] string).
- Discuss the memory efficiency of using a union versus a structure.

7. Endianness Check

- Use a union with an integer and a character array to determine if the system is little-endian or big-endian.

Hint:

```
union {
  int i;
  char c[sizeof(int)];
};
```

8. Union with Bitfields

- Write a program using a union and bitfields to manage a **configuration register**. The union should:
  - Allow access to the full register as an integer.
  - Access individual bits or groups of bits (like enable flags).

Chapter-12: File Operation-

What is file operation?

File operations in C refer to the process of working with files—creating, opening, reading, writing, and closing files—using the C programming language. Files are essential for storing data persistently.

File operations in C are performed using functions from the **stdio.h** library. These operations allow programs to interact with files stored on disk.

File open, close and read, write operation-

1. Opening a File

Before reading from or writing to a file, it must be opened using the **fopen()** function.

Syntax:

```
FILE *fopen(const char *filename, const char *mode);
```

Common Modes:

Mode	Description
"r"	Open file for reading (file must exist).
"w"	Open file for writing (creates a new file or overwrites if file exists).
"a"	Open file for appending (creates a new file if it doesn't exist).
"r+"	Open file for reading and writing (file must exist).
"w+"	Open file for reading and writing (overwrites if file exists).
"a+"	Open file for reading and appending (creates a new file if it doesn't exist).

Example:

```
FILE *filePointer;
filePointer = fopen("example.txt", "w");
if (filePointer == NULL) {
   printf("Error opening file.\n");
}
```

2. Closing a File

Files should always be closed after operations to free system resources using **fclose()**.

Syntax:

```
int fclose(FILE *stream);
```

Example:

```
fclose(filePointer);
```

3. Writing to a File

You can write to a file using functions like **fprintf()**, **fputs()**, or **fputc()**.

Writing Functions:

Function	Purpose
fprintf()	Writes formatted data to a file.
fputs()	Writes a string to a file.
fputc()	Writes a single character to a file.

Example:

```
FILE *filePointer = fopen("example.txt", "w");
if (filePointer != NULL) {
   fprintf(filePointer, "Hello, World!\n"); // Writes formatted text
   fputs("This is a file operation example.\n", filePointer); // Writes a string
   fputc('A', filePointer); // Writes a single character
   fclose(filePointer);
}
```

4. Reading from a File

You can read from a file using functions like **fscanf()**, **fgets()**, or **fgetc()**.

Reading Functions:

Function	Purpose
fscanf()	Reads formatted data from a file.
fgets()	Reads a line (string) from a file.
fgetc()	Reads a single character from a file.

Example:

```
FILE *filePointer = fopen("example.txt", "r");
char buffer[100];

if (filePointer != NULL) {
   while (fgets(buffer, sizeof(buffer), filePointer)) {
      printf("%s", buffer); // Print each line
   }
   fclose(filePointer);
}
```

5. Complete Example

```
#include <stdio.h>

int main() {
   FILE *filePointer;

   // Open file for writing
   filePointer = fopen("example.txt", "w");
   if (filePointer == NULL) {
      printf("Error opening file for writing.\n");
      return 1;
   }
   fprintf(filePointer, "Hello, World!\n");
   fputs("This is a test file.\n", filePointer);
   fclose(filePointer);

   // Open file for reading
   filePointer = fopen("example.txt", "r");
   if (filePointer == NULL) {
      printf("Error opening file for reading.\n");
```

```
    return 1;
  }

  char buffer[100];
  printf("File contents:\n");
  while (fgets(buffer, sizeof(buffer), filePointer)) {
    printf("%s", buffer);
  }

  fclose(filePointer);
  return 0;
}
```

Output:
File contents:
Hello, World!
This is a test file.

6. Key Points

1. **Error Handling:** Always check if fopen() returns NULL to handle file-related errors.
2. **Modes:** Choose the file mode carefully ("r", "w", "a", etc.) to prevent accidental data loss.
3. **Closing Files:** Always close files using fclose() to avoid memory/resource leaks.
4. **Formatted vs. Raw Access:** Use fprintf() and fscanf() for formatted text, and fgets()/fputc() for raw character-based I/O.

This summary should cover the essentials of file open, close, read, and write operations in C. Let me know if you need any clarifications!

Binary file operation-

Binary file operations in C allow you to read from and write to files in binary format, where data is handled as raw bytes rather than formatted as text. This is useful for efficiently storing and processing non-text data, such as images, multimedia files, or binary structured data.

Steps for Binary File Operations in C

1. **Include Necessary Header Files**
2. #include <stdio.h>
3. **Open the File** Use the fopen() function with the appropriate mode:
    - "rb": Open for reading in binary mode.
    - "wb": Open for writing in binary mode.

- "ab": Open for appending in binary mode.
- "rb+", "wb+", "ab+": Open for reading and writing in binary mode.

4. **Perform File Operations** Use functions like fread() and fwrite() for binary data:
   - fread() reads binary data from a file into memory.
   - fwrite() writes binary data from memory to a file.
5. **Close the File** Always close the file using fclose() to free resources.

Example: Writing and Reading Binary Data
Structure to Write and Read

```
typedef struct {
   int id;
   char name[50];
   float score;
} Student;
```

Writing to a Binary File

```
#include <stdio.h>

int main() {
   FILE *file;
   Student student = {1, "Alice", 95.5};

   // Open the file for writing in binary mode
   file = fopen("student.dat", "wb");
   if (file == NULL) {
      printf("Unable to open file!\n");
      return 1;
   }

   // Write the student structure to the file
   fwrite(&student, sizeof(Student), 1, file);

   // Close the file
   fclose(file);
   printf("Data written to file successfully.\n");

   return 0;
}
```

Reading from a Binary File

```
#include <stdio.h>

int main() {
   FILE *file;
   Student student;

   // Open the file for reading in binary mode
   file = fopen("student.dat", "rb");
   if (file == NULL) {
      printf("Unable to open file!\n");
      return 1;
   }

   // Read the student structure from the file
   fread(&student, sizeof(Student), 1, file);

   // Close the file
   fclose(file);

   // Display the data
   printf("ID: %d\n", student.id);
   printf("Name: %s\n", student.name);
   printf("Score: %.2f\n", student.score);

   return 0;
}
```

Notes

1. **Error Handling:** Always check if the file was opened successfully (fopen() returns NULL on failure).
2. **Binary Mode:** Binary files are not platform-independent. Endianness and structure padding may cause compatibility issues when transferring files between systems.
3. **Efficient Storage:** Binary files store data in a compact format without any conversion, which is ideal for large datasets.

This example demonstrates the basics of binary file operations. You can expand it further by working with arrays of structures or handling complex data types.

Practice set-12:

1. Write a program to read three integers from a file.
2. Write a program to generate multiplication table of a given number in text format. Make sure that the file is readable and well formatted.
3. Write a program to read a text file character by character and write its content twice in separate file.
4. Take name and salary of two employees as input from the user and write them to a text file in the following format:
   i. Name1, 3300
   ii. Name2, 7700
5. Write a program to modify a file containing an integer to double its value.

Chater-13: Pre-Processor Directive-

Definition-

Preprocessor directives are instructions that are processed by the C preprocessor before the actual compilation of the code begins. They are used to perform operations like including files, defining macros, conditional compilation, and more. These directives are not C language commands but are interpreted by the preprocessor, which then transforms the source code before compilation.

Pre-processor & their function-

Preprocessor and Their Functions in C

The **preprocessor** in C is a tool that processes the code before the actual compilation begins. It performs various operations like file inclusion, macro expansion, conditional compilation, and more. Preprocessor directives are not part of the C language itself, but they are processed by the preprocessor before the code is compiled.

Preprocessor directives in C are **commands** that are preceded by a hash (#) symbol. These directives instruct the preprocessor to carry out specific tasks like modifying the code or including additional files.

Common Preprocessor Directives and Their Functions

1. **#include**: Includes files into the program.
   - **Function**: The #include directive is used to include external files (usually header files) into the source code. These files typically contain function prototypes, macro definitions, and constants.
   - **Syntax**:
   - #include <headerfile.h> // System library
   - #include "headerfile.h" // User-defined header file
   - **Example**:
   - #include <stdio.h> // Standard I/O library
   - #include "myheader.h" // Custom header file

2. **#define**: Defines constants or macros.
    - **Function**: The #define directive is used to define symbolic constants or macros that are replaced by the preprocessor before compilation. Macros can perform simple textual substitution or function-like operations.
    - **Syntax**:
    - #define MACRO_NAME value
    - **Example**:
    - #define PI 3.1416
    - #define MAX(a, b) ((a) > (b) ? (a) : (b)) // Macro function
3. **#undef**: Undefines a macro.
    - **Function**: The #undef directive is used to cancel or remove the definition of a macro that was previously defined using #define.
    - **Syntax**:
    - #undef MACRO_NAME
    - **Example**:
    - #define PI 3.1416
    - #undef PI // PI is no longer defined
4. **#ifdef** (If Defined): Conditional compilation if macro is defined.
    - **Function**: The #ifdef directive checks whether a macro is defined. If the macro is defined, the code block following the directive is included in the compilation; otherwise, it is ignored.
    - **Syntax**:
    - #ifdef MACRO_NAME
    - // Code to include if MACRO_NAME is defined
    - #endif
    - **Example**:
    - #define DEBUG
    - #ifdef DEBUG
    - printf("Debugging is enabled.\n");
    - #endif
5. **#ifndef** (If Not Defined): Conditional compilation if macro is not defined.
    - **Function**: The #ifndef directive checks whether a macro is not defined. If it is not defined, the code block following it will be compiled.
    - **Syntax**:
    - #ifndef MACRO_NAME
    - // Code to include if MACRO_NAME is not defined
    - #endif
    - **Example**:
    - #ifndef MAX_BUFFER_SIZE
    - #define MAX_BUFFER_SIZE 1024

- #endif

6. **#else**: Provides an alternative for #if or #ifdef.
   - **Function**: The #else directive provides an alternative block of code to be compiled if the #if or #ifdef condition fails.
   - **Syntax**:
   - #if condition
   - // Code if condition is true
   - #else
   - // Code if condition is false
   - #endif
   - **Example**:
   - #ifdef DEBUG
   - printf("Debug mode\n");
   - #else
   - printf("Release mode\n");
   - #endif
7. **#elif** (Else If): Another condition in #if or #ifdef.
   - **Function**: The #elif directive is used to specify an alternative condition within #if or #ifdef blocks. It acts like an "else if" in C programming.
   - **Syntax**:
   - #if condition1
   - // Code if condition1 is true
   - #elif condition2
   - // Code if condition2 is true
   - #else
   - // Code if neither condition is true
   - #endif
   - **Example**:
   - #define VERSION 2
   - #if VERSION == 1
   - printf("Version 1\n");
   - #elif VERSION == 2
   - printf("Version 2\n");
   - #else
   - printf("Unknown version\n");
   - #endif
8. **#if and #endif**: Conditional compilation based on an expression.
   - **Function**: The #if directive evaluates an expression and includes the code that follows it only if the expression is true (non-zero). The #endif ends the conditional block.
   - **Syntax**:

- #if condition
- // Code to include if condition is true
- #endif
- **Example**:
- #define VERSION 2
- #if VERSION > 1
- printf("Version is greater than 1\n");
- #endif

9. **#pragma**: Provides instructions to the compiler.
    - **Function**: The #pragma directive is used to provide instructions to the compiler to control specific compiler features (like warnings, optimizations, etc.). These are compiler-specific and may vary between compilers.
    - **Syntax**:
    - #pragma directive
    - **Example**:
    - #pragma once // Ensures the header file is included only once in the program
10. **#error**: Generates a custom error during preprocessing.
    - **Function**: The #error directive allows you to generate a custom error message during the preprocessor phase. It halts the compilation if the condition is met.
    - **Syntax**:
    - #error "Custom error message"
    - **Example**:
    - #ifndef MY_MACRO
    - #error "MY_MACRO is not defined!"
    - #endif

Summary Table of Preprocessor Directives

Directive	Function
#include	Includes files (header or source) into the program.
#define	Defines a constant or macro.
#undef	Undefines a previously defined macro.
#ifdef	Conditional compilation: checks if a macro is defined.
#ifndef	Conditional compilation: checks if a macro is not defined.
#else	Provides an alternative block for #if or #ifdef.
#elif	Adds another condition in an #if or #ifdef block.
#if	Conditional compilation based on an expression.

Directive	Function
#endif	Ends a conditional compilation block.
#pragma	Compiler-specific directives.
#error	Generates a custom error message.

Key Points

- **Preprocessor directives** are not part of the C syntax; they are handled before compilation starts.
- **#include** is used to add external files (headers) to the program.
- **#define** is used for creating constants and macros.
- **Conditional compilation** (#if, #ifdef, #ifndef, etc.) is useful for platform-dependent code or debugging.
- **#pragma** is used for compiler-specific instructions and optimizations.
- **#error** allows you to stop compilation with a custom message.

Preprocessor directives help you control the compilation process and manage code portability, debugging, and optimization more effectively!

Definition of header file and list standard header file-

Definition of a Header File in C

A **header file** in C is a file containing **declarations of functions**, **macros**, **constants**, and **data types** that are shared across multiple source files. Header files typically have a .h extension and are included in C source files using the **#include** preprocessor directive.

Header files allow for **modularity** in C programs by allowing common code (such as function prototypes, type definitions, or macro definitions) to be shared among different source files. This helps in maintaining the code and reducing redundancy.

Structure of a Header File:

- Function prototypes
- Macro definitions
- Struct and type definitions
- Global variables (rarely used)

By including a header file using #include, the compiler can access the content of the header file during compilation.

Standard Header Files in C

The C Standard Library provides several standard header files that contain functions and macros for performing input/output operations, mathematical calculations, string manipulations, memory management, and more.

Here is a list of **standard header files** in C:

1. **<stdio.h>**:

    - Provides functionalities for input and output (e.g., printf(), scanf(), fopen(), fclose()).
2. **<stdlib.h>**:
    - Contains functions for memory allocation, process control, conversions, and random numbers (e.g., malloc(), free(), exit(), atoi()).
3. **<string.h>**:
    - Provides functions for string manipulation (e.g., strlen(), strcpy(), strcat(), strcmp()).
4. **<math.h>**:
    - Contains mathematical functions (e.g., sqrt(), sin(), cos(), pow()).
5. **<time.h>**:
    - Provides functions to manipulate date and time (e.g., time(), clock(), difftime()).
6. **<ctype.h>**:
    - Defines functions for character handling (e.g., isdigit(), isalpha(), toupper(), tolower()).
7. **<float.h>**:
    - Defines constants for floating-point types (e.g., FLT_MAX, DBL_MIN).
8. **<limits.h>**:
    - Provides constants for integer types (e.g., INT_MAX, LONG_MIN, CHAR_BIT).
9. **<stdarg.h>**:
    - Defines macros for variable argument functions (e.g., va_start(), va_arg(), va_end()).
10. **<assert.h>**:
    - Provides an assertion mechanism for debugging (e.g., assert()).
11. **<errno.h>**:
    - Defines macros for reporting and retrieving error codes (e.g., errno, perror()).
12. **<signal.h>**:
    - Provides functions for signal handling (e.g., signal(), raise()).
13. **<stddef.h>**:
    - Defines several types and macros like NULL, size_t, ptrdiff_t.
14. **<stddef.h>**:
    - Defines types for memory management (e.g., size_t, ptrdiff_t, NULL).
15. **<complex.h>**:
    - Defines functions for complex number arithmetic (e.g., creal(), cimag(), cabs()).
16. **<wchar.h>**:
    - Provides functions for handling wide characters and strings (e.g., wchar_t, wcslen(), wprintf()).

17. **<inttypes.h>**:
    - Provides macros for integer types of specific widths (e.g., PRId32, SCNd64).
18. **<locale.h>**:
    - Defines functions for locale-specific operations (e.g., setlocale(), localeconv()).

Example of Including a Header File

```
#include <stdio.h>  // Include the standard I/O library

int main() {
   printf("Hello, World!\n");  // Use the printf function from stdio.h
   return 0;
}
```

Why Use Header Files?

- **Code Reusability**: Header files allow functions and constants to be reused in multiple source files.
- **Separation of Concerns**: Header files separate the declaration (interface) of functions from their definition (implementation), making code modular and easier to maintain.
- **Avoid Duplication**: Header files allow common code to be included without rewriting it in every file.

In summary, **header files** help to organize code, make it modular, and enable easier management of large projects by providing common declarations and definitions.

Pre-processor of including header file in routine-

In C programming, the **preprocessor** handles the inclusion of header files before the compilation of the source code. The header files typically contain **function prototypes**, **constant definitions**, **macros**, and **type definitions**. By including header files in your C program, you can reuse code from standard libraries or external libraries, and organize your own code more efficiently.
The preprocessor directive used to include header files is **#include**.

How Header File Inclusion Works in C

1. **Preprocessor Directive**: The preprocessor begins its work when it encounters the #include directive. This directive tells the preprocessor to include the contents of another file into the source code at that location.

2. **File Inclusion**: The preprocessor processes header files by either searching for system libraries or user-defined header files.
    - **System Libraries**: These are included by using angle brackets (< >), e.g., #include <stdio.h>. The preprocessor looks for these files in predefined directories.
    - **User-Defined Header Files**: These are included using double quotes (" "), e.g., #include "myheader.h". The preprocessor first looks for the file in the current directory and then searches the system directories if it's not found.
3. **Preprocessor Expansion**: The preprocessor essentially copies the contents of the included header file into the place where #include is used. This happens **before** the actual compilation of the C code.
4. **Avoiding Multiple Inclusions**: Header files might be included multiple times in a program (directly or indirectly). To avoid this redundancy and the errors caused by it, preprocessor guards are commonly used in header files:
    - **Include Guards**: The use of #ifndef, #define, and #endif ensures that the header file is included only once, preventing issues like multiple function declarations or type definitions.

Example of an include guard:

```
// myheader.h
#ifndef MYHEADER_H  // Check if MYHEADER_H is not defined
#define MYHEADER_H  // Define MYHEADER_H

// Function declarations or other contents
void myFunction();

#endif  // End of the include guard
```

This ensures that **myheader.h** is included only once, no matter how many times it is referenced in different source files.

Example of Including a Header File in a C Program

1. Standard Library Header

```
#include <stdio.h>  // Standard library header for input/output functions

int main() {
   printf("Hello, World!\n");  // Use printf function from stdio.h
   return 0;
}
```

2. User-Defined Header

Suppose we have a custom header file mathfunctions.h with function declarations.

- **mathfunctions.h**

```
// mathfunctions.h
#ifndef MATHFUNCTIONS_H
#define MATHFUNCTIONS_H

int add(int a, int b);  // Function declaration
int multiply(int a, int b);

#endif
```

mathfunctions.c

```
// mathfunctions.c
#include "mathfunctions.h"  // Include the header file

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

int multiply(int a, int b) {
   return a * b;
}
```

main.c

```
// main.c
#include <stdio.h>       // Standard I/O header
#include "mathfunctions.h" // Custom header for math functions

int main() {
   int result1 = add(2, 3);
   int result2 = multiply(4, 5);
   printf("Sum: %d, Product: %d\n", result1, result2);
   return 0;
}
```

In this example:

- mathfunctions.h contains the function declarations.
- mathfunctions.c implements the functions.
- main.c includes both the standard library header and the user-defined header, allowing access to the add() and multiply() functions.

Why Use Header Files and the #include Directive?

1. **Modularity**: Header files allow the code to be modular by separating the declaration of functions and types from their implementation.
2. **Code Reusability**: Once a header file is created, it can be included in multiple source files, avoiding redundancy.
3. **Separation of Concerns**: The implementation of functions is separated from their interface (declaration), which makes the program more readable and maintainable.
4. **Efficiency**: By including only the necessary header files and using preprocessor directives, you can control which parts of the code are compiled, especially in larger programs.

Important Preprocessor Directives for Header Files

1. **#include**: Includes the contents of a header file.
    - #include <stdio.h>: Includes system library headers.
    - #include "myheader.h": Includes a user-defined header file.
2. **#ifndef, #define, and #endif**: Protects the header file from multiple inclusions using an **include guard**.
    - **Example**:
    - #ifndef HEADER_NAME_H
    - #define HEADER_NAME_H
    - // Declarations
    - #endif
3. **#pragma once**: An alternative to #ifndef and #define used to ensure the file is only included once, but this is compiler-dependent.
    - **Example**:
    - #pragma once
    - // Declarations

Summary of Preprocessor for Including Header Files in C

- The **#include** preprocessor directive is used to include external files in your C program, either standard libraries or user-defined files.
- **Standard header files** are included using angle brackets (< >), while **user-defined header files** are included using double quotes (" ").
- The **preprocessor** replaces #include with the contents of the header file before actual compilation.
- To **avoid multiple inclusions** of the same header file, use **include guards** with #ifndef, #define, and #endif.
- **#pragma once** is another option to prevent multiple inclusions but is compiler-specific.

By using header files properly, you can manage your C code more effectively, improve its organization, and ensure that it remains modular, reusable, and maintainable.

Constand directive-

In C programming, constants are values that do not change during the execution of a program. The **constant directive** is typically defined using the **#define** preprocessor directive or the **const** keyword. These are used to define constants that can be accessed throughout the program.
The main difference is that #define is a preprocessor directive, while const is a type modifier for variables.

1. Using #define for Constants
The **#define** directive is a preprocessor command that allows you to define symbolic constants. When the compiler encounters a #define statement, it replaces all instances of the symbolic constant with its defined value before compilation.
Syntax:
#define CONSTANT_NAME value

- **CONSTANT_NAME** is the name of the constant, and it is typically written in **uppercase** by convention.
- **value** is the value that the constant represents.

Example:

```
#include <stdio.h>

#define PI 3.1416
#define MAX_LENGTH 100

int main() {
   printf("The value of PI is: %f\n", PI);
   printf("The maximum length is: %d\n", MAX_LENGTH);
   return 0;
}
```

In this example:

- PI is a constant that represents the value 3.1416.
- MAX_LENGTH is a constant representing 100.

The #define directive makes it possible to use these constants throughout the code, and the preprocessor will replace PI and MAX_LENGTH with their respective values during compilation.

Advantages of Using #define for Constants

1. **Code readability**: Constants are given meaningful names, which makes the code more readable and maintainable.
2. **No memory usage**: Constants defined with #define are replaced by their values at compile time, so they do not consume memory during program execution.
3. **Global visibility**: Constants defined with #define are accessible throughout the program, even across multiple source files (if declared in a shared header).

2. Using const Keyword for Constants

The **const** keyword is used to declare constants as variables. Unlike #define, which replaces the constant in the source code, const creates variables whose values cannot be changed after they are initialized.

Syntax:

const data_type CONSTANT_NAME = value;

- **data_type** is the type of the constant (e.g., int, float).
- **CONSTANT_NAME** is the name of the constant, and it is typically written in **uppercase**.
- **value** is the constant's value.

Example:

```
#include <stdio.h>

const float PI = 3.1416;
const int MAX_LENGTH = 100;

int main() {
   printf("The value of PI is: %f\n", PI);
   printf("The maximum length is: %d\n", MAX_LENGTH);
   return 0;
}
```

In this example:

- PI is a constant of type float.
- MAX_LENGTH is a constant of type int.

Unlike #define, the const keyword creates a constant variable, and its value cannot be changed during execution. If you attempt to modify a const variable, the compiler will produce an error.

Advantages of Using const for Constants

1. **Type safety**: Constants defined with const are type-checked by the compiler, which can prevent certain types of errors (e.g., trying to assign a non-compatible value to the constant).

2. **Memory allocation**: const variables are allocated memory, unlike #define, which is a simple text substitution. This can be useful when debugging or inspecting the constant values at runtime.
3. **Scope control**: Constants defined with const can be scoped within functions or files, which provides better control over where they can be accessed, unlike #define, which has global visibility.

Differences Between #define and const Constants

Feature	**#define**	**const**
Type	No type checking (text replacement)	Type is checked by the compiler
Memory Allocation	No memory allocated	Memory is allocated
Scope	Global, unless defined in a header	Limited to the scope of declaration
Debugging	Cannot be debugged or inspected	Can be debugged and inspected
Reusability	Suitable for constants used globally	Suitable for scoped constants

3. Constant Expressions with #define and const

Sometimes, you may want to define constants that are **expressions** rather than simple values.

Using #define for expressions:

#define SQUARE(x) ((x) * (x))

Here, SQUARE is a macro that calculates the square of a number. It is **not a constant** in the strict sense, but it allows you to perform calculations at compile time.

Using const for expressions:

const int area = 5 * 5;

In this case, area is a constant variable initialized with the result of an expression. It is evaluated at runtime, unlike #define, which replaces the expression with its value at compile time.

Summary

- **#define** is a preprocessor directive that defines symbolic constants or macros, and it is **replaced by the preprocessor** during compilation.
- **const** is a keyword used to define constant variables that are **type-safe** and **cannot be modified** during execution.
- Use **#define** for simple constants and macros, especially when you need a constant that doesn't require type checking.

- Use **const** when you need a constant with a specific type and scope, especially when type safety and debugging are important.

By using constants effectively in your program, you can improve its readability, maintainability, and reduce errors caused by magic numbers or hardcoded values.

Conditional compilation directive-

In C, **conditional compilation** allows you to include or exclude parts of code based on specific conditions, often determined by preprocessor directives. These directives are processed before the compilation starts, enabling you to manage code that should only compile under certain conditions (e.g., different platforms, debug builds, or specific feature flags).

Common Conditional Compilation Directives in C

1. # if
   This directive checks whether a given expression evaluates to true (non-zero). If true, the block of code following it is compiled.

```
code
  # if CONDITION
  // Code to be compiled if CONDITION is true
  #endif
```

3. ifdef
   This checks whether a macro (predefined or user-defined) is defined. If it is, the following code block is compiled.

```
 code
  # ifdef MY_MACRO
  // Code to be compiled if MY_MACRO is defined
  #endif
```

4. ifndef
   This checks if a macro is not defined. If the macro has not been defined, the block of code is compiled.

```
  code
  #ifndef MY_MACRO
  // Code to be compiled if MY_MACRO is NOT defined
```

```
#endif
```

4. #else
   This provides an alternative block of code if the condition in `#if`, `#ifdef`, or `#ifndef` is not met.

```
code
#if CONDITION
// Code for true condition
#else
// Code for false condition
#endif
```

5. #elif
   This stands for "else if" and allows you to check additional conditions if the previous `#if` condition is false.

```
code
#if CONDITION1
// Code for CONDITION1
#elif CONDITION2
// Code for CONDITION2
#else
// Code for neither condition
#endif
```

5. #endif
   This ends a conditional block, closing off the scope of `#if`, `#ifdef`, or `#ifndef`.

Example

Here's a simple example that uses conditional compilation:

```
code
#include <stdio.h>

// Define a macro to control which code gets compiled
#define DEBUG

int main() {
   int x = 10;
```

```
    #ifdef DEBUG
    printf("Debug mode: x = %d\n", x);  // This code will be included if DEBUG is defined
    #endif

    printf("This always runs.\n");

    return 0;
}
```

In this example, the `printf` inside the `#ifdef DEBUG` block will only be compiled if the `DEBUG` macro is defined.

Use Cases for Conditional Compilation

- **Platform-specific code**: Including/excluding code based on the operating system or hardware platform.

```
code
#ifdef _WIN32
// Code specific to Windows
#elif __linux__
// Code specific to Linux
#endif
```

Debugging: Adding code that is only compiled when debugging (e.g., logging or assertions).

```
code
#ifdef DEBUG
printf("Debugging information\n");
#endif
```

Feature flags: Enabling or disabling features based on build configurations.

```
code
#define FEATURE_X
#ifdef FEATURE_X
// Code for Feature X
#endif
```

This feature is especially helpful in large software systems where the same codebase needs to be compiled under different conditions without modifying the source code manually.

Pragma directive-

In C, the #pragma directive is used to provide special instructions to the compiler. Unlike other preprocessor directives, which are used for conditional compilation or macro definitions, #pragma is more compiler-specific and is used to enable or disable specific features or optimizations in the compilation process.
The #pragma directive does not have a standardized behavior across all compilers; its meaning and effects depend on the specific compiler being used. However, common uses of #pragma across different compilers include controlling warnings, optimizations, packing structures, and thread-related operations.

General Syntax of #pragma

#pragma directive_name arguments

Where directive_name is a specific instruction supported by the compiler, and arguments are the parameters required for that particular directive.

Common #pragma Directives

1. **#pragma once** This is used to ensure that a header file is included only once in a single compilation unit, preventing multiple inclusions of the same header file. It is a modern, preferred alternative to include guards.

```
#pragma once
// Contents of the header file
```

Example Usage:

#pragma once
#include <stdio.h>
This will guarantee that the header file is included only once, even if multiple #include statements appear for the same file.

2. **#pragma GCC (specific to GCC)** In the GNU Compiler Collection (GCC), #pragma is used to control optimizations, alignments, and warnings.

Example:

#pragma GCC optimize("O3") // Optimization level O3 for the following code
This tells the GCC compiler to optimize the code following this directive at level O3 (max optimization).

3. **#pragma pack** The #pragma pack directive controls the alignment of members in a structure, allowing you to specify the byte boundary to which structure

members should be aligned. This is often used to reduce memory consumption or control the binary layout for communication with hardware.

Example:

```
#pragma pack(1)
struct MyStruct {
  char a;
  int b;
};
```

In this example, #pragma pack(1) ensures that the structure members are packed with 1-byte alignment, which may save memory but can lead to less efficient memory access.

4. **#pragma warning (specific to MSVC)** This directive allows you to disable or enable specific compiler warnings.

Example:

#pragma warning(disable : 4996)

This disables warning 4996 (which is typically related to the use of unsafe functions like gets() in MSVC compilers).

5. **#pragma message** This allows you to generate a custom message during compilation. It can be used to display warnings or other informative messages for the developer.

Example:

```
#pragma message("Compiling for the debug version")
```

This will print the message "Compiling for the debug version" during the compilation process.

6. **#pragma region and #pragma endregion (MSVC specific)** These directives are used to define collapsible code blocks in the Visual Studio IDE. It does not affect the compiled code but helps improve the organization and readability of code within the IDE.

Example:

```
#pragma region MyRegion
// Some code
#pragma endregion
```

7. **#pragma omp (OpenMP)** OpenMP (Open Multi-Processing) uses #pragma to specify parallel regions and directives for multi-threaded programming.

Example:

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
   printf("Parallel thread: %d\n", i);
}
```

This tells the compiler to parallelize the for loop, using multiple threads to execute iterations in parallel.

Compiler-Specific Pragmas

- **GCC** and **Clang** support many pragmas for optimization, alignment, and target-specific instructions.
- **Microsoft Visual Studio (MSVC)** provides several pragmas for controlling warnings, optimizations, and the organization of code within the IDE.

Example Program with Pragmas

```
#include <stdio.h>

#pragma pack(1)  // Ensure structures are packed to 1-byte alignment

struct MyStruct {
   char a;  // 1 byte
   int b;   // 4 bytes (but packed, so no padding)
};

#pragma warning(disable : 4996)  // Disable warning about unsafe functions

int main() {
   struct MyStruct s;
   s.a = 'X';
   s.b = 12345;
   printf("Size of MyStruct: %zu bytes\n", sizeof(s));  // Should print 5 bytes due to
packing
   return 0;
}
```

Notes on Pragmas

- **Portability**: Since #pragma directives are often compiler-specific, code that uses them may not be portable across different compilers. It's important to check the compiler documentation for supported pragmas.

- **Effect on Code**: Unlike other preprocessor directives (like #define or #ifdef), #pragma does not modify the code directly; it influences how the compiler processes the code during the compilation.

Pragmas are a powerful tool for influencing compilation but should be used with caution, especially when targeting multiple platforms or compilers.

Special directive-

In C programming, **special directives** are preprocessor commands that affect the compilation process or the structure of the code in specific ways. These directives are typically used to control compiler behavior, manage code inclusion, or influence memory management. While not all preprocessor directives are "special" in the sense of being unique or rarely used, there are a few that play distinct roles in how C programs are compiled or executed.

Here's a breakdown of some notable special directives in C:

1. `#define` (Macro Definition)

   Purpose: Used to define macros or symbolic constants that can be used throughout the program. It is also used for function-like macros.

   Syntax:

   ```
   #define MACRO_NAME value
   ```

   Example:

   ```
   #define PI 3.14159
   ```

   This will replace every occurrence of `PI` in the code with `3.14159` before compilation.

2. `#include` (File Inclusion)

   Purpose: Used to include the contents of a file in the current file. It is often used to include header files for function prototypes, structure definitions, etc.

   Syntax:

   ```
   #include <filename>  // For standard libraries
   #include "filename"  // For user-defined header files
   ```

   Example:

   ```
   #include <stdio.h>  // Standard library for I/O functions
   #include "myheader.h"  // User-defined header
   ```

Explanation: The #include directive tells the preprocessor to copy the contents of the specified file into the current file before compilation.

3. #ifdef, #ifndef, #endif (Conditional Compilation)

Purpose: These directives are used to include or exclude code based on whether a macro is defined or not. They help in making code conditional based on various factors, such as platform, build configuration, or debugging settings.

Syntax:

```
#ifdef MACRO_NAME
  // Code to be included if MACRO_NAME is defined
#endif
```

Example:

```
#define DEBUG
#ifdef DEBUG
printf("Debugging enabled\n");
#endif
```

Explanation: If DEBUG is defined, the printf statement will be included; otherwise, it will be excluded during compilation.

4. #if, #elif, #else, #endif (Extended Conditional Compilation)

Purpose: Allows more complex conditional compilation using expressions and multiple conditions.

Syntax:

```
#if expression
  // Code if expression is true
#elif expression
  // Code for an alternative condition
#else
  // Code if no conditions are met
#endif
```

Example:

```
#define OS_LINUX
#if defined(OS_WINDOWS)
  printf("Windows OS\n");
#elif defined(OS_LINUX)
  printf("Linux OS\n");
#else
  printf("Unknown OS\n");
#endif
```

Explanation: This example checks whether OS_WINDOWS or OS_LINUX is defined and prints the corresponding message.

5. #undef (Macro Undefinition)

Purpose: Used to undefine a previously defined macro. This can be useful for clearing or modifying macro definitions during the program.

Syntax:

```
#undef MACRO_NAME
```

Example:

```
#define PI 3.14159
#undef PI
#define PI 3.14  // Redefines PI with a new value
```

Explanation: The #undef directive removes the previous definition of PI, allowing you to redefine it.

6. #pragma (Compiler-Specific Directives)

Purpose: Allows for special instructions to the compiler. The behavior of #pragma is compiler-dependent and can affect optimizations, warnings, and other aspects of the compilation process.

Syntax:

```
#pragma directive_name
```

Example:

```
#pragma pack(1)  // Tells the compiler to pack structure members with 1-byte alignment
```

Explanation: #pragma directives can be used to control various features in the compilation process, like alignment of structure members, optimization settings, or enabling/disabling warnings.

7. #error (Error Message)

Purpose: Used to generate a custom error message during compilation. This can be helpful for checking certain conditions during preprocessing or enforcing specific configurations.

Syntax:

```
#error "Error message"
```

Example:

```
#ifndef MY_MACRO
#error "MY_MACRO must be defined"
#endif
```

Explanation: If MY_MACRO is not defined, the compiler will stop compilation and display the error message.

8. #line (Line Number Control)

Purpose: This directive changes the line number and file name that the compiler uses for error reporting. It is commonly used in code generation tools or when manipulating code dynamically.

Syntax:

```
#line line_number "filename"
```

Example:

```
#line 100 "mygeneratedfile.c"
// From now on, the compiler will report errors as if this code was in "mygeneratedfile.c" at line 100.
```

Explanation: This is useful in situations where code is generated dynamically, and the line numbers need to be adjusted for debugging purposes.

9. #typedef (Not a Standard Directive, but Often Used in Code Generation)

- While typedef is a language feature and not a preprocessor directive, it is commonly used in C programs for defining new types. It allows for renaming existing types, making the code easier to read and more portable.

```
typedef unsigned int uint;
```

Summary of Special Directives

#define: Defines constants or macros.

#include: Includes header files in the source code.

#ifdef, #ifndef, #endif: Conditional inclusion based on macro definitions.

#if, #elif, #else, #endif: More flexible conditional inclusion.

#undef: Undefines a macro.

#pragma: Compiler-specific instructions (e.g., optimizations, alignment).

#error: Causes the preprocessor to stop with an error message.

#line: Modifies the line number and file name for error reporting.

These special directives play a crucial role in controlling compilation, debugging, and structuring code in C, enabling conditional compilation, improving portability, and managing code complexity effectively.

Uses of macro-

A **macro** in C is a fragment of code defined using the #define directive. Macros are processed by the **preprocessor** and provide a way to create reusable, substitutable pieces of code that can enhance readability, reduce repetition, and control conditional compilation. Below are the key uses of macros in C programming:

1. Defining Constants

Purpose: Macros are often used to define symbolic constants, making the code more readable and easier to maintain.

Syntax:

#define CONSTANT_NAME value

Example:

#define PI 3.14159

#define MAX_BUFFER_SIZE 1024

Advantages:

Easy to update constants in one place.

Improves readability by using meaningful names instead of hard-coded values.

2. Inline Code Substitution

Purpose: Macros can replace repetitive code with a single, reusable symbol, improving readability and reducing the chance of errors.

Syntax:

#define MACRO_NAME code

Example:

#define SQUARE(x) ((x) * (x))

#define PRINT_HELLO() printf("Hello, World!\n");

Advantages:

Reduces code repetition.

Saves time by defining common operations once.

3. Conditional Compilation

Purpose: Macros can control whether parts of the code are included or excluded during compilation. This is useful for platform-specific code, debugging, or feature toggles.

Syntax:

```
#ifdef MACRO
// Code to include if MACRO is defined
#endif
```

Example:

```
#define DEBUG
#ifdef DEBUG
printf("Debug mode is enabled.\n");
#endif
```

Advantages:

- Enables selective compilation.
- Helps manage code for multiple environments (e.g., Linux vs. Windows).

4. Simplifying Complex Expressions

Purpose: Macros can simplify the use of complex expressions or formulas by giving them a readable name.

Example:

```
#define AREA_OF_CIRCLE(r) (PI * (r) * (r))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Advantages:

- Improves code clarity by replacing complex expressions with intuitive names.

5. Improving Performance

Purpose: Macros eliminate the function call overhead by substituting code directly, which can result in faster execution.

Example:

```
#define MULTIPLY(a, b) ((a) * (b))
```

Note: While macros improve performance, they do not provide type safety like functions. Inline functions may be preferred for this purpose in modern C.

6. Avoiding Magic Numbers

Purpose: Macros help eliminate "magic numbers" (unnamed numeric constants) in code, improving maintainability.

Example:

```
#define MAX_STUDENTS 100
#define DEFAULT_TIMEOUT 30
```

Advantages:

Makes code easier to read and modify.
Reduces the risk of introducing bugs when changing constants.

7. Platform-Specific Code

Purpose: Macros can manage code that needs to be compiled differently for various platforms or compilers.

Example:

```
#ifdef _WIN32
printf("Running on Windows\n");
#else
printf("Running on Unix-based system\n");
#endif
```

Advantages:

Facilitates cross-platform development.

8. Debugging and Logging

Purpose: Macros can simplify the addition of debugging or logging statements, which can be conditionally compiled.

Example:

```
#ifdef DEBUG
#define LOG(msg) printf("DEBUG: %s\n", msg)
#else
#define LOG(msg)
#endif
```

Advantages:

Adds debugging code without cluttering the main logic.
Can easily enable or disable logging.

9. Guarding Header Files

Purpose: Prevent multiple inclusions of the same header file, avoiding redefinition errors.

Syntax:

```
#ifndef HEADER_FILE_NAME
#define HEADER_FILE_NAME
// Header file content
#endif
```

Example:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
// Header content
#endif
```

Advantages:

Ensures proper inclusion of header files.

Avoids compilation errors due to duplicate definitions.

10. Placeholder for Future Features

Purpose: Macros can act as placeholders for features that may be added later, making the code adaptable for changes.

Example:

```
#define FEATURE_X_ENABLED
#ifdef FEATURE_X_ENABLED
printf("Feature X is enabled.\n");
#endif
```

Advantages of Macros

1. **Improved Readability**: Use symbolic names instead of cryptic or complex expressions.
2. **Reusability**: Define once, use anywhere.
3. **Performance**: Faster than functions as macros avoid function call overhead.
4. **Portability**: Helps write code that works across platforms.
5. **Custom Debugging**: Useful for logging or tracking code during debugging.

Disadvantages of Macros

1. **No Type Checking**: Unlike functions, macros are not type-safe, which may lead to bugs.
2. **Code Bloat**: Macros substitute code inline, increasing the compiled size if used excessively.
3. **Debugging Challenges**: Debugging can be difficult because macros are replaced during preprocessing, and their effects are not visible in runtime debuggers.
4. **Complex Syntax**: Overuse of macros with complex expressions can reduce code clarity.

Example Program Using Macros

```
#include <stdio.h>

#define PI 3.14159//macro
#define SQUARE(x) ((x) * (x))//macro
#define DEBUG//macro

int main() {
```

```
    double radius = 5.0;
    printf("Area of circle: %.2f\n", PI * SQUARE(radius));

    #ifdef DEBUG
    printf("Debugging is enabled.\n");
    #endif

    return 0;
}
```

In this program:

- PI is used as a constant.
- SQUARE simplifies the calculation of the square.
- #ifdef DEBUG conditionally includes debugging code.

Conclusion

Macros in C are versatile tools for simplifying code, reducing redundancy, managing configurations, and improving performance. However, their misuse or overuse can lead to maintenance issues, so they should be used judiciously. For safer and modern practices, inline functions or const variables are often preferred over macros for certain tasks.

Practice set-13:

1. Write a program use macro determine the value of circle.
2. Write a program use macro determine the value of Summation of two value.
3. Write a program use macro determine the value of average number of three number.

Chapter-14: Dynamic memory allocation

Dynamic Memory Allocation-

Dynamic memory allocation in C allows you to allocate memory at runtime, meaning you don't need to know the amount of memory required by your program beforehand. This is done using four key functions provided by the C standard library:

1. **malloc()** - Memory Allocation
2. **calloc()** - Contiguous Memory Allocation
3. **realloc()** - Reallocation of Memory
4. **free()** - Freeing the Allocated Memory

Let's go through each of these functions with explanations and code examples.

1. malloc() - Memory Allocation

malloc() stands for memory allocation. It allocates a specified number of bytes of memory and returns a pointer to the first byte of the allocated space. If the memory allocation fails, it returns NULL.

Syntax:

void* malloc(size_t size);

- size: The number of bytes to allocate.

Example:

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

int main() {
   int *ptr;
   ptr = (int*)malloc(5 * sizeof(int));  // Allocating memory for 5 integers

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

   // Initializing and displaying values
   for (int i = 0; i < 5; i++) {
      ptr[i] = i + 1;
      printf("%d ", ptr[i]);
   }

   free(ptr);  // Freeing the allocated memory
   return 0;
}
```

2. calloc() - Contiguous Memory Allocation

calloc() allocates memory for an array of elements and initializes all the memory locations to zero. It is a safer alternative to malloc() when you want the allocated memory initialized.

Syntax:

void* calloc(size_t num_elements, size_t element_size);

- num_elements: The number of elements to allocate.
- element_size: The size of each element in bytes.

Example:

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

int main() {
   int *ptr;
   ptr = (int*)calloc(5, sizeof(int));  // Allocating memory for 5 integers initialized to zero

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

   // Displaying values (should all be 0)
   for (int i = 0; i < 5; i++) {
      printf("%d ", ptr[i]);
   }

   free(ptr);  // Freeing the allocated memory
   return 0;
}
```

3. realloc() - Reallocation of Memory

realloc() is used to resize a previously allocated memory block. It can either increase or decrease the size of the memory block. If the memory is expanded, the new memory will contain garbage values (not initialized).

Syntax:

void* realloc(void* ptr, size_t new_size);

- ptr: A pointer to the previously allocated memory.
- new_size: The new size of the memory block in bytes.

Example:

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

int main() {
   int *ptr;
   ptr = (int*)malloc(3 * sizeof(int));  // Allocating memory for 3 integers

   if (ptr == NULL) {
```

```
    printf("Memory allocation failed.\n");
    return 1;
  }

  for (int i = 0; i < 3; i++) {
    ptr[i] = i + 1;
  }

  // Reallocating memory for 5 integers
  ptr = (int*)realloc(ptr, 5 * sizeof(int));

  if (ptr == NULL) {
    printf("Memory reallocation failed.\n");
    return 1;
  }

  // Displaying the resized array
  for (int i = 0; i < 5; i++) {
    printf("%d ", ptr[i]);
  }

  free(ptr);  // Freeing the allocated memory
  return 0;
}
```

4. free() - Freeing the Allocated Memory

free() is used to release the memory that was previously allocated by malloc(), calloc(), or realloc(). Once memory is freed, the pointer becomes invalid, and it should not be used unless reassigned.

Syntax:

void free(void* ptr);

- ptr: A pointer to the memory block to be freed.

Example:

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

int main() {
  int *ptr;
  ptr = (int*)malloc(5 * sizeof(int));  // Allocating memory for 5 integers
```

```
    if (ptr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Freeing the allocated memory
    free(ptr);

    // After free, ptr becomes invalid
    // Uncommenting the following line would cause undefined behavior:
    // printf("%d", ptr[0]);

    return 0;
}
```

Summary of Functions

Function	Purpose	Syntax
malloc	Allocates memory of a given size	void* malloc(size_t size);
calloc	Allocates memory for an array of elements and initializes them to zero	void* calloc(size_t num_elements, size_t element_size);
realloc	Resizes a previously allocated memory block	void* realloc(void* ptr, size_t new_size);
free	Frees previously allocated memory	void free(void* ptr);

Key Points:

- **Memory management**: Always ensure that dynamically allocated memory is freed after use to avoid memory leaks.
- **NULL Check**: Always check whether memory allocation or reallocation has failed (i.e., if the returned pointer is NULL).
- **Pointer Validity**: After calling free(), the pointer is no longer valid. Avoid using it unless reassigned.

By using dynamic memory allocation functions, you can manage memory more efficiently, especially when dealing with variable-size data structures.

Advantages of Dynamic Memory Allocation

1. **Efficient Memory Usage**:
    - Dynamic memory allocation allows you to allocate memory at runtime based on actual needs, so you can avoid wasting memory. You can allocate memory when needed and release it when it is no longer required, leading to more efficient memory use.

2. **Flexibility**:
    - You can adjust the size of the memory allocation during runtime using functions like realloc(). This is particularly useful when the size of the data (e.g., arrays or buffers) is not known in advance.
3. **Variable Size Data Structures**:
    - Dynamic memory allows the creation of data structures like linked lists, trees, and graphs, where the size can grow or shrink dynamically during program execution.
4. **Improved Resource Management**:
    - By allocating memory dynamically, you can handle large data sets more effectively by allocating memory as needed rather than allocating a fixed block, reducing memory wastage.
5. **Better Performance in Some Cases**:
    - For programs that require varying memory sizes or work with large data sets, dynamic memory allocation can improve performance because it avoids over-allocating memory upfront.

Disadvantages of Dynamic Memory Allocation

1. **Memory Management Overhead**:
    - Dynamic memory allocation requires careful management to avoid memory leaks. If memory is allocated but never freed, it can lead to resource exhaustion, which is particularly problematic in long-running applications.
    - The program must explicitly manage memory allocation and deallocation, making the code more complex and prone to errors.
2. **Slower Execution**:
    - Dynamic memory allocation typically involves more overhead compared to static memory allocation because the system must manage the memory at runtime (e.g., finding free blocks of memory and allocating them).
    - Operations like malloc(), realloc(), and free() can be slower than using statically allocated arrays.
3. **Fragmentation**:
    - Over time, if memory is allocated and freed repeatedly, it can lead to memory fragmentation, where free memory is split into small, non-contiguous blocks. This can reduce the performance and efficiency of memory allocation.
    - Fragmentation might prevent large blocks of memory from being allocated, even if enough total memory is available.
4. **Error Handling**:
    - Dynamic memory allocation can fail if there is not enough memory available. Functions like malloc(), calloc(), and realloc() return NULL

when they cannot allocate memory. Handling such errors properly adds complexity to the code.
    - If dynamic memory is not properly managed or freed, it can result in undefined behavior, including memory corruption.
5. **Increased Complexity**:
    - Dynamic memory allocation adds complexity to the program because the programmer must ensure that memory is allocated and freed correctly.
    - Mistakes like using uninitialized memory, accessing freed memory (dangling pointers), or freeing memory multiple times (double free) can lead to bugs that are hard to detect and fix.
6. **Memory Leaks**:
    - If dynamically allocated memory is not freed (or released), memory leaks occur, which can slowly degrade the performance of a program or cause it to run out of memory over time.

Summary Table

Advantages	**Disadvantages**
Efficient memory usage	Requires careful memory management
Flexibility to resize memory at runtime	Slower execution due to overhead
Allows creation of variable-size data structures	Can cause fragmentation in memory
Better resource management in large data sets	Error handling and memory allocation failures
More efficient with varying data sizes	Increases program complexity

In general, dynamic memory allocation is very useful for scenarios where memory size cannot be determined in advance or when memory needs to change during the execution of the program. However, it requires extra care in its use to avoid errors and inefficiencies.

Function for DMA in C-

Direct Memory Access (DMA) is a mechanism that allows peripherals or memory blocks to directly communicate with the memory of the system without involving the CPU, thus improving the efficiency of data transfer in embedded systems and hardware programming. While C does not have built-in support for DMA, it can be programmed using hardware-specific libraries or through the interaction with system-level functions provided by the operating system.

In embedded systems programming, DMA functionality is typically implemented using hardware registers, specific peripheral drivers, or through specific libraries provided by microcontroller manufacturers (like ARM, AVR, etc.). The function for DMA setup is usually specific to the hardware platform you are working with.

Below is a conceptual framework and a general example for how DMA might be configured in a C program (in the context of an embedded system like ARM Cortex-M microcontrollers).

Conceptual DMA Function in C

In C, DMA is typically configured through memory-mapped registers of the DMA controller. The configuration steps usually involve:

1. **Selecting the DMA channel.**
2. **Specifying source and destination addresses.**
3. **Configuring transfer size and type.**
4. **Starting the DMA transfer.**
5. **Handling interrupts (optional) after the transfer is completed.**

General DMA Setup in C (Conceptual Example)

```
#include <stdint.h>
#include <stdbool.h>

// Example: DMA channel structure (specific to your platform)
typedef struct {
   uint32_t control;
   uint32_t srcAddress;
   uint32_t dstAddress;
   uint32_t transferSize;
} DMA_Channel_TypeDef;

// DMA controller instance (platform-specific)
#define DMA_CHANNEL_0 ((DMA_Channel_TypeDef*) 0x40026000)  // Example address for DMA channel 0

// DMA configuration settings (example)
#define DMA_TRANSFER_SIZE 256  // Size of data to transfer in bytes

// DMA control settings (just an example)
#define DMA_CONTROL_ENABLE   (1 << 0)  // Example control setting: Enable DMA transfer

void DMA_Init(void) {
   // Step 1: Initialize DMA channel (example for channel 0)
   DMA_CHANNEL_0->control = 0;  // Clear any previous settings

   // Step 2: Set source and destination addresses
```

```
  DMA_CHANNEL_0->srcAddress = (uint32_t)sourceBuffer;  // Address of source data
  DMA_CHANNEL_0->dstAddress = (uint32_t)destinationBuffer;  // Address of destination

  // Step 3: Set the size of the data to transfer (e.g., 256 bytes)
  DMA_CHANNEL_0->transferSize = DMA_TRANSFER_SIZE;

  // Step 4: Configure DMA control settings (enable the channel)
  DMA_CHANNEL_0->control = DMA_CONTROL_ENABLE;

  // Step 5: Trigger the DMA transfer (if applicable, depending on hardware)
  // The hardware might trigger DMA automatically once enabled, or a specific start command may be needed
}

void DMA_Transfer(void) {
  // Assuming DMA is triggered automatically by the hardware after initialization
  // Or this function could trigger the DMA transfer manually, depending on the platform

  // Step 6: Wait for DMA to complete (usually through interrupts or polling a status flag)
  while (!DMA_TransferComplete()) {
    // Polling DMA status (or use interrupt-based notification)
  }

  // Step 7: DMA Transfer Complete
  // Optionally clear any flags and disable DMA if needed
  DMA_CHANNEL_0->control &= ~DMA_CONTROL_ENABLE;  // Disable DMA channel (optional)
}

bool DMA_TransferComplete(void) {
  // Check the transfer completion status (example, depends on hardware)
  // For example, check a status register flag
  return (DMA_CHANNEL_0->control & (1 << 1)) != 0;  // Example: Check completion flag
}
```

```
// Sample source and destination buffers
uint8_t sourceBuffer[DMA_TRANSFER_SIZE];
uint8_t destinationBuffer[DMA_TRANSFER_SIZE];

int main(void) {
  // Initialize DMA and start transfer
  DMA_Init();

  // Perform the transfer
  DMA_Transfer();

  // Now the data from sourceBuffer is transferred to destinationBuffer

  return 0;
}
```

Detailed Steps Involved in DMA Setup

1. **Initialize DMA Channel**:
    - Set the DMA channel control register to reset or configure it.
    - The control register could configure the DMA for one-shot or continuous mode, set interrupt flags, and configure transfer direction (memory-to-peripheral or peripheral-to-memory).
2. **Source and Destination Addresses**:
    - Set the memory or peripheral address for the source and destination.
    - For memory transfers, this could be a pointer to a buffer in RAM.
3. **Transfer Size**:
    - Define the size of the data to be transferred (usually in terms of bytes, words, or blocks).
4. **Control Settings**:
    - Configure the DMA to initiate the transfer by enabling the DMA channel.
    - The transfer could be triggered by a peripheral event, a software request, or a timer interrupt, depending on the hardware configuration.
5. **Interrupt Handling** (Optional):
    - After the DMA transfer completes, an interrupt can be triggered to notify the program.
    - The interrupt handler can check for the completion of the transfer and handle any post-transfer actions.
6. **Monitoring Transfer Status**:

- After triggering the DMA, you can monitor the transfer status by polling a status register or using interrupts. Once the transfer completes, the DMA can be stopped, and control can be returned to the program.

Hardware-Specific Considerations

- **Platform/Chip-Specific Registers**: DMA setup typically involves setting values in specific hardware registers, so the actual register names, control flags, and configurations will depend on the microcontroller or platform you're using. For example, ARM Cortex-M processors, AVR, and STM32 chips have different registers and settings for DMA control.
- **Interrupt Handling**: You may need to set up interrupt service routines (ISRs) to handle DMA completion, errors, or other events. This is highly specific to the hardware being used.
- **Buffer Alignment**: Ensure that the source and destination buffers are aligned to the appropriate memory boundaries (e.g., 4-byte or 8-byte aligned) as required by the DMA controller.

Example: DMA in ARM Cortex-M (Using STM32 HAL)

In STM32 microcontrollers, the STM32 HAL library abstracts the hardware and provides a simpler interface for DMA. Here's an example of initializing a DMA transfer in STM32 HAL:

```
#include "stm32f4xx_hal.h"

DMA_HandleTypeDef hdma;

void DMA_Init(void) {
  // Configure DMA
  __HAL_RCC_DMA2_CLK_ENABLE();  // Enable DMA clock (example for DMA2)

  hdma.Instance = DMA2_Stream0;  // Select DMA stream
  hdma.Init.Channel = DMA_CHANNEL_0;  // Select DMA channel
  hdma.Init.Direction = DMA_MEMORY_TO_MEMORY;  // Data direction
  hdma.Init.PeriphInc = DMA_PINC_ENABLE;  // Peripheral address increment mode
  hdma.Init.MemInc = DMA_MINC_ENABLE;  // Memory address increment mode
  hdma.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;  // Peripheral data
alignment
  hdma.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;  // Memory data
alignment
  hdma.Init.Mode = DMA_NORMAL;  // Normal mode (not circular)
  hdma.Init.Priority = DMA_PRIORITY_LOW;  // Low priority
```

```
  HAL_DMA_Init(&hdma);  // Initialize DMA with the configuration

  // Associate DMA handle to the peripheral (example for UART)
  __HAL_LINKDMA(&huart1, hdmatx, hdma);
}

void DMA_Transfer(uint8_t *source, uint8_t *destination, uint32_t size) {
  HAL_DMA_Start(&hdma, (uint32_t)source, (uint32_t)destination, size);  // Start DMA
transfer

  // Wait for transfer completion (or use interrupt)
  while (HAL_DMA_GetState(&hdma) != HAL_DMA_STATE_READY) {
    // Poll for transfer completion
  }
}

int main(void) {
  HAL_Init();  // Initialize HAL library

  uint8_t sourceBuffer[256];
  uint8_t destinationBuffer[256];

  // Initialize the DMA peripheral
  DMA_Init();

  // Start the DMA transfer
  DMA_Transfer(sourceBuffer, destinationBuffer, 256);

  return 0;
}
```

Conclusion

While DMA programming in C depends heavily on the underlying hardware and platform, the general principles remain consistent: DMA involves setting up control registers, specifying memory addresses, and managing the flow of data between peripherals and memory. In embedded systems, DMA is a critical feature to optimize data transfer without burdening the CPU, making your system more efficient and responsive.

Practice set-14:

1. Write a program to dynamically create an array of size 6 capable of storing 6 integers.
2. Use the array in problem 1 to store 6 integers entered by the user.
3. Solve problem 1 using calloc().
4. Create an array dynamically capable of storing 5 integers. Now use realloc so that it can now store 10 integers.
5. Create an array of multiplication table of 7 upto 10 (7 x 10 = 70). Use realloc to make it store 15 number (from 7 x 1 to 7 x 15).
7. Attempt problem 4 using calloc().

PROJECT 2: SNAKE, WATER, GUN

Snake, water, gun or rock, paper, scissors is a game most of us have played during school time. (I sometimes play it even now). Write a C program capable of playing this game with you. Your program should be able to print the result after you choose snake/water or gun.

GitHub-link: https://github.com/AbuBokorSiddik67/C_programming/tree/master/