STUDY MATERIAL FOR B.Sc., COMPUTER SCIENCE WITH ARTIFICIAL INTELLIGENCE

C - PROGRAMMING

SEMESTER – I

ACADEMIC YEAR 2025-26

PREPARED BY

COMPUTER SCIENCE DEPARTMENT

INDEX

UNIT- I

Introduction To C:

- It has been developed by Dennis Ritchie at Bell laboratories in the year of 1972.
- C is a middle level general purpose language
- C is structured Language
- It is called the "mother of all languages" because many modern languages like C++, Java, and Python are influenced by it.

Features of C

- Simple → Easy to learn with basic commands.
- Fast & Efficient → Provides low-level access to memory.
- Portable → A C program written on one computer can run on another with little or no change.
- Structured → Programs can be divided into functions (modules).
- Rich Library → Comes with many built-in functions.
- Middle-level language → Combines features of high-level and low-level languages.

The C Declarations:

A Program is a set of statements for a specific task, which will be executed in a sequential manner to produce the desired output.

Syntax:

data_type variable_name;

Examples:

```
int age;     // declares an integer variable
float salary;  // declares a floating-point variable
char grade;    // declares a character variable
```

Declaration with Initialization:

```
int age = 20;      // declare and assign value
float salary = 2500.50;
char grade = 'A';
```

Multiple Declarations:

```
int a, b, c;       // declares 3 integers
float x = 1.2, y;  // one initialized, one not
```

The C Character Set:

The characters used to form words, numbers and expressions.

The characters in C are Classified into the following:

1. Letters – (Capital A – Z , Small a – z)
2. Digits – (0 - 9)
3. White Spaces – (Blank space, Vertical tab, New Line)
4. Special Characters – (, . ; : ‘ “ ! / \ ~ - $? * & ^ % # @ { } [ ] ( ) + = _ < >)

Delimiters:

: Colon Useful for label

; Semi Colon Terminates statements

( ) Parenthesis Used in expression and function

[ ] Square Bracket Used for array declaration

{ } Curly Brace cope of Statements

Hash Preprocessor directives

, Comma Variable separator

C Tokens

Keywords:

- The C Keywords are reserved words by the compiler
- All the C Keywords have been assigned some fixed meaning

So The C keywords cannot be used as variable name.

Ex:

Auto	double	int	struct
Break	else	long	switch
Case	enum	union	unsigned
Const	do	goto	sizeof
If	static	while	main

Identifiers:

Identifiers are names of variables, functions and arrays.

Lower case letters are preferred, the upper case letters are also permitted. (_) under score symbol can be used as an identifier

Ex:

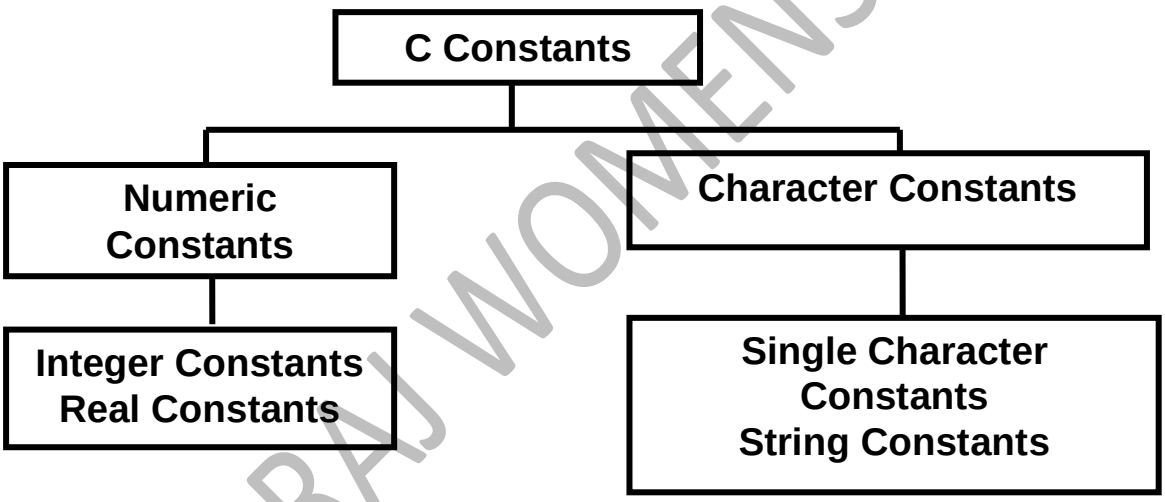
define N 10

define a 15

Constants: (8 mark)

The constants in C are applicable to the values, which do not change during the execution of a program.

They are classified into the following group:



A. Numeric Constants:

i) Integer Constants:

- These are the sequence of numbers from 0 to 9 without decimal points or fractional part or any other symbols
- It requires 2 bytes or maximum 4 bytes
- It could either +ve , -ve or zero

Ex: 10, 20, +30, -25, 0

ii) Real Constants:

- Real constants are often known as floating point constants
- Real constants can be written in exponential notation, which contains a fractional part and a exponential part Ex: 2.5, 5.32, 3.14 etc.

B. Character Constant:

i) Single Character Constants:

- A character constant is a single character enclosed with single quotes.
- It includes single digit, single character, white space Ex: ‘a’, ‘7’, ‘ ’ etc.

ii) String Constants:

- Sting constants are sequence of characters enclosed with a double quote marks. Ex: “Heber”, “Basil”, “888”, “a”

Strings:

Sequence of characters enclosed in double quotes " ".

Example: "Hello", "C Programming

printf("Welcome to C!");

Operators

Symbols that perform operations on variables/values.

Types:

- Arithmetic → + - * / %
- Relational → < <= > >= == !=
- Logical → && || !
- Assignment → = += -=
- Increment/Decrement → ++ --

Example:

```
int a = 5, b = 10;

int sum = a + b;   // '+' is an operator
```

- **Special Symbols**

Used in program structure.

Examples:

; → statement terminator

{ } → block of code

[] → array subscript

() → function call

, → separator

→ pre-processor directive

Variables:

- A variable is a data name used for storing a data value.
- Its value may be changed during the execution of the program

Ex: a , height, sum, avg

Rules for defining variables:

1. They must begin with a character without spaces but underscore is permitted
2. Generally most compilers support 8 characters length Maximum length of a variable upto 31 characters
3. The Variable should not be a C Keywords
4. The variable names may be combination of uppercase and lowercase characters

   Ex: suM, AVg
5. The Variable name should not start with a digit

Format Specifier:

In C, printf (for output) and scanf (for input) use format specifies to tell the compiler what type of data you are printing or reading.

- %d → integer (int)
- %f → float (decimal numbers)
- %c → character
- %s → string (word/text)

Example:

```
#include <stdio.h>

int main() {
   int age = 20;   // integer variable
   printf("My age is %d years old.", age);
   return 0;
}
```

Explanation:

- int age = 20; → here age is an integer.
- printf("My age is %d years old.", age);
- %d is replaced by the value of age.

Output: My age is 20 years old.

Example:

```
#include <stdio.h>

int main() {

   int roll = 101;

   float marks = 95.5;

   char grade = 'A';

   printf("Roll No: %d\n", roll);

   printf("Marks: %f\n", marks);

   printf("Grade: %c\n", grade);

   return 0;

}
```

Output:

Roll No: 101

Marks: 95.500000

Grade: A

Data Types: (8 mark or 15 mark)

The C compiler supports a variety of data types. They are:

Name	Range	Storage Size	Format String	Example
Short integer	-32,768 to +32,767	2 bytes	%d or %I	Int a=2; Short int b=2;
Long Integer	-2,147,483,648 to 2,147,483,647	4 bytes	%ld	Long b; Long int c;
Signed Integer	-32,768 to +32,767	2 bytes	%d or %i	Int a=5; Signed int d;
Un signed integer	0 to 65535	2 bytes	%u	Unsigned integer c=7;
Signed Char	-127 to 127	1 byte	%c	Char c;
Unsigned char	0 to 255	1 byte	%c	Unsigned char c;
float	-3.4e38 to 3.4e38	4 bytes	%f	Float d;
Double	-1.7e-308 to 1.7e+308	8 bytes	%lf	Double d; Long double x;

Declaration of variables:

The Variables must be declared before they are used in the program Declaration Provide two things:

1. Compiler Obtains the variable name
2. It tells the compiler data type of the variable an allocating memory

Syntax: Data_type Variable_Name;

Ex: int age; char c;

int b, c; float d;

Data types:

Char , signed char , Unsigned char , int , signed int , unsigned int , unsigned short int , signed long int , unsigned long int , float , double , long double

Initializing Variables:

Variables can be assigned or initialized using an assignment operator '=' Declaration & initialization can be done in the same line.

Syntax: Data_type Variable_name =Constant;

Ex: int a=10; float b=2.17; int a=b=c=20;

Type Conversion:

Sometimes the programmer needs the result in certain data type

Ex:

division of 5 with 2 should return float value

Ex:

```
# include <stdio.h>
# include <conio.h>//console input output.header file
 main() {
Printf("Two integers (5 & 2) : %d ", 5/2);
Printf("Two integers (5 & 2) : %g ", (float) 5/2); getch();
}
```

Output:

Two integers (5 & 2) : 2

Two integers (5 & 2) : 2.5

2. Operators & Expressions:

An Operator indicates an operation to be performed on data they yield a value.

C Provides 4 Classes of operators:

i) Arithmetic Operators iii) Relational Operators

ii) Logical Operators iv) Bitwise Operators

Types of Operators:

Types of Operators	Symbolic Representation
Arithmetic Operators	+ , - , * , / and %
Relational Operators	> , < , >= , <= , = = , and !=
Increment & Decrement Operator	++ and - -
Assigned Operators	=
Bitwise Operators	& , | , ^ , >> , << and ~

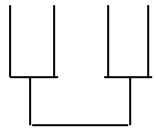
Comma Operator	,
Conditional Operator	? :
Logical Operators	&& , || , !

Priority Of Operators:

```
( )     [ ]     ->     .
+       -       ++     - -     !     ~     *     &
*       /       %
<<      >>
<       <=      >      >=
= =     !=
&
^
|
```

Ex:

X = 5 * 4 + 8 / 2



12

3

i) Comma & Conditional Operator:

a)Comma operator:

It is used to separate two or more variable or expressions

Ex: int a, b, c; a=10, b=20, c=a+b;

Ex:

```
#include <stdio.h>
#include<conio.h>
main() {
clrscr();
printf("Addition = %d \t Subtraction = %d", 10+20, 5-2);
getch();
}
```

Output:

Addition = 30

Subtraction = 3

Note:

<conio.h> Console Input Output Header file

It is a header file in C (used in old compilers like Turbo C / Borland C).

It contains functions for handling input and output on the console screen (not standard C).

Common Functions in <conio.h>

1. clrscr(); → Clears the screen.

2. getch(); → Waits for a single key press (doesn't show it on screen).

3. getche(); → Reads a single key press and displays it on screen.

4. cprintf(); → Prints formatted text in the console.

5. gotoxy(x,y); → Moves cursor to a particular position on the screen.

b) Conditional Operator: (?;)

- The conditional operator contains a condition followed by two statements or values.
- If the condition is true the first statement is executed otherwise the second statement is executed
- The Conditional Operator? And : are sometimes called ternary operators

Syntax: Condition? (expression1): (expression2);

Ex:

```
main() {
clrscr();
3 > 2 ?Printf("3 is big") : printf("2 is big");
getch();
}
```

Output:

3 is big

ii) Arithmetic Operators: (8 Marks):

Two type of arithmetic operators:

a) Binary Operators b) Unary Operators

Binary operator:

Binary arithmetic operators are used for numerical calculations between the two constant values.

+ Addition - subtraction

* Multiplication / Division

% Modular Division (Remainder)

Ex: You should give Example Program

EXAMPLE:

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

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

   printf("a + b = %d\n", a + b);  // Addition
   printf("a - b = %d\n", a - b);  // Subtraction
   printf("a * b = %d\n", a * b);  // Multiplication
   printf("a / b = %d\n", a / b);  // Division
   printf("a %% b = %d\n", a % b); // Modulus

   getch();
}
```

Unary Operators:

- Minus ++ Increment

- Increment & Address operator

Size of gives the size of variables

a) Minus (-):

Indicates the algebraic sign of a value

Ex: int x = -10; int y = -25;

b) Increment (++) and Decrement (--) Operators:

- The operator ++ adds one to its operand
- The operator – subtract one from its operand

Ex:

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

```
int main() {
   int x = 10, y;
   clrscr();

   y = x++;  // Post-increment
   printf("After y = x++ : x = %d, y = %d\n", x, y);

   y = ++x;  // Pre-increment
   printf("After y = ++x : x = %d, y = %d\n", x, y);

   y = x--;  // Post-decrement
   printf("After y = x-- : x = %d, y = %d\n", x, y);

   y = --x;  // Pre-decrement
   printf("After y = --x : x = %d, y = %d\n", x, y);

   getch()
```

op:

After y = x++ : x = 11, y = 10

After y = ++x : x = 12, y = 12

After y = x-- : x = 11, y = 12

After y = --x : x = 10, y = 10

Size of () and & operator:

Size of() gives the size of the variable

& prints address of the variable in the memory

Ex:

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

void main() {
   int a;
   float b;
   char c;
   clrscr();
```

Output:

Size of int = 4 bytes

Size of float = 4 bytes

Size of char = 1 byte

Size of double = 8 bytes

```
   printf("Size of int = %d bytes\n", sizeof(a));
   printf("Size of float = %d bytes\n", sizeof(b));
   printf("Size of char = %d byte\n", sizeof(c));
   printf("Size of double = %d bytes\n", sizeof(double));

   getch();
}
```

Relational Operators: (8 Marks):

- These operators provide the relationship between the two expressions
- If the relation is true then it returns a value 1 otherwise 0 for false relation

Ex:

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

int main() {
   int a = 10, b = 20;
   clrscr();
   printf("a = %d, b = %d\n", a, b);
   printf("\na == b : %d", a == b);   // Equal to
   printf("\na != b : %d", a != b);   // Not equal to
   printf("\na > b  : %d", a > b);    // Greater than
   printf("\na < b  : %d", a < b);    // Less than
   printf("\na >= b : %d", a >= b);   // Greater than or equal
   printf("\na <= b : %d", a <= b);   // Less than or equal

   getch();
}
```

Output:[1 = true 0 = false]

a = 10, b = 20

a == b : 0
a != b : 1
a > b : 0
a < b : 1
a >= b : 0
a <= b : 1

Logical Operators:

- The logical relationship between the two expressions are checked
- Using these operators two expressions can be joined
- After checking it provides true(1) or false(0) status

Ex:

```
#include <stdio.h>
#include <conio.h>
void main() {
   clrscr();
   printf("\n 5 > 3 && 5 < 10 : %d", 5 > 3 && 5 < 10);
// AND
   printf("\n 5 > 20 || 5 < 10 : %d", 5 > 20 || 5 < 10); //
OR
   printf("\n !(7 == 7) : %d", !(7 == 7)); // NOT
   getch();
}
```

Output:

```
5 > 3 && 5 < 10 : 1
5 > 20 || 5 < 10 : 1
!(7 == 7) : 0
```

Explanation:

- 5 > 3 && 5 < 10 → true AND true → 1
- 5 > 20 || 5 < 10 → false OR true → 1
- !(7 == 7) → !(true) → 0

iv) Bitwise Operators:

C support 6 bit operators

Operators	Meaning
>>	Right shift
<<	Left shift
^	Bitwise XOR (Exclusive OR)
~	One's Complement
&	Bitwise AND
|	Bitwise OR

Bitwise Operators

1. & (Bitwise AND)

- Compares each bit of two numbers.
- Result bit = 1 only if both bits are 1, otherwise 0.
- Example:
- 5 & 3 → 0101 & 0011 = 0001 → 1

2. | (Bitwise OR)

- Compares each bit of two numbers.
- Result bit = 1 if at least one bit is 1, otherwise 0.
- Example:
- 5 | 3 → 0101 | 0011 = 0111 → 7

3. ^ (Bitwise XOR – Exclusive OR)

- Compares each bit of two numbers.
- Result bit = 1 if bits are different, 0 if same.
- Example:
- 5 ^ 3 → 0101 ^ 0011 = 0110 → 6

4. ~ (Bitwise NOT / One's Complement)

- Flips all bits of a number (0 → 1, 1 → 0).
- Works on a single operand (unary).
- In signed integers, it gives the negative (2's complement form).
- Example:
- ~5 → 0101 → 1010 (in 2's complement, = -6)

5. << (Left Shift)

- Shifts all bits of a number to the left by given positions.
- Each shift left multiplies the number by 2.
- Example:
- 5 << 1 → 0101 → 1010 = 10

6. >> (Right Shift)

- Shifts all bits of a number to the right by given positions.
- Each shift right divides the number by 2.
- Example:
- 5 >> 1 → 0101 → 0010 = 2

Example:

```
 #include <stdio.h>
#include <conio.h>
int main() {
   int a = 5, b = 3;   // binary: a=0101, b=0011
   clrscr();
   printf("a = %d, b = %d\n", a, b);

   printf("\n a & b  = %d", a & b);   // Bitwise AND
   printf("\n a | b  = %d", a | b);   // Bitwise OR
   printf("\n a ^ b  = %d", a ^ b);   // Bitwise XOR
   printf("\n ~a = %d", ~a);      // One's Complement
   printf("\n a << 1 = %d", a << 1);  // Left shift
   printf("\n a >> 1 = %d", a >> 1);  // Right shift

   getch();
}
```

Output:

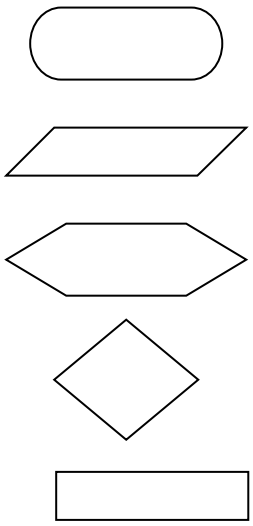
```
a = 5, b = 3
a & b  = 1
a | b  = 7
a ^ b  = 6
~a     = -6
a << 1 = 10
a >> 1 = 2
```

Explanation (for a = 5 (0101) and b = 3 (0011)):

- AND (&) → 0101 & 0011 = 0001 → 1
- OR (|) → 0101 | 0011 = 0111 → 7
- XOR (^) → 0101 ^ 0011 = 0110 → 6

- NOT (~) → ~0101 → flips all bits → -6 (in 2's complement)
- Left Shift (<< 1) → 0101 → 1010 = 10
- Right Shift (>> 1) → 0101 → 0010 = 2

Algorithm & Flow chart:



Start & Stop

Read & Write

For Loop

Condition Checking

Calculation

Algorithm & Flow chart:

Program:

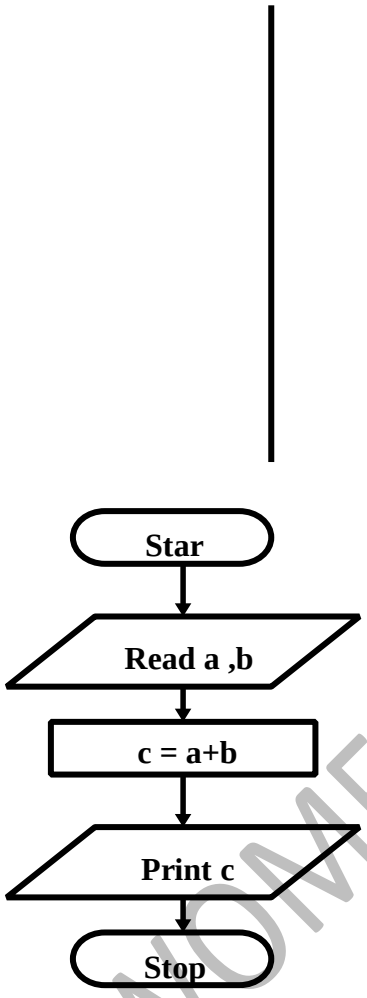
main() { int a, b, c; clrscr(); printf("Enter the

value of A \n"); scanf("%d", &a); printf("Enter the value of B \n"); scanf("%d", &b); c = a+b; printf("Addtion of Two Numbers:%d",c); gethc(); }

Algorithm:

1. Start the Program
2. Declare the variable a, b and c
3. Read the Value of a
4. Read the Value of b
5. Add two numbers and store this value to c
6. Print the Value of c
7. Stop the Program

Flowchart:



Structure of a C Program: (8 Marks):

A C program comprises of the following sections

Include Header File Sections
Global Declaration section

/* Comments */
main()
{
/* comments */
Declaration Part
Executable Part
}
User defined functions
{
}

a) Include Header File Section:

- Each header file by default is extended with .h
- The file should be included using # include

directive Ex:

#include <stdio.h>

include <conio.h>

b) Global Declaration:

- These variables must be declared outside of all the function

c) Function main():

- C language must contain main() function
- The program execution must start with "{" and "}"

d) Declaration part:

- The declaration part declares the entire variable that are used in execution

e) Execution Part:

- This part contains the statements following the declaration of the variables

f) User defined Functions:

- The functions defined by the user are called user-defined functions

g) Comments:

- Comments are nothing but some kind of statements which are placed between the delimiter / * and * /

Ex:

```
#include <stdio.h>
#inlcude <conio.h>
main() {
 int a, b, c;
clrscr();
printf("Enter the value of A \n");
scanf("%d", &a);
printf("Enter the value of B \n");
scanf("%d", &b); c = add(a,b);
printf("C value is =%d", c);
```

Output:

Enter the value of A

20

Enter the Value of B

30

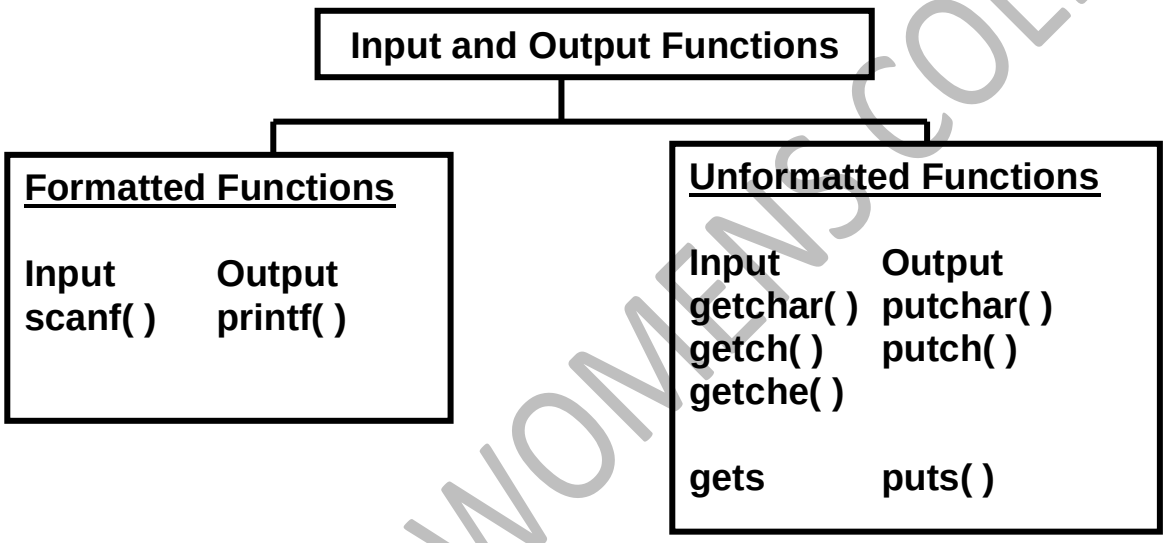
C value is = 50

```
}
add(int a, int b)
{
int c;
c = a+b;
Return c;
}
```

Input and Output in C:

There are number of I/O functions in C. The input/output functions are classified into two types:

i) Formatted Functions ii) Unformatted Functions



i) Formatted Functions: (8 Marks):

- The formatted I/O functions read and write all types of data values.
- They required conversion symbol to identify the data type.

a) The printf( ) Function:

- This printf( ) function prints all types of data values to the console
- It requires type conversion symbol and variable name to print the data
- The conversion symbol & variable should be the same number

Ex:

```
main( ) { Int x=10; flaot y=20.2; char z='H'
printf("%d \t %f \t %c " , x, y, z);
}
```

Output: H

10 20.2

b) The scanf( ) function:

- The scanf() Function reads all types of data Values
- It is used for runtime assignment of variables
- It requires type conversion symbol to identify the data to read

Syntax:

```
scanf("%d %f %c", &x, &y, &z);
```

Ex:

```
main() { int x; float y; char z; printf("Enter the value of
x, y and z\n"); scanf("%d %f %c", &x, &y, &z);
printf("%d %f %c", x, y, z);
getch();
}
```

Output:

Enter the valueof x, y and z

10

20.2 H

10 20.2 H

Escape sequence:

\n New Line \t Horizontal tab, space

\b Back space \v Vertical tab

\a alert

Ex:

```
#include <stdio.h>
int main() {
   printf("Warning!\a\n");
   return 0;
}
```

ii) Unformatted Functions: (8 Marks)

- The Unformatted i/o functions only work with the character data type.
- They do not required conversion symbol for identification of data type.
- Because they work only with character data type.

C has three types of unformatted I/O Functions:

a) Character I/O b) String I/O c) File I/O

a) Character I/O:

1. getchar()

Reads a single character from the keyboard (shows it when you type).

```
#include <stdio.h>
int main() {
   char ch;
   printf("Enter a character: ");
   ch = getchar();      // input shown on screen
   printf("\nYou entered: ");
   putchar(ch);         // output the same character
   return 0;
}
```

☞ If input = A

Output:

```
Enter a character: A
You entered: A
```

2. putchar()

- Prints a single character.

```
#include <stdio.h>
int main() {
   putchar('H');
```

```
    putchar('\n');
    putchar('I');
    return 0;
}
```

Output:

H

I

3. getch()

- Reads a single character without showing it.

```
#include <stdio.h>
#include <conio.h>
int main() {
    char ch;
    printf("Press a key: ");
    ch = getch();     // input hidden
    printf("\nYou pressed: %c", ch);
    return 0;
}
```

☞ If you press X, it won't show while typing.

Output:

Press a key:

You pressed: X

4. getche()

- Reads a single character and shows it immediately.

```
#include <stdio.h>
#include <conio.h>
int main() {
```

```
    char ch;
    printf("Press a key: ");
    ch = getche();   // input visible
    printf("\nYou pressed: %c", ch);
    return 0;
}
```

☞ If you press X

Output:

Press a key: X

You pressed: X

5. putch()

- Prints a single character (non-standard, from <conio.h>).

```
#include <stdio.h>
#include <conio.h>
int main() {
    putch('A');
    putch('\n');
    putch('B');
    return 0;
}
```

Output:

A

B

b) String I/O:

6. gets()

- Reads a string (multiple characters) until Enter is pressed.

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

```
int main() {
   char name[20];
   printf("Enter your name: ");
   gets(name);   // input string
   printf("Hello, ");
   puts(name);   // output string
   return 0;
}
```

☞ If input = John

Output:

Enter your name: John

Hello, John

Note: gets() is unsafe (can cause buffer overflow). In modern C, fgets() is recommended.

7. puts()

- Prints a string with a newline at the end.

```
#include <stdio.h>
int main() {
   puts("Hello Students!");
return 0;
}
```

Output:

Hello Students!

8. cgets() (Rare, Turbo C only)

- Reads a string from console into a buffer.
- Needs a buffer where the first byte = buffer size.
- Very old function, not used in modern compilers.

EX:

```
#include <stdio.h>
#include <conio.h>
int main() {
```

```
    char str[50];
    str[0] = 48;     // max characters to read
    printf("Enter a string: ");
    cgets(str);
    printf("You entered: %s", str+2); // actual string starts from str[2]
    return 0;
}
```

☞ If input = Hello

Output:

Enter a string: Hello

You entered: Hello

Function means:

getchar() → Input 1 char (shows on screen).

putchar() → Output 1 char.

getch() → Input 1 char (hidden).

getche() → Input 1 char (visible).

putch() → Output 1 char (Turbo C)

gets() → Input string.

puts() → Output string.

cgets() → Input string (old Turbo C).

Commonly used Library Functions:

1. clrscr( ) :

- This function is used to clear the screen
- It clears the previous output from the screen and display the current program output
- It is defined in conio.h

Syntax: clrscr( );

2. exit( ) :

- This is terminates the program
- It is defined in process.h header file

Syntax: exit( );

3. sleep( ) :

- This function pauses the execution of the program for a given number of seconds
- The number of seconds is to be enclosed between the parenthesis
- It is defined in dos.h header file

Syntax: sleep(1);

```
#include <stdio.h>
#include <unistd.h>   // for sleep()

int main() {
   printf("Start\n");
   sleep(3);   // wait for 3 seconds
   printf("End after 3 seconds\n");
   return 0;
}
```

o/p:
Start
(3 second pause...)
End after 3 seconds

4. system( ) :

- This function is helpful in executing the different dos commands
- It returns 0 on success and -1 on failure

Syntax: system("command");

- The command should be enclosed within the double quotation marks

Example Function:

- system("dir"); → runs a command (os),(here it show the lists files).
- system("calc"); → opens calculator.
- system("cls"); → clears the screen.

EX:

```
#include <stdio.h>

#include <stdlib.h>   // for system()

int main() {

   printf("Opening Calculator...\n");
```

```
    system("calc");   // Opens Calculator in Windows
    printf("Opening Notepad...\n");
    system("notepad"); // Opens Notepad in Windows
    return 0;
}
```

O/p:

Opening Calculator...

At this point, Windows Calculator app will open.

After you close the Calculator, the program continues:

Opening Notepad...

Then Notepad will open.

UNIT- II

Decision Statements:

Decision statements in C are control structures that allow a program to make choices and execute certain parts of the code depending on whether a specified condition evaluates to true or false.

C Language supports the control statements as listed below:

1) The if statement 2) The if-else statement

The Nested if-else statement

The switch ( )case statement

The if statement:

The keyword if to execute a set of command lines or one command line when the logical condition is true

Syntax:

```
if (test condition)
 {
 statements;
 }
```

i) The statements are executed only when the condition is true ii) In case condition is false the compiler skips the lines inside the if block iii) Test condition always enclosed with pair of parenthesis

iv) The Conditional statement should not terminated with semi-colon(;)

Ex 1:

```
#include <stdio.h>
int main() {
   int num = 10;
   if (num > 0) {
      printf("Number is positive.\n");
   }
   return 0;
}
```

Output1:

Number is positive.

Output1:

Eligible to vote.

Ex 2:

```
#include <stdio.h>
int main() {
   int age = 20;
   if (age >= 18) {
      printf("Eligible to vote.\n");
   }
   return 0;
}
```

The if-else statement:

- The if-else statement takes care of true as well as false conditions.
- It has two blocks.

1. if block it is executed only when the condition is true
2. else block it is executed only when the condition is false

- The else statement cannot be used without if.
- No multiple else statement are allowed with one if

Syntax:

```
if (test condition)
{
Statemnet1;
Statement2;
} else
{
Statement3;
Statement4;
}
```

Output 1:

Odd number.

Output 2:

Fail.

Example 1:

```
#include <stdio.h>
int main() {
   int num = 7;
   if (num % 2 == 0) {
      printf("Even number.\n");
   } else {
      printf("Odd number.\n");
   }
   return 0;
}
```

Example 2:

```
#include <stdio.h>
int main() {
   int marks = 45;
   if (marks >= 50) {
      printf("Pass.\n");
   } else {
      printf("Fail.\n");
   }
   return 0;
}
```

The Nested if-else statement:

- Number of logical conditions are checked
- If any logical condition is true the compiler executes the block followed by if condition
- Otherwise it skips and executes else block
- Nested if – else can be chained with one another

Syntax:

```
if(condition1) {
Statements; }
elseif(condition2)
{
Statements; }
elseif(condion3) {
Statements; } else
{
Statements;
}
```

Output 1:

Positive and Even.

Output 2:

Excellent Performance.

Example 1:

```
#include <stdio.h>
int main() {
   int num = 12;
   if (num > 0) {
      if (num % 2 == 0) {
         printf("Positive and Even.\n");
      }
   }
   return 0;
}
```

Example 2:

```
#include <stdio.h>
int main() {
   int marks = 85;
   if (marks >= 50) {
      if (marks >= 80) {
         printf("Excellent
Performance.\n");
      }
   }
   return 0;
}
```

The if-else statement:

- Used for multiple conditions.

Example1:

```
#include <stdio.h>
int main() {
   int marks = 65;
   if (marks >= 90) {
      printf("Grade A\n");
   } else if (marks >= 75) {
      printf("Grade B\n");
   } else if (marks >= 50) {
```

```
        printf("Grade C\n");
    } else {
        printf("Fail\n");
    }
    return 0;
}
```

Output:

Grade C

Example 2:

```
#include <stdio.h>
int main() {
    int temp = 30;
    if (temp > 40) {
        printf("Very Hot\n");
    } else if (temp > 25) {
        printf("Warm\n");
    } else if (temp > 10) {
        printf("Cool\n");
    } else {
        printf("Cold\n");
    }
    return 0;
}
```

Output 2 :

Warm

Unconditional Statements:

The break statement:

- The keyword break allows the programmers to terminate the loop

The continue statement:

- It is exactly opposite to break statement.
- The continue statement is used for continuing next iteration of loop statements
- When it occurs in the loop it does not terminate, but it skips the statements after this statement The goto statement:
- This statement passes control anywhere in the program
- Control is transferred from one part to another

Syntax: goto label;

Ex: goto s;

The switch statement:

- The Switch statement is a multi-way branching statement.
- The switch statement requires only one argument of any data type, which is checked with number of case options
- If the value match with case constant, this particular case statement is executed
- The switch( ) statement is useful for writing menu driven program
- Test condition should be pair of parenthesis and it should not be terminated with semicolon

Syntax:

switch(variable or expression)

{ case constant1: statements;

case constant2: statements;

default: statements;

}

Example:

#include <stdio.h>

```
int main() {
   int day = 2;
   switch (day) {
      case 1: printf("Monday\n"); break;
      case 2: printf("Tuesday\n"); break;
      case 3: printf("Wednesday\n"); break;
      default: printf("Invalid day\n");
   }
   return 0;
```

Output1:

Tuesday

Looping Statements:

What is Loop?

A loop is defined as a block of statements which are repeatedly executed for certain number of times

The c language supports three types of loop control statements

i) The for loop ii) The while loop iii)The do-while loop

i) The for Loop:

The for loop allows to execute a set of instruction until a certain condition is satisfied

A for loop is used when we know how many times the code should repeat.

Syntax:

```
for(initialization; condition; increment/decrement) {
// code to repeat
}
```

Example 1:

```
#include <stdio.h>
int main() {
   int i;
```

```
    for(i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

1
2
3
4
5

Explanation:

- i = 1 → start
- i <= 5 → condition true → print i
- i++ → increase i by 1 each time
- Stops when i = 6 (condition false

Example2:

```
main( ) {
int i;
clrscr();
for(i=1; i<5; i++)
{
printf("KW College");
} getch(); }
```

Example 3:

```
 main( ) {
int i; clrscr(); for(i=1; i<=10; i++)
{
printf("%d", i); } getch(); }
```

example 2 output:

[1]= KW College

[2]= KW College

[3]= KW College

[4]= KW College

[5]= terminate the program

Example 3 Output:

1
2
3
4
5
6
7
8
9
10

Nested for Loop:

One for loop can be inserted into another for loop

The number of iterations =

The number of iterations in the outer loop X number of iterations in the inner loop

Syntax:

```
for( initialize counter; test condition; re-evaluation parameter)
        {
for( initialize counter; test condition; re-evaluation parameter)
        {
        statements;
        }
        }
```

Example:

```
#include <stdio.h>
#include <conio.h>
int main() {
   int i, j;
   clrscr();
   for(i = 1; i <= 3; i++) {          // outer loop runs 3 times
      for(j = 1; j <= 2; j++) {       // inner loop runs 2 times
         printf("I Like KWC\n");     // statement executed 3 x 2 = 6 times
      }
   }
   getch();
   return 0;
}
```

Output:

```
[1,1] = I Like KWC
[1,2] = I Like KWC
[1,3] = I Like KWC
[2,1] = I Like KWC
[2,2] = I Like KWC
[2,3] = I Like KWC
[3,1] = I Like KWC
[3,2] = I Like KWC
[3,3] = I Like KWC
```

The while loop:

A while loop keeps executing as long as the condition is true.

- Used when we don't know exact repetitions.

Syntax:

```
while(condition) {
  // code to repeat
}
```

- The test condition may be any expression
- The loop statements will be executed till the condition is true If the condition is true, then the body of the loop is executed.
- When the condition becomes false the execution will be out of the loop

Example 1:

```
#include <stdio.h>

int main() {
   int i = 1;
   while(i <= 5) {
      printf("%d\n", i);
      i++;
   }
   return 0;
}
```

Output:

```
1
2
3
4
5
```

Explanation:

- Start with i = 1.
- Check i <= 5.
- If true → print i.
- Increase i → check again.
- Stops when i = 6 (condition false).

Example:

```
main( ) {
 int i=1;
clrscr( );
 while(i<5)
{
printf("I Like KWC");
i++;
 }
} getch();
}
```

Output:

[1]= I Like KWC

[2]= I Like KWC

[3]= I Like KWC

[4]= I Like KWC

[5]= condition becomes false, so terminate the program

iii) The do-while loop:

A do…while loop is like while, but it runs at least once, even if the condition is false.

- The difference between the while and do-while loop is in the place where the condition is to be tested
- The do-while statement, the condition is checked at the end of the loop
- The do-while loop will execute at least one time even if the condition is false initially
- This loop executes until the condition becomes false

Syntax:

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

Example Program:

```
#include <stdio.h>
int main() {
   int i = 1;
   do {
      printf("%d\n", i);
      i++;
   } while(i <= 5);
   return 0;
}
```

Output:

1
2
3
4
5

Explanation:

- First run → print i = 1
- Then check condition i <= 5.
- If true → run again.
- If false → stop

Example 2:

```
main( ) {
int i=1;
clrscr( );
do {
printf("I Like KWC");
i++; } while(i<5); } getch();
}
```

Output:

[1]= I Like KWC

[2]= I Like KWC

[3]= I Like KWC

[4]= I Like KWC

[5]= condition becomes false, so terminate the program

THREE TYPE EXAMPLES:

Program 1: For Loop – Multiplication Table of 5

```
#include <stdio.h>
int main() {
   int i;
   printf("Multiplication Table of 5 using For Loop:\n");
   for(i = 1; i <= 5; i++) {
      printf("5 x %d = %d\n", i, 5 * i);
   }
   return 0;
}
```

Output:

Multiplication Table of 5 using For Loop:

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

Program 2: While Loop – Multiplication Table of 5

```
#include <stdio.h>
int main() {
   int i = 1;
   printf("Multiplication Table of 5 using While Loop:\n");
   while(i <= 5) {
      printf("5 x %d = %d\n", i, 5 * i);
      i++;
   }
   return 0;
}
```

Output:

Multiplication Table of 5 using While Loop:

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

Program 3: Do…While Loop – Multiplication Table of 5

```
#include <stdio.h>

int main() {

   int i = 1;

   printf("Multiplication Table of 5 using Do While Loop:\n");

   do {
```

```
    printf("5 x %d = %d\n", i, 5 * i);
    i++;
} while(i <= 5);
  return 0;
}
```

Output:

Multiplication Table of 5 using Do While Loop:

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

UNIT- III

Arrays:

Definition:

- Array is a collection of similar data types in which each element is unique one and located in separate memory locations.
- Array elements can be stored in sequential manner or stored at contiguous (continuous) memory locations.

Example:

Instead of writing 5 variables int a, b, c, d, e;, we can use one array int marks[5];.

Array Declaration:

int a[5]; float b[10]; char c[25];

Array Initialization:

int a [5] = { 10, 40, 20, 30, 50 };

Why use arrays?

- To store multiple values in one variable name.
- Easy to process using loops.
- Saves memory and reduces complexity.

Dimensional Array:

- The element of an integer array a[5] are stored in continuous memory locations.
- It is assumed that the starting location is 2000.
- Each integer element requires 2 bytes
- Hence each element appears after gap of 2 locations

Element	A[0]	A[1]	A[2]	A[3]	A[4]
Address	2000	2002	2004	2006	2008

Example:

```
# include < stdio.h >
 main( )
{
          int a [5];
          printf("Enter the Value One by One\n");
          for( i = 0 ; i < 5 ; i++)
          {
          scanf("%d", &a[i]);
          }
          printf("Entered Values are:\n")
         for( i = 0 ; i < 5 ; i++)
          {
          printf("%d", a[i]);
          }
         getch ()
         }
```

Output:

Enter the value One by One

a[0] = 20

a[1] = 30

a[2] = 10

a[3] = 40

a[4] = 50

Entered Values are:

20

30

10

40

50

One-Dimensional Arrays

Definition

A one-dimensional (1D) array is like a list of elements stored in a row.

Example: int arr[5]; → stores 5 integers.

Syntax

data_type array_name[size];

Declaration of One-Dimensional Arrays

Syntax

int arr[10]; // integer array with 10 elements

float price[20]; // float array with 20 elements

char name[30]; // character array (string)

Array index always starts from 0.

So, arr[10] → elements are arr[0] to arr[9].

Initialization of One-Dimensional Arrays

Syntax

int arr[5] = {10, 20, 30, 40, 50};

If not given all values, remaining are set to 0.

Example:

int arr[5] = {1, 2}; // arr = {1, 2, 0, 0, 0}

Example Program (1D Array)

```
#include <stdio.h>
int main() {
   int marks[5];   // Declaration
   int i;

   // Initialization
   marks[0] = 50;
   marks[1] = 60;
   marks[2] = 70;
   marks[3] = 80;
   marks[4] = 90;

   // Printing array elements
   printf("Student Marks:\n");
   for(i = 0; i < 5; i++) {
      printf("marks[%d] = %d\n", i, marks[i]);
   }
   return 0;
}
```

Output:

Student Marks:

marks[0] = 50

marks[1] = 60

marks[2] = 70

marks[3] = 80

marks[4] = 90

Explanation:

Array marks stores 5 integers.

We initialized them manually.

for loop prints all values.

Two-Dimensional Arrays

- Two-Dimensional array can be thought as a rectangular display of elements with rows & columns.
- The elements are shown in matrix form
- It is a collection of a One-Dimensional arrays, which are placed one after another.
- A two-dimensional (2D) array is like a table (rows and columns).

Example:

int matrix[2][3]; → 2 rows and 3 columns.

Syntax

data_type array_name[rows][columns];

Example:

```
main( ) {
int i, j;
int a[3][3] = { {1,2,3} ,
                {4,5,6} ,
               {7,8,9} };
clrscr( );
for (i=0; i<3 ; i++)
{
for (j=0; j<3 ; j++)
{
printf (“%d \n”, &a[i][j]);
}
printf(“\t”);
} getch();
}
```

Output:

1	2	3
4	5	6
7	8	9

Initializing Two-Dimensional Arrays

Syntax

```
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
```

Memory map of Two-Dimensional Array:

Row, Column	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]
Value	1	2	3	4	5	6	7	8	9
Address	2000	2002	2004	2006	2008	2010	2012	2014	2016

Example Program (2D Array)

```
#include <stdio.h>
int main() {
   int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
   int i, j;
   printf("Matrix Elements:\n");
   for(i = 0; i < 2; i++) {
      for(j = 0; j < 3; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

NOTE:

Outer loop (i)	Inner loop (j)	matrix[i][j]	Printed Output
i = 0	j = 0	matrix[0][0] = 1	1
i = 0	j = 1	matrix[0][1] = 2	2
i = 0	j = 2	matrix[0][2] = 3	3
	Inner loop ends, \n printed		new line
i = 1	j = 0	matrix[1][0] = 4	4
i = 1	j = 1	matrix[1][1] = 5	5
i = 1	j = 2	matrix[1][2] = 6	6
	Inner loop ends, \n printed		new line

Output:

Matrix Elements:

1 2 3

4 5 6

Explanation:

matrix[2][3] means 2 rows × 3 columns.

Nested loops print row by row.

Multi-Dimensional Arrays

Definition

A multi-dimensional array is an array with more than two dimensions.

Example: 3D array is like a cube.

Syntax

data_type array_name[size1][size2][size3]...[sizeN];

Example Program (3D Array)14

cube[0][0][0] = 1

cube[0][0][1] = 2

cube[0][1][0] = 3

cube[0][1][1] = 4

cube[1][0][0] = 5

cube[1][0][1] = 6

cube[1][1][0] = 7

cube[1][1][1] = 8

Explanation:

3D array = 2 blocks, each block has 2 rows × 2 columns.

Triple nested loop prints all elements.

Summary

Array → Collection of same type values in continuous memory.

Types of Arrays in C:

1. 1D Array → list

2. 2D Array → table (rows × columns)

3. Multi-Dimensional → cube, etc

Character Arrays and String Handling Functions

Character arrays and strings are fundamental for handling text in C programming, though strings are not a built-in data type like in other languages. A C string is a null-terminated character array, meaning it's a sequence of characters stored in memory that ends with a null character ('\0'). This terminator signals the end of the string to functions that process it.

Declaring and initializing string variables

There are several ways to declare and initialize a string variable, which is essentially a char array.

1. Assigning a string literal

This is the most common method. The compiler automatically adds the null terminator.

- With a specified size: Ensure the size is large enough to hold the string plus the '\0' character.

char name[10] = "Brenda";

- Without a specified size: The compiler will automatically determine the size of the array based on the length of the string literal.

char message[] = "Hello"

2. Assigning character by character

You can initialize a string by listing its characters individually. In this case, you must explicitly include the null terminator.

char city[] = {'D', 'e', 'l', 'h', 'i', '\0'};

Reading strings from the terminal

Several functions can be used to get string input from a user.

1. scanf()

This function reads a string using the %s format specifier.

- Limitation: It stops reading at the first whitespace character (space, tab, or newline), making it unsuitable for reading sentences.
- Syntax: scanf("%s", string_name);
- Note: The ampersand (&) is not needed for arrays, as the array name is already a pointer to the first element.

2. fgets()

- This is the safest and most recommended way to read a string, as it prevents buffer overflow by limiting the number of characters read.
- Reads an entire line: fgets() reads input until it encounters a newline character or reaches the specified buffer size.
- Syntax: fgets(char_array, size, stdin);
    - char_array: The character array where the string will be stored.
    - size: The maximum number of characters to read.
    - stdin: Specifies that input should be read from the standard input stream (keyboard).

3. gets() (Deprecated)

- Reads an entire line: Like fgets(), gets() reads an entire line of input, including spaces.

- Unsafe: It does not perform bounds checking and can easily cause a buffer overflow. It is no longer part of the C standard and should be avoided.

Writing strings to the screen

Strings can be displayed on the console using standard output functions.

1. printf()

This function uses the %s format specifier to print a string stored in a character array. It prints characters until it finds the null terminator.

- Syntax: printf("%s", string_name);

2. puts()

This function prints a string followed by a newline character.

- Syntax: puts(string_name);

Comparison of two strings

You cannot use the == operator to compare strings directly, as this would compare their memory addresses, not their contents.

strcmp()

The standard library function strcmp() is used to compare two strings lexicographically (based on dictionary order).

- Include: <string.h>
- Syntax: int strcmp(const char *str1, const char *str2);
- Return values:
  - 0: If the strings are identical.
  - Negative value: If str1 is lexicographically less than str2.
  - Positive value: If str1 is lexicographically greater than str2.

Example:

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

int main() {
   char s1[] = "apple";
   char s2[] = "apricot";
   int result = strcmp(s1, s2);
```

```
    if (result < 0) {
        printf("s1 is less than s2\n");
    } else if (result > 0) {
        printf("s1 is greater than s2\n");
    } else {
        printf("s1 is equal to s2\n");
    }
    return 0;
}
```

String handling functions

To use these functions, you must include the <string.h> header file.

Common functions

- strlen(str): Returns the number of characters in the string, excluding the null terminator.
- strcpy(dest, src): Copies the string src to the string dest. The destination array must be large enough to hold the source string to prevent buffer overflow.
- strcat(dest, src): Appends the string src to the end of the string dest.
- strcmp(str1, str2): Compares two strings. Returns 0 if equal.
- strncpy(dest, src, n): Copies at most n characters from src to dest. It is safer than strcpy.
- strncat(dest, src, n): Appends at most n characters from src to dest.
- strchr(str, ch): Returns a pointer to the first occurrence of the character ch in the string str.
- strstr(haystack, needle): Returns a pointer to the first occurrence of the substring needle in the string haystack.
- strrev(str): Reverses the string str in place. Note that this is a non-standard function, though it is available in some compilers.

UNIT - IV

Functions

Definition of Functions: A self-contained block of code designed to perform a specific task. Using functions helps break complex problems into smaller, manageable parts, making code more modular, readable, and reusable.

- **Elements of User-Defined Functions:**
- Function declaration (or prototype): Informs the compiler about the function's name, return type, and parameters before it is used. It ends with a semicolon.
- Function definition: Provides the actual body of the function, which contains the code to be executed.
- Function call: The process of invoking or executing a function. It transfers program control to the called function.

Return Values and Their Types:

- A function can return a single value to the calling code using the return statement.
- The return_type in the function's declaration and definition specifies the data type of the value it will return (e.g., int, float, char).
- If a function does not return a value, its return type is void.

Syntax example :

```
// Function Declaration
int add(int a, int b);

// Function Call
int main() {
  int result = add(5, 3); // Calls the 'add' function
  return 0;
}

// Function Definition
int add(int a, int b) {
  return a + b; // Returns an integer value
}
```

Category of functions

Functions can be categorized based on whether they take arguments and/or return a value:

- No Arguments and No Return Value

- Arguments but No Return Value
- No Arguments but a Return Value
- Arguments and a Return Value

Functions in programming languages like C can be categorized into four types based on whether they accept arguments and whether they return a value.

1. No arguments and no return value

- Description: This type of function does not receive any data from the calling function and does not send any data back. Communication is completely independent.
- Best for: Performing a self-contained task, such as printing a menu or a line separator.

Example program :

```
#include <stdio.h>

// Function declaration
void print_message();

int main() {
   print_message(); // Function call
   return 0;
}

// Function definition
void print_message() {
   printf("Hello, this is a function with no arguments and no return value.\n");
}
```

Output: Hello, this is a function with no arguments and no return value.

2. Arguments but no return value

- Description: This function accepts data from the calling function to perform its task, but does not return any value. It's a one-way communication channel.
- Best for: Performing an action that depends on external data, like printing a formatted output or performing an operation on variables passed to it.

Example program :

```
#include <stdio.h>

// Function declaration
void print_sum(int a, int b);
```

```
int main() {
   int x = 5, y = 10;
   print_sum(x, y); // Function call with arguments
   return 0;
}

// Function definition
void print_sum(int a, int b) {
   int sum = a + b;
   printf("The sum of %d and %d is %d.\n", a, b, sum);
}
```

Output: The sum of 5 and 10 is 15.

No arguments but a return value

- Description: This function does not receive any data from the calling function, but it computes a value and returns it. The calling function receives data, but does not provide any.
- Best for: Encapsulating data retrieval or calculation that doesn't need external input, such as reading user input and returning it, or getting a value from a global variable.

Example program :

```
#include <stdio.h>

// Function declaration
int get_number();

int main() {
   int my_number;
   my_number = get_number(); // Function call, assigns returned value
   printf("The number received from the function is: %d\n", my_number);
   return 0;
}

// Function definition
int get_number() {
   int num;
   printf("Enter a number: ");
   scanf("%d", &num);
   return num; // Returns the entered integer
}
```

Output (assuming the user enters 25):

- Enter a number: 25
- The number received from the function is: 25

Arguments and a return value

- Description: This is the most versatile type of function, as it allows for two-way communication. It accepts data from the caller, processes it, and returns a result.
- Best for: The majority of computational tasks, such as mathematical calculations, string manipulations, or any operation where both input and a result are necessary.

Example program :

```
#include <stdio.h>

// Function declaration
int multiply(int a, int b);

int main() {
   int result;
   int x = 4, y = 6;
   result = multiply(x, y); // Call with arguments, store returned value
   printf("The product of %d and %d is %d.\n", x, y, result);
   return 0;
}

// Function definition
int multiply(int a, int b) {
   return a * b; // Returns the product
}
```

Output: The product of 4 and 6 is 24.

Recursion

Recursion is a technique where a function calls itself to solve a problem that can be broken into smaller, similar subproblems. A recursive function needs a base case to stop the recursion and a recursive step that moves towards the base case. An example is calculating factorial, where the function calls itself with a decreasing number until it reaches the base case of 0 or 1.

Recursion is a technique where a function calls itself to solve a smaller version of the same problem. This approach requires careful handling to prevent infinite loops and manage memory efficiently.

Key components of a recursive function

Every recursive function must have two main parts to work correctly:

- **Base case:** The condition that terminates the recursion. It provides a direct solution for the smallest possible input, without making any further recursive calls. Without a base case, the function would call itself indefinitely, leading to a stack overflow error.
- **Recursive step:** The part of the function where it calls itself with a modified, simpler input. This step breaks the problem down, moving it closer to the base case.

Example 1: Factorial calculation

The factorial of a non-negative integer n is defined as n!=n×(n−1)!, with 0!=1 as the base case.

C program:

```
#include <stdio.h>

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

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

You can see how factorial(4) works in detail in the referenced web document.

Example 2: Fibonacci sequence

The Fibonacci sequence is defined by F(n)=F(n−1)+F(n−2), with base cases F(0)=0 and F(1)=1

C program:

```
#include <stdio.h>

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

int main() {
   int n = 6;
   printf("The Fibonacci sequence up to %d terms:\n", n);
   for (int i = 0; i < n; i++) {
      printf("%d ", fibonacci(i));
   }
   return 0;
}
```

The referenced web document explains how fibonacci(4) works through recursive calls and summing the results.

Example 3: Reversing a string

This function recursively prints a string in reverse by printing the last character after the recursive call has processed the rest of the string.

C program:

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

// Recursive function to reverse and print a string
void reverse_string(char* str) {
   if (*str == '\0') { // Base case
```

```
        return;
    }
    reverse_string(str + 1); // Recursive step
    printf("%c", *str);      // Print after recursive call returns
}

int main() {
    char my_string[] = "hello";
    reverse_string(my_string);
    return 0;
}
```

Output: olleh

Recursion and the call stack

Each recursive call adds a stack frame to the call stack, storing local data. Deep recursion can lead to a "stack overflow" error if memory is exhausted. This overhead can make recursion slower than iterative solutions.

Recursion vs. iteration

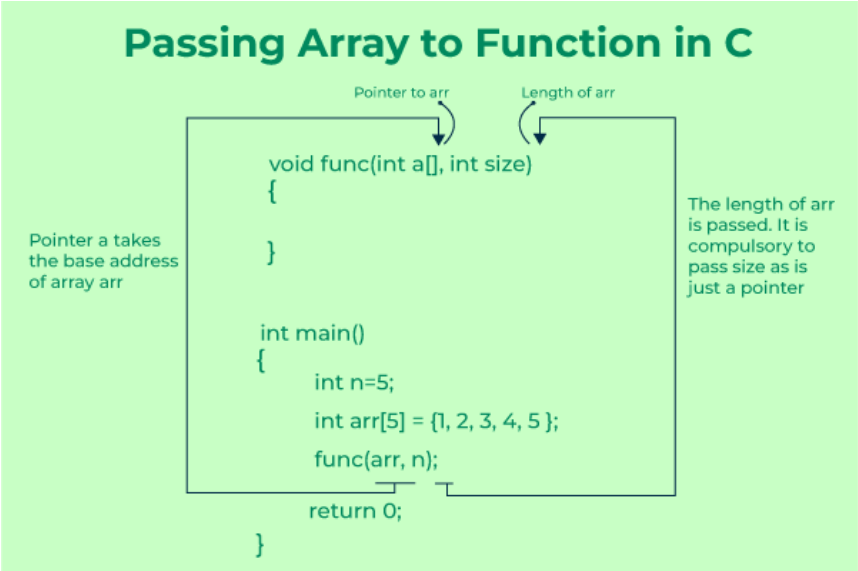
Aspect	Recursion	Iteration
Control	Function calls itself until a base case.	Code block repeats until a condition is false.
Termination	Base case.	Loop condition.
Memory	Uses call stack; potentially high memory usage.	Lower memory usage.
Performance	Can be slower due to overhead.	Generally faster and more efficient.
Code Length	Can be concise.	Can be longer.
Readability	Intuitive for naturally recursive problems.	Easier to trace and debug for many tasks.

When to use recursion

Recursion is suitable for problems with inherent recursive structures, such as tree and graph traversals (like Depth-First Search), Divide and Conquer algorithms (Merge Sort, Quick Sort), and backtracking problems. It can also be preferred when the recursive solution is significantly clearer than an iterative one.

Passing arrays and strings to functions

In C, arrays and strings are typically passed to functions as pointers to their first element, meaning changes within the function affect the original array or string. Strings are treated as character arrays ending with a null character \0. Arrays can be passed using sized, unsized, or pointer notation in the function signature.



In C, arrays and strings are handled differently from simple data types like int or float when passed to functions. This is because C does not copy the entire array when it's passed. Instead, the function receives a pointer to the first element of the array. This is known as "pass by reference" and has important implications for how your data is handled.

Key concepts

When passing an array to a function, the array name "decays" into a pointer to its first element. This means the function works with the original array in memory, so any modifications made inside the function are permanent.

Function signatures for receiving an array can be declared using either array notation (int arr[]) or pointer notation (int *arr). Since the array size information is lost when it decays to a pointer, it's crucial to pass the size as a separate parameter to avoid accessing memory outside the array's bounds.

Example: Passing arrays to a function

The following code demonstrates passing an integer array to a function doubleArrayElements that modifies the array's contents and a printArray function:

#include <stdio.h>

```
// Function to double every element of an array
void doubleArrayElements(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      arr[i] = arr[i] * 2; // This modifies the original array
   }
}

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


int main() {
   int numbers[] = {1, 2, 3, 4, 5};
   int size = sizeof(numbers) / sizeof(numbers[0]);
   printf("Original array: ");
   printArray(numbers, size);
doubleArrayElements(numbers, size);
   printf("Modified array: ");
   printArray(numbers, size);
   return 0;
}
```

The output shows the array being modified by the function:

Original array: 1 2 3 4 5

Modified array: 2 4 6 8 10

Passing strings to a function

A string in C is a character array ending with a null character (\0). Passing a string to a function follows the same principles as passing other arrays: the function receives a pointer to the first character and uses the null terminator to determine the string's end.

Example: Passing strings to a function

This example illustrates passing a string to printString and reverseString functions:

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

// Function to print a string
```

```
void printString(char str[]) {
   printf("The string is: %s\n", str);
}

// Function to reverse a string in-place
void reverseString(char str[]) {
   int length = strlen(str);
   for (int i = 0; i < length / 2; i++) {
      char temp = str[i];
      str[i] = str[length - i - 1];
      str[length - i - 1] = temp;
   }
}
int main() {
   char greeting[] = "Hello";

   printf("Original string: ");
   printString(greeting);

   reverseString(greeting);

   printf("Reversed string: ");
   printString(greeting);

   return 0;
}
```

The output demonstrates the string being printed and then reversed:

Original string: The string is: Hello

Reversed string: The string is: olleH

Scope, visibility, and lifetime of variables

These concepts define where variables can be accessed and how long they exist.

- Scope is the region of the program where a variable is accessible. Common scopes include local (within a function or block), global (outside all functions), and function prototype scope.
- Visibility determines if an accessible variable can actually be used in a specific code region, considering potential naming conflicts that hide variables from wider scopes.
- Lifetime is the duration a variable occupies memory. Automatic lifetime variables are created and destroyed within their block, while static lifetime variables exist for the program's entire duration (like global variables or those declared with static).

Structures

A struct is a user-defined data type that allows you to combine variables of different data types into a single unit. Each member of a structure has its own unique memory location.

Defining a structure

The struct keyword is used to define a structure, followed by a tag name and a list of members enclosed in curly braces.

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

Declaring structure variables

You can declare variables of a structure type in two common ways:

1. With the definition:

```
struct Point {
    int x;
    int y;
} p1; // Variable p1 is declared here
```

2. Separately, like a standard variable:

```
struct Point {
    int x;
    int y;
};
int main() {
    struct Point p1; // Variable p1 is declared here
}
```

Accessing structure members

Use the dot (.) operator to access a member of a structure variable. If you have a pointer to a structure, use the arrow (->) operator.

```
struct Student s1;

s1.rollNumber = 101;

// To access with a pointer

struct Student *ptr = &s1;

ptr->marks = 95.5;
```

Unions

A union is a user-defined data type similar to a structure, but all of its members share the same memory location. This means a union variable can only store one member's value at a time. Unions are primarily used for memory efficiency, especially in embedded systems.

Defining a union

You define a union using the union keyword, following a similar syntax to a structure.

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

Declaring union variables

Declaring union variables is the same as declaring structure variables.

```
union Data d1;
```

Accessing union members

Members are accessed using the dot (.) or arrow (->) operator, but you must only access the member that was most recently written to. Writing to one member overwrites the values of the others.

```
union Data d1;

d1.i = 10; // The integer member is active

//... some code
```

d1.f = 3.14; // The float member is now active, overwriting the integer

In C programming, both structures and unions are used to group different types of data under a single name, but they behave in different ways. The main difference lies in how they store data.

The below table lists the primary differences between the C structures and unions:

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

Bit fields

Bit fields are a feature that allows you to specify the number of bits a structure or union member should occupy, enabling highly memory-efficient storage. This is useful for storing flags, status bits, or other data that requires only a few bits.

Defining a bit field

Within a struct or union, declare a bit field member by adding a colon (:) followed by the number of bits.

```
struct StatusFlags {
    unsigned int error : 1;
    unsigned int ready : 1;
    unsigned int mode : 2;
};
```

In this example, the StatusFlags structure will likely occupy only a single byte of memory.

Accessing bit fields

Access bit fields using the standard dot (.) or arrow (->) operators.

```
struct StatusFlags status;
status.error = 1;  // Set the 'error' flag
if (status.ready == 1) {
    // ...
}
```

Limitations of bit fields:

- You cannot take the address of a bit field.
- Their implementation can be non-portable across different compilers.
- They are restricted to integer types.

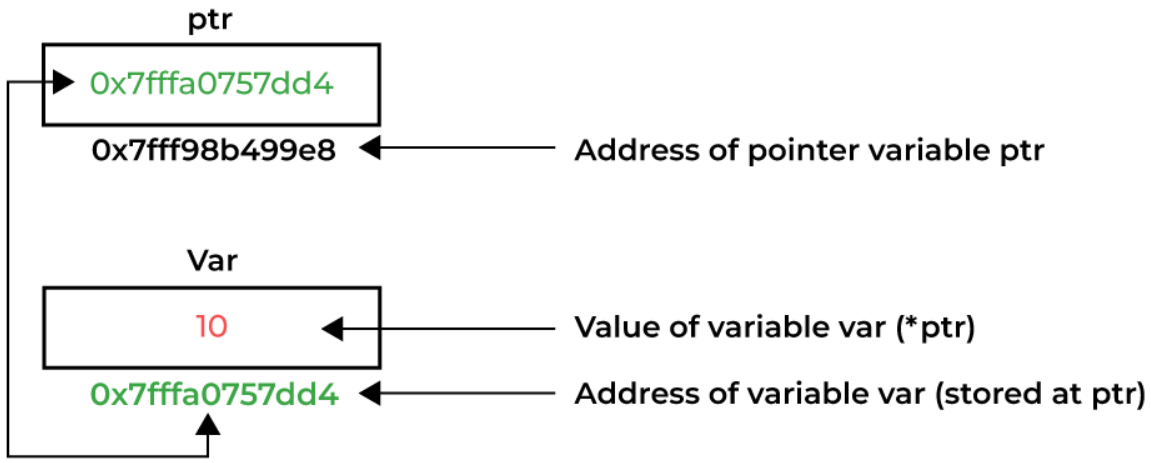
UNIT - V

Pointers in C

Pointers are a fundamental and powerful feature in C that allows for direct memory manipulation and efficient data handling. A pointer is a special type of variable that stores the memory address of another variable, rather than a value directly. This provides low-level control over memory, essential for tasks like dynamic memory allocation, array traversal, and building complex data structures like linked lists.

Understanding pointers

Imagine your computer's memory as a series of numbered boxes. Each box has a unique number, its memory address, and holds a value. A regular variable is like a box with a label and a value inside. A pointer is like a piece of paper that only contains the address of another box. By looking at the address on the paper, you can find the actual box and access its content.



Accessing the address of a variable

In C, the address-of operator (&) is used to obtain the memory address of a variable.

- Syntax: &variable_name
- Purpose: Returns the memory address where the variable_name is stored.
- Example: If int num = 25;, then &num would return the memory address of num.

Declaring pointer variables

Just like regular variables, pointers must be declared before use. When declaring a pointer, you must specify the type of data the pointer will point to, followed by an asterisk (*) and the pointer's name.

- Syntax: data_type *pointer_name;

- data_type: Indicates the type of data the pointer will point to (e.g., int, char, float).
- The asterisk indicates that the variable being declared is a pointer.

Example:

int *ptr_to_int; // Declares a pointer to an integer

char *ptr_to_char; // Declares a pointer to a character

float *ptr_to_float; // Declares a pointer to a float

Initialization of pointer variables

Initializing a pointer involves assigning a valid memory address to it. Uninitialized pointers (often called wild pointers) contain garbage values and may point to random memory locations, leading to unpredictable program behavior or crashes. You can initialize a pointer by assigning the address of an existing variable using the & operator.

- Syntax: pointer_name = &variable_name;
- Initialization at declaration: You can also declare and initialize a pointer in a single step.
- data_type *pointer_name = &variable_name;

Example:

int number = 100;

int *ptr; // Declaration

ptr = &number; // Initialization: ptr now holds the address of 'number'

Use code with caution.

Alternatively, combining declaration and initialization:

int num = 100;

int *ptr = # // Declares 'ptr' and initializes it with the address of 'num'

Use code with caution.

Note: If you don't have a valid address to assign yet, initialize the pointer to NULL to avoid it becoming a wild pointer. A null pointer doesn't point to any valid memory location.

int *safe_ptr = NULL; // Initialized to NULL

Accessing a variable through its pointer

Once a pointer has been initialized with a valid memory address, you can access or modify the value stored at that address using the dereference operator (*), also known as the indirection operator.

- Syntax: *pointer_name
- Purpose: The * operator, when used not in a declaration, refers to the value stored at the memory address the pointer is pointing to.

Example:

```
#include <stdio.h>

int main() {
   int value = 42;
   int *ptr; // Declare a pointer to an integer

   ptr = &value; // Initialize ptr with the address of 'value'

   printf("Value of 'value' directly: %d\n", value);     // Access 'value' directly
   printf("Address of 'value': %p\n", &value);           // Print address of 'value'
   printf("Value stored in 'ptr' (address): %p\n", ptr);  // Print the address stored in 'ptr'
   printf("Value of 'value' via ptr: %d\n", *ptr);         // Access value of 'value' using dereference
operator

   // Modifying the value using the pointer
   *ptr = 99; // Change the value stored at the address pointed to by ptr

   printf("New value of 'value' directly: %d\n", value);  // Check 'value' directly
   printf("New value of 'value' via ptr: %d\n", *ptr);    // Check value via ptr

   return 0;
}
```

- Sample Output: (Addresses may vary)
- Value of 'value' directly: 42
- Address of 'value': 0x7ffcd3e2a014
- Value stored in 'ptr' (address): 0x7ffcd3e2a014
- Value of 'value' via ptr: 42
- New value of 'value' directly: 99
- New value of 'value' via ptr: 99

This example demonstrates how *ptr can both retrieve the value and modify it. When *ptr = 99; is executed, the value at the memory location pointed to by ptr (which is the address of value) is changed to 99, effectively updating value itself.

Chains of pointers (pointer to a pointer)

A chain of pointers, or a pointer to a pointer, is a variable that stores the address of another pointer. This creates a chain of indirection, allowing you to access a variable through multiple layers of pointers.

Declaration

To declare a pointer to a pointer, you use an asterisk (*) for each level of indirection.

- An int variable x.
- An int * pointer p pointing to x.
- An int ** pointer pp pointing to p.
- An int *** pointer ppp pointing to pp.

Example program:

```
#include <stdio.h>

int main() {
   int value = 10;
   int *ptr1;     // Pointer to an integer
   int **ptr2;    // Pointer to a pointer to an integer
   int ***ptr3;   // Pointer to a pointer to a pointer to an integer

   ptr1 = &value;
   ptr2 = &ptr1;
   ptr3 = &ptr2;

   printf("Value: %d\n", value);      // Direct access
   printf("Value via ptr1: %d\n", *ptr1);     // One level of dereference
   printf("Value via ptr2: %d\n", **ptr2);    // Two levels of dereference
   printf("Value via ptr3: %d\n", ***ptr3);   // Three levels of dereference

   // Update the value using the third-level pointer
   ***ptr3 = 30;
   printf("\nAfter updating via ptr3, new value: %d\n", value);

   return 0;
}
```

Dereferencing

To access the value of the original variable, you must dereference the chain of pointers one level at a time. The number of asterisks required is equal to the level of the pointer in the chain.

- *ptr1 gives you the value of value.
- **ptr2 first gives you the value of ptr1 (the address of value), and then dereferences that to get the value of value.
- ***ptr3 works the same way, traversing the chain of addresses to reach the final value.

Pointer increments and scale factor

When you perform arithmetic operations on a pointer, the compiler automatically adjusts the address based on the size of the data type it points to. The amount the pointer's address increases or decreases is called the scale factor.

Pointer increment (++)

- When you increment a pointer (e.g., ptr++), the compiler increases the memory address stored in the pointer, but not by just one byte.
- The address is increased by the sizeof() the data type the pointer is pointing to. This allows the pointer to move to the next "element" of that type in memory.

Example: Pointer increment with an array

```
#include <stdio.h>

int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int *ptr = arr; // ptr points to the first element of the array

   printf("Address of first element (arr[0]): %p\n", ptr);
   printf("Value of first element: %d\n", *ptr);

   ptr++; // Increment the pointer

   printf("Address of second element (arr[1]): %p\n", ptr);
   printf("Value of second element: %d\n", *ptr);

   return 0;
}
```

Scale factor

The scale factor is the size in bytes of the data type a pointer is declared to point to.

- For an int* (integer pointer) on most systems, the scale factor is typically 4 bytes. So ptr++ adds 4 to the memory address.
- For a char* (character pointer), the scale factor is 1 byte.
- For a double* (double pointer), the scale factor is 8 bytes.

Example demonstrating different scale factors:

```
#include <stdio.h>

int main() {
   int i = 10;
   char c = 'A';
   double d = 3.14;

   int *iptr = &i;
   char *cptr = &c;
   double *dptr = &d;

   printf("Initial int pointer address: %p\n", iptr);
   printf("Initial char pointer address: %p\n", cptr);
   printf("Initial double pointer address: %p\n", dptr);

   iptr++;  // Increments address by sizeof(int) (e.g., 4 bytes)
   cptr++;  // Increments address by sizeof(char) (e.g., 1 byte)
   dptr++;  // Increments address by sizeof(double) (e.g., 8 bytes)

   printf("\nAfter incrementing:\n");
   printf("New int pointer address: %p\n", iptr);
   printf("New char pointer address: %p\n", cptr);
   printf("New double pointer address: %p\n", dptr);

   return 0;
}
```

This behavior of pointer arithmetic is fundamental for efficiently traversing data structures stored in contiguous memory, such as arrays.

Pointers and Arrays:

In C, the name of an array acts as a constant pointer to its first element. This relationship means you can use pointer arithmetic to traverse and access array elements.

Key concepts

- The array name arr is equivalent to &arr[0].
- arr[i] is equivalent to *(arr + i). This means you can use either array subscript or pointer arithmetic to access elements.
- Pointer arithmetic automatically scales to the size of the data type. Incrementing a pointer moves it to the next element, not the next byte.

Example: Accessing array elements with a pointer

```
#include <stdio.h>
int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int *ptr = arr; // ptr points to the first element
   printf("First element using array syntax: %d\n", arr[0]);
   printf("First element using pointer syntax: %d\n", *ptr);
   // Move to the third element (index 2)
   ptr += 2;
   printf("Third element using pointer syntax: %d\n", *ptr);
   return 0;
}
```

Pointers and character strings

A string in C is a character array terminated by a null character \0. You can use a character pointer to point to the start of a string. This is efficient for handling strings of varying lengths.

Key concepts

- A char * can point to the first character of a string literal or a character array.
- When a character pointer is used with printf("%s", ptr), it prints characters from the address stored in ptr until it encounters the null terminator.
- Pointer arithmetic can be used to iterate through the string character by character.

Example: Manipulating strings with a pointer

```
#include <stdio.h>
int main() {
   char *str = "Hello"; // Pointer to a string literal
   printf("The string is: %s\n", str);
   // Iterate and print character by character
   char *temp = str;
   while (*temp != '\0') {
      printf("%c", *temp);
```

```
    temp++;
  }
  printf("\n");
  return 0;
}
```

Array of pointers

An array of pointers is an array where each element stores the address of another variable or data structure. This is particularly useful for handling arrays of strings of different lengths, as it avoids memory waste associated with 2D character arrays.

Example: Array of pointers to strings

```
#include <stdio.h>
int main() {
  char *names[] = {"Apple", "Banana", "Orange"};
  for (int i = 0; i < 3; i++) {
    printf("Fruit %d: %s\n", i, names[i]);
  }
  return 0;
}
```

Pointers as function arguments

Passing a pointer to a function allows it to directly access and modify the original variable in the calling function. This is known as "call by reference".

Example: Swapping values using pointers

```
#include <stdio.h>
// The function takes pointers as arguments
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
int main() {
  int m = 10, n = 20;
  printf("Before swap: m=%d, n=%d\n", m, n);
  swap(&m, &n); // Pass the addresses of m and n
  printf("After swap: m=%d, n=%d\n", m, n);
  return 0;
}
```

Functions returning pointers

A function can return a pointer to a data type. This is commonly used for dynamic memory allocation but requires careful handling to avoid returning a pointer to a local variable, which will be deallocated after the function exits.

Example: Returning a pointer to dynamically allocated memory

```
#include <stdio.h>
#include <stdlib.h>
int *create_array(int size) {
   int *arr = (int *)malloc(size * sizeof(int));
   if (arr == NULL) {
      return NULL;
   }
   // Initialize array elements
   for (int i = 0; i < size; i++) {
      arr[i] = i * 10;
   }
   return arr;
}
int main() {
   int size = 5;
   int *my_array = create_array(size);
   if (my_array != NULL) {
      for (int i = 0; i < size; i++) {
         printf("%d ", my_array[i]);
      }
      printf("\n");
      free(my_array); // Free the allocated memory
   }

   return 0;

}
```

Pointers to functions

A function pointer is a variable that stores the address of a function. It is declared with the function's return type, an asterisk, the pointer name in parentheses, and the function's parameter types. Function pointers enable powerful techniques like callback functions and function tables.

Example: Using a function pointer

```
#include <stdio.h>
int add(int a, int b) {
   return a + b;
}
int main() {
   // Declare a function pointer
   int (*fp)(int, int);
   // Assign the address of the 'add' function to the pointer
   fp = add;
   // Call the function through the pointer
   int result = fp(10, 20);
   printf("Result of addition: %d\n", result);
   return 0;
}
```

File management in C

File management in C involves storing and retrieving data from files on a disk, a crucial feature for data persistence that goes beyond the volatile nature of program memory. Key concepts and functions are defined in the <stdio.h> header file.

Defining and operating on a file

1. Define a file pointer: A FILE * pointer is needed to connect your program to a file.

```
#include <stdio.h>

FILE *file_ptr;
```

2. Open the file: Use fopen() to open or create a file with a specific mode.

3. Syntax: fopen("filename", "mode");

4. Modes: Common modes include "r" (read), "w" (write, overwrites), "a" (append), and their + variants for reading/writing. Add b for binary modes (e.g., "rb").

Example: Opening a file

```
#include <stdio.h>

int main() {

   FILE *file_ptr;

   file_ptr = fopen("example.txt", "w"); // Open in write mode

   if (file_ptr == NULL) {
```

```
        printf("Error: Could not open file.\n");
        return 1;
    }
    printf("File opened successfully.\n");
    return 0;
}
```

Closing a file

Always use fclose() to close a file after use. This flushes buffers and releases resources.

- Syntax: fclose(file_ptr);
- Return value: 0 on success, EOF on failure.

Example: Closing a file

```
#include <stdio.h>
int main() {
    FILE *file_ptr;
    file_ptr = fopen("example.txt", "w");
    if (file_ptr != NULL) {
        fclose(file_ptr);
        printf("File closed successfully.\n");
    }
    return 0;
}
```

Input/Output operations on files

Text files

Functions like fprintf(), fputs(), and fputc() are used for writing text data, while fscanf(), fgets(), and fgetc() are for reading.

Example: Writing and reading text

```
#include <stdio.h>
int main() {
    FILE *fp;
    char tsext_buffer[100];
    // Writing to a file (see source for full example)
    fp = fopen("text.txt", "w");
    if (fp != NULL) {
```

```
    fprintf(fp, "This is a test.");
    fclose(fp);
  }
  // Reading from a file (see source for full example)
  fp = fopen("text.txt", "r");
  if (fp != NULL) {
    if (fgets(text_buffer, 100, fp) != NULL) {
      printf("Read: %s", text_buffer);
    }
    fclose(fp);
  }
  return 0;
}
```

Binary files

Use fwrite() to write and fread() to read binary data efficiently.

Example: Writing and reading a struct

```
#include <stdio.h>
#include <stdlib.h>
struct Data {
  int id;
  float value;
};
int main() {
  struct Data my_data = {1, 99.9};
  struct Data read_data;
  FILE *fp;
  // Writing to a binary file (see source for full example)
  fp = fopen("data.bin", "wb");
  fwrite(&my_data, sizeof(struct Data), 1, fp);
  fclose(fp);
  // Reading from a binary file (see source for full example)
  fp = fopen("data.bin", "rb");
  fread(&read_data, sizeof(struct Data), 1, fp);
  printf("Read from file: ID=%d, Value=%.1f\n", read_data.id, read_data.value);
  fclose(fp);
  return 0;
}
```

Error handling during I/O operations

Handle file errors by checking fopen()'s return value for NULL.

- Use perror() to print system error messages.
- feof() checks for the end of the file.
- ferror() checks the error indicator for the stream.
- exit() can terminate the program on critical errors.
- clearerr() resets error indicators.