GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING FOR WOMEN
(Autonomous)
Approved by AICTE, New Delhi and Permanently Affiliated to Andhra University, Visakhapatnam)
Madhurawada :: Visakhapatnam – 530 048

PROBLEM SOLVING USING C

I.B.Tech. –II SEM- EEE End Examination

1. All questions carry equal marks
2. Must answer all parts of the question at one place

Date : 25-06-2025 **Time: 3Hrs.** **Max Marks: 70**

Solutions

UNIT-I

1. (a) Outline and Explain different data types with an example for each. **[7M]**

DATA TYPES IN C

A datatype is a keyword/predefined instruction used for allocating memory for data. A data type specifies the type of data that a variable can store such as integer, floating, character etc. It used for declaring/defining variables or functions of different types before to use in a program.

There are **three main data types** are in C programming:

1. Primitive (Primary) data types.
2. Derived data types.
3. User-defined data types (UDTs).

1.Primitive or built-in data types are used to represent simple data values, including characters, integers, void, float and double data. C language supports both signed and unsigned literals.

• Character (**char):** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
Ex: In a program storing person's first initial, the variable would be a character (eg.'A', '7', '$', ' ').

• Integer **(int):** As the name suggests, an int variable is used to store an integer as whole numbers (positive, negative, or zero).
Ex: In a program counting the number of items in a store, the variable holding that count would likely be an integer (e.g., 10, -5, 0).

• Floating-Point (**float):** It is used to store decimal numbers (numbers with floating point value).
Ex: In a program calculating the average temperature of a city, the temperature variable would be a floating-point number (e.g., 25.5).

• Double (**double):** It is used to store decimal numbers (numbers with floating point value) with double precision.
Ex: similar to float but longer in length .*double num2 = 3.198728764857268945*

• Boolean (**bool**): Booleans represent logical values, either True or False.
• Void (**void**) – This type indicates no value or empty value.

In addition to the above data types, C language also support data types like date and time together in a specific format, enumerated (a set of predefined values), long (a long positive or negative sequence of integers) and short (a short positive or negative sequence of integers).

2.Derived Data Type: These data types are derived from the primitive data types. They include arrays, pointers and functions.

• **Array:** A list with multiple elements of the same type and mentioned in a specific order.

Ex. The total number of elements in an array a[ ] = {1,2,3,4,5},

3.User-Defined Data Types : These data types in C is that a user can define and derive from an existing data type. It includes structure, union, typedef and enum.

• **Structures** allow users to group items of different data types into a single type.

• **enum** is useful to create custom data types with a set of named integer constants.

• **Typedef** is used to rename existing data types.

• **Union** data type can contain elements of different data types, with all members of the union stored in the same memory location.

(b) Evaluate the following expression a*b+c/d-e and describe the concept of arthematic precedence and associativity. **[7M]**

- **Arthematic Precedence : Arithmetic Operators are evaluated left to right using** the precedence of operator when the expression is written without the paranthesis.
  precedence,
  They are two levels of arithmetic operators in C.
  1. High Priority * / %
  2. Low Priority + -.

  Using Arthematic a*b and c/d are computed before the addition.

- **Associativity:** Associativity specifies the order in which the operators are evaluated with the same precedence in a expression. Commonly, operators are either left-associative (evaluated from left to right) or right-associative (evaluated from right to left). In the example a*b+c/d, the * and / operators are left-associative, so a*b is evaluated before c/d. The + operator is also left-associative, so the entire expression is evaluated as: ((a * b) + (c / d)).

Program:

```
#include <stdio.h>
int main() {
 int a = 10, b = 5, c = 20, d = 2, e = 3;
 int result = a * b + c / d - e;
 printf("Result: %d\n", result);
 return 0;
}
```

Calculation:

The expression a * b + c / d - e is evaluated according to operator precedence:

- a * b = (10 * 5) is calculated first, resulting in 50.
- c / d = (20 / 2) is calculated next, resulting in 10.
- Then, the results are combined: 50 + 10 - 3.
- Finally, the result (60 - 3) is 57.

(OR)

2. (a) With a suitable example explain the basic structure of a C program.

General Structure of a C program:

```
/* Documentation section */
/* Link section */
/* Definition section */
 /* Global declaration section */
main()
 {
Declaration part
Executable part (statements)
 }
 /* Sub-program section */
```

Explanation:

- **Documentation Section**: The documentation section is used for displaying any information about the program like the purpose of the program, name of the author, date and time written etc, and this section should be enclosed within comment lines. The statements in the documentation section are ignored by the compiler.
- **Link Section:** The link section consists of the inclusion of header files.
- The **definition section** consists of macro definitions, defining constants etc. The **#define** preprocessor directive is used to define constant or micro substitution. It can use any basic data type. Ex. **#define PI 3.14**
- Anything declared in the **global declaration section** is accessible throughout the program, i.e. accessible to all the functions in the program.
- **main() function** is mandatory for any program and it includes two parts, the declaration part and the executable part.
- The last section, i.e. **sub-program section** is optional and used when we require including user defined functions in the program.

Example:

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main()
{
4.printf("Hello C Language");
5.getch();
}
```

1. **#include <stdio.h>** - includes the **standard input output** library functions. The **printf() function** is defined in stdio.h .
2. **#include <conio.h>** - includes the **console input output** library functions. The **getch() function** is defined in conio.h file.
3. **void main()** - function is **the entry point of every program** in c language. The **void keyword** specifies that **it returns no value.**
4. The **printf()**- function is **used to print data** on the console.
5. The **getch()** - function **asks for a single character**. Until you press any key, it blocks the screen.

(b) What do you mean by type conversions? Explain different type conversions with an example for each. **[7M]**

Type conversion in C refers to the process of changing a variable from one data type to another. They are of two types.

A) Implicit type conversion can be done automatically by the compiler. This type of conversion is required when different data types are used in an expression or assignment, and the compiler automatically converts one to match the other.

Example:

```
#include <stdio.h>
int main() {
int num_int = 10;
float num_float;
num_float = num_int; // Implicit conversion from int to float
printf("Integer value: %d\n", num_int);
printf("Float value after implicit conversion: %f\n", num_float);
return 0;
}
```

Output:

Integer value: 10
Float value after implicit conversion: 10.000000

B) **Explicit type conversion** also known as casting, is when the programmer explicitly specifies the desired data type using a cast operator. This is done using the cast operator, which is the desired data type enclosed in parentheses, followed by the variable to be converted.

Example:

```
#include <stdio.h>
int main() {
 float num_float = 10.5;
 int num_int;
 num_int = (int) num_float; // Explicit conversion from float to int
 printf("Float value: %f\n", num_float);
 printf("Integer value after explicit conversion: %d\n", num_int);
return 0;
 }
```

Output:

Float value: 10.500000
Integer value after explicit conversion: 10

UNIT-II

3. (a) List and explain decision making using different if statements. **[7M]**

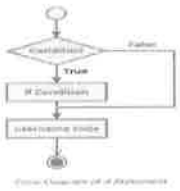
A statement that controls the sequence of statement execution, depending on the value of a integer expression.

if Statement: The if Statement may be implemented in different forms.
1: simple if statement. 2: if –else statement 3: nested if-else statement. 4: else if ladder.

1. **if statement.** The if statement controls conditional branching. The body of an if statement is executed if the value of the expression is nonzero. If the condition/expression is true, then the true statement will be executed otherwise the true statement block will be skipped and the execution will jump to the statement-x.

Syntax :

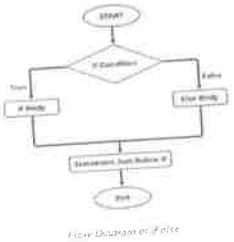
```
if(condition/expression)
{
    true statement;
}
statement-x;
```



2. **If-else statement:** It is used to carry out one of the two possible actions depending on the outcome of a logical test. The else portion is optional.

Syntax :

```
if (condition/expression)
{
    true statement;
}
else
{
    false statement;
}
statement-x;
```

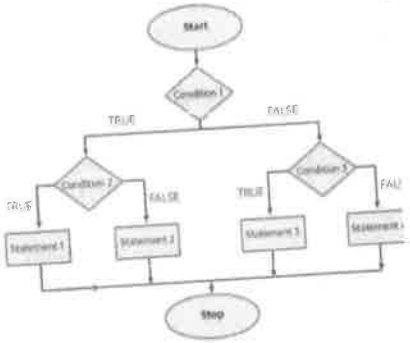


Here expression is a logical expression enclosed in parenthesis. if expression is true ,statement 1 or statement 2 is a group of statements ,they are written as a block using the braces { }.

3. **Nested if-else statement**: Within if block or else block another if – else statement can come. Such statements are called nested if statements.

Syntax :

```
if(condition-1) {
    if (condition-2) {
        statement-1;
      }
    else {
        statement-2;
      }
    }
    else {
     statement-3;
    }
   statement-x;
```

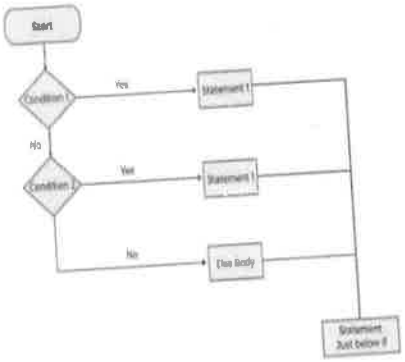


If the condition-1 is false, the statement-3 and statement-x will be executed. Otherwise it continues to perform the second test. If the condition-2 is true, the true statement-1 will be executed otherwise the statement-2 will be executed and then the control is transferred to the statement-x.

4.Else-if ladder: The if else-if statement is used to execute one code from multiple conditions. Inorder to create a situation in which one of several courses of action is executed we use ladder – if statements.

Syntax :

```
if( condition-1) {
        statement-1;
        }
        else if (condition-2) {
                statement-2;
                }
                else if (condition-3) {
                statement-3;
                }
                else if (condition-n) {
                statement-n;
                }
                else {
                default-statement;
                }
                statement-x;
```



(b) Write a C program to copy one string to another with and without using String manipulation functions. **[7M]**

- **Copying a String Using strcpy() (String Manipulation Function):** The **strcpy()** function, found in the <string.h> header, offers a straightforward way to copy strings.

Program-1: (Using String function)

```
#include <stdio.h>
#include <string.h> // Required for strcpy()
int main() {
   char source[] = "Hello, World!";
   char destination[20];
 // Copy the contents of 'source' to 'destination'
   strcpy(destination, source);
   printf("Original string: %s\n", source);
   printf("Copied string: %s\n", destination);
   return 0;
}
```

Output:

Original string: Hello, World!
Copied string: Hello, World!

- **Copying a String Without Using String Manipulation Functions:** This method involves iterating through the source string character by character and copying each to the destination string until the null terminator (\0) is encountered. The null terminator must also be copied to ensure the destination string is properly terminated.

Program-2: (Without String function)

```
#include <stdio.h>
int main() {
   char source[] = "Hello, World!";
   char destination[30];
   int i = 0;
 // Loop until the null terminator of the source string is reached
   while (source[i] != '\0') {
      destination[i] = source[i];
      i++;
   }
   // Null-terminate the destination string
   destination[i] = '\0';
   printf("Original string: %s\n", source);
   printf("Copied string: %s\n", destination);
   return 0;
 }
```

Output:

Original string: Hello, World!; **Copied string:** Hello, World!

(OR)

4. (a) Write a C program to find multiplication of two matrices using arrays. [7M]

```
#include <stdio.h>
int main() {
int r1, c1, r2, c2;
int i, j, k;
// Get dimensions of the first matrix
 printf("Enter rows and columns for the first matrix: ");
 scanf("%d %d", &r1, &c1);
 // Get dimensions of the second matrix
 printf("Enter rows and columns for the second matrix: ");
 scanf("%d %d", &r2, &c2);
 // Check if matrix multiplication is possible
  while (c1 != r2) {
      printf("Error! Column of first matrix not equal to row of second matrix.\n");
      printf("Enter rows and columns for the first matrix again: ");
      scanf("%d %d", &r1, &c1);
      printf("Enter rows and columns for the second matrix again: ");
      scanf("%d %d", &r2, &c2);
  }
 int firstMatrix[r1][c1];
 int secondMatrix[r2][c2];
 int resultMatrix[r1][c2];
 printf("\nEnter elements of the first matrix:\n"); // Get elements of the first matrix
 for (i = 0; i < r1; ++i) {
    for (j = 0; j < c1; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &firstMatrix[i][j]);
    }
 }
 printf("\nEnter elements of the second matrix:\n"); // Get elements of the second matrix
 for (i = 0; i < r2; ++i) {
    for (j = 0; j < c2; ++j) {
      printf("Enter element b%d%d: ", i + 1, j + 1);
      scanf("%d", &secondMatrix[i][j]);
    }
  }
// Initialize elements of result matrix to 0
    for (i = 0; i < r1; ++i) {
     for (j = 0; j < c2; ++j) {
       resultMatrix[i][j] = 0;
     }
    }
 // Perform matrix multiplication
    for (i = 0; i < r1; ++i) {
      for (j = 0; j < c2; ++j) {
        for (k = 0; k < c1; ++k) {
          resultMatrix[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
        }
      }
    }
 // Display the result matrix
    printf("\nResultant Matrix:\n");
    for (i = 0; i < r1; ++i) {
```

```
    for (j = 0; j < c2; ++j) {
       printf("%d ", resultMatrix[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

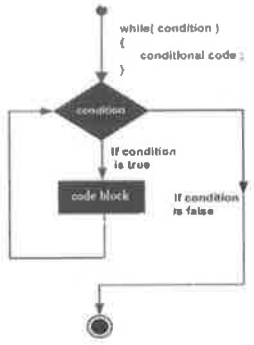
Output:

Enter rows and columns for the first matrix: 2 2
Enter rows and columns for the second matrix: 2 2
Enter elements of the first matrix:
Enter element a11: 1
Enter element a12: 2
Enter element a21: 3
Enter element a22: 4
Enter elements of the second matrix:
Enter element b11: 5
Enter element b12: 6
Enter element b21: 7
Enter element b22: 8
Resultant Matrix:
19 22
43 50

(b) Compare and contrast while and do-while loop. Give example for each. **[7M]**

LOOPING It is to execute a group of instructions repeatedly, a fixed no of times or until a specified condition is satisfied. The **while** loop is often called the **entry verified loop**, whereas the **do-while** loop is an **exit verified loop**. **while statement** carrys out a set of statements to be executed repeatedly until some condition is satisfied.

- The **while** keyword is followed by a parenthesis, in which there should be a Boolean expression. Followed by the parenthesis, there is a block of statements inside the curly brackets.
- The statement is executed so long as the expression is true. Statement can be simple or compound.



Syntax :

```
while (expression){
   statement(s);
}
```

Example for while loop:

```
#include<stdio.h>
int main() {
int a=1;
while(a <= 5)   {
   printf("Hello World \n");
   a++;
  }
printf("End of loop");
return 0; }
```

Output

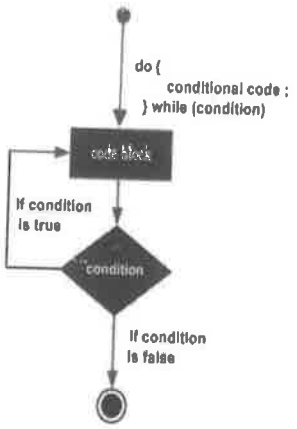
Hello World
Hello World

Hello World
Hello World
Hello World
End of loop

- **do while statement** carrys out a set of statements to be executed repeatedly so long as a condition is true.

Syntax :

```
do
 {
   statement(s);
 }
while(condition);
```



Example for while loop:

```
#include<stdio.h>
int main() {
int a=0; //Loop variable declaration and initialization
//do while loop
do
{
   printf("Hello World \n");
   a++;
}
   while(a < 3);
   return 0;
}
```

Output:

Hello World
Hello World
Hello World

THE DIFFERENCE BETWEEN while loop AND do – while loop

1) In the while loop the condition is tested in the beginning whereas in the other case it is done at the end.

2) In while loop the statements in the loop are executed only if the condition is true whereas in do – while loop even if the condition is not true the statements are executed atleast once.

UNIT-III

5. (a) What are the advantages of a function? List and explain different parameter passing methods to a function. **[7 M]**

DEFINITIONS OF FUNCTIONS:

- Self-contained program segment that is placed separately from the main program to perform some specific well defined task is called function.
- Function is a small segment of program that carries out some specific and well-defined task.
- A program segment that is placed separately from the main program is called function.
- Functions are the building blocks of a C program.

Advantages:

- It reduces the length of source program.
- Breaks the complexity of a program.

- Break small program into small program.
- It is easy to maintain, modify and understand.
- A program can be divided into smaller subprograms.
- It facilitates top down modular programming.
- It facilitates top down modular programming.
- The length of the source program can be reduced using functions
- Avoid rewriting the same sequence of code at two or more locations in a program.

Different Parameter Passing Methods: A function based on whether the Argument is present or not, whether the value is returned or not.

Syntax: *return_Type function_Name(parameter1, parameter2, parameter3)* {
// code to be executed
}

There are two primary methods for passing parameters to a function in many programming languages:

- **Call by Value (Pass by Value)**:

Explanation: When parameters are passed by value, a copy of the actual argument's value is passed to the function's formal parameter. Any modifications made to the formal parameter within the function do not affect the original actual argument in the calling function. This is because the function operates on a separate copy.

Syntax:

```
void functionName(int parameter) {
    parameter = parameter + 10;
}
int main() {
    int x = 5;
    functionName(x); // x's value (5) is copied to parameter
    // x remains 5
    return 0;
}
```

- **Call by Reference (Pass by Pointer)**:

Explanation: When parameters are passed by reference (using pointers), the memory address of the actual argument is passed to the function's formal parameter. The formal parameter is a pointer that stores this address. This allows the function to directly access and modify the original actual argument in the calling function through its memory address.

Syntax:

```
void functionName(int *parameter_ptr) {
    // parameter_ptr holds the address of the original variable
    *parameter_ptr = *parameter_ptr + 10; // Dereference to modify the original value
}
int main() {
    int x = 5;
    functionName(&x); // Address of x is passed to parameter_ptr
    // x is now 15
    return 0;
}
```

(b) Write a C program to find the sum of array elements using Pointers [7 M]

```
#include <stdio.h>
int main() {
  int arr[] = {10, 20, 30, 40, 50}; // Declare and initialize an array
  int size = sizeof(arr) / sizeof(arr[0]);
  int sum = 0;
  int *ptr = arr; // Declare a pointer and initialize

// Loop through the array using pointer arithmetic
  for (int i = 0; i < size; i++) {
    sum += *ptr;
    ptr++;
  }
  printf("Sum of array elements: %d\n", sum); // Print the sum
  return 0;
}
```

Output:

Sum of array elements: 150

(OR)

6. (a) Define recursion. Write a C program to find Fibonacci series upto 10 terms using recursion. **[7 M]**

When a function is called repeatedly until specific condition is satisfied, then it is known as Recursion. A function is called recursive if a statemcnt within the body of a function calls the same function.

Program:

```
#include <stdio.h>
int fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
  int terms = 10;
  printf("Fibonacci Series (first %d terms):\n", terms);
  for (int i = 0; i < terms; i++) {
    printf("%d ", fibonacci(i));
  }
  printf("\n");
  return 0;
}
```

Output:

Fibonacci Series (first 10 terms): 0 1 1 2 3 5 8 13 21 34

(b) With suitable example explain about dynamic memory allocation. **[7 M]**

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. The process of allocating memory at runtime is known as dynamic memory allocation. Library functions are used for allocating and freeing memory during execution of a program. These functions are defined in stdlib.h. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc() 2. calloc() 3. realloc() 4. free()

Syntax

ptr=(cast-type*)malloc(byte-size)

Example

```
int *x;
x = (int*)malloc(100 * sizeof(int)); //memory space allocated to variable x
free(x); //releases the memory allocated to variable x
```

/*Read n numbers and find their sum */

```
#include<stdio.h>
#include<stdlib.h>
int main() {
int num, i, *ptr, sum = 0;
printf("Enter number of elements: ");
scanf("%d", &num);
ptr = (int*) malloc(num * sizeof(int)); //memory allocated using malloc
if(ptr == NULL) {
   printf("Error! memory not allocated.");
   exit(0);
}
printf("Enter elements of array: ");
for(i = 0; i < num; ++i) {
   scanf("%d", ptr + i);
   sum += *(ptr + i);
}
printf("Sum = %d", sum);
free(ptr);
return 0;
}
```

UNIT-IV

7. **(a) With an example distinguish the concepts of Structures and Unions.** **[7 M]**

Difference between Structures and Unions:

C Structures	C Unions
Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
Structure occupies higher memory space.	Union occupies lower memory space over structure.
We can access all members of structure at a time.	We can access only one member of union at a time.

Structure example:
struct student {
int mark;
char name[6];
double average;
};
	Union example:
	union student {
	int mark;
	char name[6];
	double average;
	};

For above structure, memory allocation will be like below.	For above union, only 8 bytes of memory will be allocated since double data type will occupy

int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	maximum space of memory over other data types. Total memory allocation=8 bytes

Program:
#include<stdio.h>
#include<conio.h>
struct emp {
int id;
name[36];
float sal;
};
void main() {
struct emp e;
printf("Enter employee Id, Name, Salary: ");
scanf("%d",&e.id);
scanf("%s",&e.name);
scanf("%f",&e.sal);
printf("Id:%d",e.id);
printf("\nName:%s",e.name);
printf("\nSalary: %f",e.sal);
getch();
}
Output:
Enter employee Id, Name, Salary:
5 John 45000
Id : 05
Name: John
Salary: 45000.00
	Program:
	#include<stdio.h>
	#include<conio.h>
	union employee {
	int id;
	char name[50];
	}e1; //declaring e1 variable for union
	int main() {
	e1.id=101;
	strcpy(e1.name, "John");
	//copying string into char array
	printf("employee 1 id : %d\n", e1.id);
	printf("employee 1 name : %s\n", e1.name);
	return 0;
	}
	Output:
	employee 1 id : 1869508435
	employee 1 name : John

(b) What are nested structures? Explain it with a program. **[7 M]**

A nested structure, particularly in programming languages like C, refers to a structure that contains another structure as one of its members. This allows for the creation of more complex and organized data types by grouping related data hierarchically.

Consider the information about a Person needs to be stored, including **name, ID, and Address**. The Address itself is a collection of related data: **street, city, and zip_code.**

Example:

```
#include <stdio.h>
#include <string.h>
// Define the inner structure for Address
struct Address {
   char street[50];
   char city[50];
   int zip_code;
};
// Define the outer structure for Person, which includes Address
struct Person {
   char name[50];
   int id;
   struct Address person_address; // Nested structure member
};
```

```
int main() {
struct Person p1 = {" John", 202, {"143 Main St", "Milton Kings", 56789}};
   // Accessing members of the nested structure
   printf("Person Name: %s\n", p1.name);
   printf("Person ID: %d\n", p1.id);
   printf("Address: %s, %s %d\n", p1.person_address.street, p1.person_address.city,
p1.person_address.zip_code);
   return 0;
}
```

OUTPUT:

Person Name: John
Person ID: 202
Address: 143 Main St, Milton Kings 56789

(OR)

8. (a) Apply the concept of structures and display the marks of the three students in three different subjects [7M]

```
#include <stdio.h>
#include <string.h>
struct Student {
 char name[50];
 int roll_number;
 float subject1_marks;
 float subject2_marks;
 float subject3_marks;
};
int main() {
struct Student students[3];
// Input data for each student
for (int i = 0; i < 3; i++) {
printf("Enter details for student %d:\n", i + 1);
printf("Roll Number: ");
scanf("%d", &students[i].roll_number);
printf("Marks in Subject 1: ");
scanf("%d", &students[i].subject1_marks);
printf("Marks in Subject 2: ");
scanf("%d", &students[i].subject2_marks);
printf("Marks in Subject 3: ");
scanf("%d", &students[i].subject3_marks);
}
// Display student information
   printf("\nStudent Information:\n");
   for (int i = 0; i < 3; i++) {
   printf("Roll Number: %d\n", students[i].roll_number);
   printf("Marks in Subject 1: %d\n", students[i].subject1_marks);
   printf("Marks in Subject 2: %d\n", students[i].subject2_marks);
   printf("Marks in Subject 3: %d\n", students[i].subject3_marks);
   printf("\n");
    }
 return 0;
}
```

Enter details for student 1:
Roll Number: 1
Marks in Subject 1: 70
Marks in Subject 2: 80
Marks in Subject 3: 90
Enter details for student 2:
Roll Number: 2
Marks in Subject 1: 75
Marks in Subject 2: 85
Marks in Subject 3: 98
Enter details for student 3:
Roll Number: 3
Marks in Subject 1: 50
Marks in Subject 2: 60
Marks in Subject 3: 70

Output: Student Information:
Roll Number: 1
Marks in Subject 1: 70.00
Marks in Subject 2: 80.00
Marks in Subject 3: 90.00
Roll Number: 2
Marks in Subject 1: 75.00
Marks in Subject 2: 85.00
Marks in Subject 3: 98.00
Roll Number: 3
Marks in Subject 1: 50.00
Marks in Subject 2: 60.00
Marks in Subject 3: 70.00

(b) What about typedef and enumerated types. Write C program to find a particular day in a week using enumerated types. **[7 M]**

A. typedef : The typedef is a keyword that allows the programmer to create a new data type name for an existing data type. The purpose of typedef is to redefine the name of an existing variable type.

Syntax: typedef datatype alias_name;

Example of typedef:

```
#include<stdio.h>
void main()
{
 typedef int digits;
 digits a,b,sum;
 printf("Enter a and b values:");
 scanf("%d%d",&a,&b);
 sum=a+b;
 printf("The sum is:%d",sum);
 }
```

Output:
Enter a and b values: 5 10
Sum: 20

B. Enumerations Type: An enum is a keyword, it is an user defined data type. All properties of integer are applied on Enumeration data type so size of the enumerator data type is 2 byte. It work like the Integer. It is used for creating an user defined data type of integer. Using enum we can create sequence of integer constant value.

Syntax: enum tagname {value1, value2, value3,....};

- In above syntax enum is a keyword. It is a user defiend data type.
- In above syntax tagname is our own variable. tagname is any variable name.
- value1, value2, value3,.... are create set of enum values.

Example:

```
#include<stdio.h>
enum week {sun, mon, tue, wed, thu, fri, sat};
void main()
{
enum week today;
today=tue;
 printf("%d day",today+1);
}
```

Output:

3 day

UNIT-V

9.(a) Construct a C program using Files to copy the contents of one file to another file. [7 M]

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
FILE *inputFile,*outputFile;
char ch;
char inputFileName[100],outputFileName[100];
printf(Enter the input file name: “);
scanf(“%s”, inputFileName);
printf(Enter the input file name: “);
scanf(“%s”, inputFileName);
inputFile = fopen("inputFileName","r");
if(inputFile= = NULL){
printf(“Error opening input file\n”);
return 1;
}
outputFile = fopen("outputFileName","w");
if(outputFile = = NULL){
printf(“File is not available\n”);
fclose(inputFile);
return 1;
}
while(ch = fgetc(inputFile) !=EOF)
{
fputc(ch, outputFile);
}
printf(“File copied successfully\n”);
fclose(inputFile);
fclose(outputFile);
return 0; }
```

(b) Demonstarte random access to files with a suitable program using fseek(). [7M]

Random Access Files: Every open file has an associated file position indicator, which describes where read and write operations take place in the file. The position is always specified in bytes from the beginning of the file. The random access file handling, accesses only the file at the point at which the data should be read or written, rather than having to process it sequentially.

Three functions:

- long ftell(FILE *fp);
- int fseek(FILE *fp, long offset, int from);
- void rewind(FILE *fp)

fseek(): The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location. The fseek() function moves the file pointer associated with the stream to a new location that is offset bytes from origin.

Syntax: int fseek(FILE *stream_pointer, long offset, int origin);

Argument:

* **stream_pointer** is a pointer to the stream FILE structure of which the position indicator should be changed;

* **offset** is a long integer which specifies the number of bytes from origin where the position indicator should be placed;

* **origin** is an integer which specifies the origin position. It can be:

- SEEK_SET: origin is the start of the stream
- SEEK_CUR: origin is the current position
- SEEK_END: origin is the end of the stream

Program:

```
#include<stdio.h>
#include<stdlib>
void main (){
FILE *fp;
int length;
fp = fopen("file.txt", "r");
fseek(fp, 0, SEEK_END);
length = ftell(fp);
fclose(fp);
printf("Size of file: %d bytes", length);
getch();
}
```

Output: Size of file: 21 bytes

(OR)

9. (a) Differentiate Text and Binary files with a program. [7M]

File Opening Modes:

Modes	Description	Program

r	opens a text file in read mode	#include<stdio.h>
		int main() {
		int num;
		FILE *fptr;
		fptr = fopen("program.txt","r");
		printf("Enter num: ");
		scanf("%d",&num);

w	opens a text file in write mode	
a	opens a text file in append mode	
r+	opens a text file in read and write mode	
w+	opens a text file in read and write mode	

a+	opens a text file in read and write mode	fprintf(fptr,"%d",num);
		fclose(fptr);
		return 0;
		}

rb	opens a binary file in read mode	#include<stdio.h>
		int main() {
		int num;
		FILE *fptr;
		fptr = fopen("program.txt","**rb**");
		printf("Enter num: ");
		scanf("%d",&num);
		fprintf(fptr,"%d",num);
		fclose(fptr);
		return 0;
		}

wb	opens a binary file in write mode	
ab	opens a binary file in append mode	
rb+	opens a binary file in read and write mode	
wb+	opens a binary file in read and write mode	
ab+	opens a binary file in read and write mode	

(b) Illustrate the working of file functions with a program. **[7 M]**

File handling in C allows programs to interact with files on the system, enabling operations like creating, reading, writing, and closing files.

Basic Steps for File Handling:

- **Declare a file pointer:** A pointer of type FILE* is declared to represent the file in the program.
- **Open the file:** The fopen() function opens a file and returns a FILE* pointer. It takes the filename and mode (e.g., "w" for write, "r" for read, "a" for append) as arguments.
- **Perform file operations:** Functions like fprintf() (write formatted data), fscanf() (read formatted data), fputc() (write a character), fgetc() (read a character), fgets() (read a line), and fputs() (write a string) are used for data manipulation.
- **Close the file:** The fclose() function closes the file, releasing the associated resources.

Program:

```
#include <stdio.h>
int main() {
   FILE *fptr;
// --- Writing to a file ---
   fptr = fopen("abc.txt", "w"); // Open in write mode ("w")
   if (fptr == NULL) {
      printf("Error opening file for writing!\n");
      return 1;
   }
   fprintf(fptr, "Hello from C file handling!\n"); // Write a string
   fprintf(fptr, "This is a second line.\n");
   fclose(fptr); // Close the file
   printf("Data written to example.txt\n");
 // --- Reading from a file ---
   char buffer[100]; // Buffer to store read data
   fptr = fopen("abc.txt", "r"); // Open in read mode ("r")
   if (fptr == NULL) {
      printf("Error opening file for reading!\n");
```

```
        return 1;
    }
    printf("Content of abc.txt:\n");
    while (fgets(buffer, sizeof(buffer), fptr) != NULL) { // Read line by line
        printf("%s", buffer);
    }
    fclose(fptr); // Close the file
// --- Appending to a file ---
    fptr = fopen("abc.txt", "a"); // Open in append mode ("a")
    if (fptr == NULL) {
        printf("Error opening file for appending!\n");
        return 1;
    }
    fprintf(fptr, "This line is appended.\n");
    fclose(fptr);
    printf("Data appended to example.txt\n");
    // --- Reading again to see appended content ---
    fptr = fopen("abc.txt", "r");
    if (fptr == NULL) {
        printf("Error opening file for reading after append!\n");
        return 1;
    }
    printf("\nContent of abc.txt after appending:\n");
    while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
        printf("%s", buffer);
    }
    fclose(fptr);
    return 0; // Successful execution
}
```

Prepared By

(V Sree Vidhya)

Verified by

R.Sridevi

(R.Sridevi)