2074-Chaitra

1 a) What is programming language? What is the difference between source code and object code?

A programming language is a formalized set of instructions that a computer can interpret and execute. It provides a way for humans to communicate with computers and instruct them to perform specific tasks. Programming languages consist of a syntax and a set of rules that define how code is written and structured. Developers use programming languages to write software applications, scripts, and other computer programs.

Source code refers to the human-readable version of a program written in a programming language.

Object code, on the other hand, refers to the machine-readable version of the program that is produced after the source code is compiled.

Readability: Source code is written in a programming language and is readable by humans, while object code is in a machine-readable binary format.

Purpose: Source code is written, edited, and maintained by programmers to create and modify software. Object code is the output of the compilation process and is used for execution by the computer.

Level of Abstraction: Source code operates at a higher level of abstraction, using human-readable constructs, while object code operates at a lower level, directly representing machine instructions.

Editability: Source code is easily editable and can be modified to make changes or improvements to the program. Object code is not meant to be directly edited by humans and is typically generated from the source code.

1 b) What is debugging and testing? What are the steps to be followed for developing application software?

Debugging:

Debugging is the process of identifying and fixing errors or bugs in software code. It involves tracing and analyzing the code to find the root cause of issues that lead to unexpected behavior, crashes, or incorrect results. Debugging often requires tools and techniques that assist developers in isolating and rectifying problems in the code.

Testing:

Testing is the process of evaluating a software application to ensure it behaves as expected and meets the defined requirements. Testing involves running the software and various test cases to identify defects and verify that the application functions correctly under different scenarios.

Here are the general steps to be followed for developing application software:

Requirements Gathering and Analysis:

Understand the project's goals and requirements.
Communicate with stakeholders to gather detailed specifications.

Planning:
Define the scope of the project.
Create a development and testing plan.
Determine the resources, timeline, and milestones.

Design:
Create a high-level architecture and design for the software.
Define data structures, algorithms, and user interfaces.

Implementation (Coding):
Write source code based on the design.
Follow coding standards and best practices.
Use version control to manage code changes.

Testing:
Write test cases and scenarios based on requirements.
Perform unit testing (testing individual components) and integration testing (testing interactions between components).
Execute functional, performance, security, and usability tests.

Debugging:
Identify and isolate defects using debugging tools and techniques.
Analyze error messages, logs, and stack traces to locate issues.
Fix bugs by modifying the code.

Quality Assurance:
Review code and test cases for accuracy and completeness.
Conduct peer reviews and code inspections.
Use automated testing tools to ensure consistent quality.

Documentation:
Create user documentation, API documentation, and technical documentation.
Document code, design decisions, and architecture.

Deployment and Release:
Prepare the software for deployment to the target environment.
Release the software to users or customers.

Maintenance and Updates:
Monitor and gather feedback from users.
Address issues and release updates as needed.

Throughout the development process, the steps of debugging and testing are intertwined. Debugging is an ongoing process to identify and fix issues in the code, while testing ensures that the application meets quality standards and performs as expected.

2 a) What is the difference between variable declaration and variable definition in C? Explain with examples.

In C programming, variable declaration and variable definition are two different concepts, although they are often used interchangeably. Let's break down the differences between them with examples.
Variable Declaration: Variable declaration is the act of announcing the existence and data type of a variable to the compiler. It informs the compiler about the name and data type of the variable that will be used in the program, but it doesn't allocate memory for the variable at this stage. Variable declarations typically occur at the beginning of a code block or function to indicate the variables that will be used within that scope.

For example-:
extern int x;
extern float y;

Variable Definition: Variable definition is the act of actually creating the memory space for a variable and initializing its value. When you define a variable, you allocate memory for it and, if provided, give it an initial value. A variable can be declared and defined simultaneously.

Variable Definition
int x; // Defines and allocates memory for an integer variable 'x'
int main() {
    // Variable Declaration and Definition
    int z = 10;
    return 0;
}

2.b) What is macro expansion and file inclusion in C. Explain with examples.

Macro Expansion: Macro expansion involves the replacement of macro identifiers with their corresponding macro definitions. Macros are defined using the #define preprocessor directive, and they allow you to create shortcuts for frequently used code snippets or values.

#include <stdio.h>
#define PI 3.14159
#define SQUARE(x) ((x) * (x))
int main() {
    float radius = 5.0;
    float area = PI * SQUARE(radius);
    printf("Area of the circle: %f\n", area);
    return 0;
}
File inclusion: It allows you to include the content of another file in your C program. This is often used to modularize code, separate declarations from implementations, and reuse code across multiple files.

// main.c
#include <stdio.h>
#include "functions.h"
int main() {
    int result = add(5, 3);
    printf("Result: %d\n", result);

```
    return 0;
}
// functions.h
int add(int a, int b);

// functions.c
#include "functions.h"
int add(int a, int b) {
    return a + b;
}
```

3. What are functions for formatted and unformatted output? Write down its syntax. Write down the output of printf() function for the following sections of the statements.

In C programming, there are two types of functions for output: formatted output and unformatted output.
Formatted Output: Formatted output is achieved using the printf() function. It allows you to display formatted text, including variables, constants, and other data, with specified formatting options.

```
    printf("The number is: %d\n", num);
```

Unformatted Output: Unformatted output is achieved using the putc() and putchar() functions. These functions output individual characters.

```
int putc(int character, FILE *stream);
int putchar(int character);
```

The output of printf() function for the given code is-:
a=5.789100 and b=6789 from first line
a=5.79 and b=006789 from second line
a=5.79 and b=6789 from third line

4a) Explain about while loop with its syntax and flowchart.
A "while loop" is a control structure in programming that allows a set of instructions to be executed repeatedly as long as a specified condition is true. This loop type is particularly useful when you want to repeat a block of code an unknown number of times, based on a condition that is evaluated before each iteration.

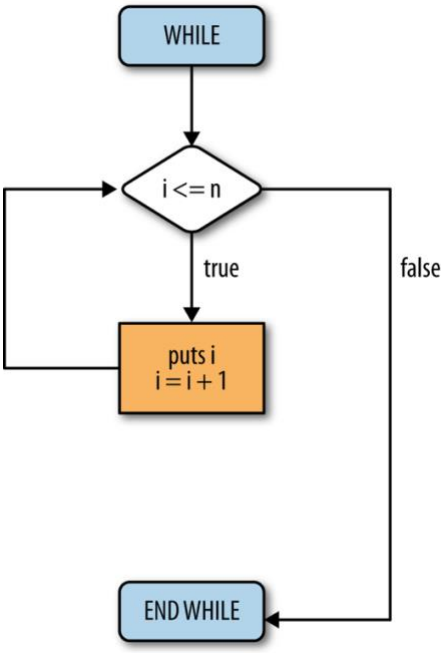
Syntax of a "while loop" in C:

```
while (condition) { // Code block to be executed while the condition is true }
```

Explanation of the "while loop" in C:
The condition is evaluated. If the condition is true, the code block inside the loop is executed. If the condition is false initially, the loop is skipped entirely, and the program continues with the code following the loop.

After the code block is executed, the condition is evaluated again. If it is still true, the code block is executed again. This process continues until the condition becomes false. Once the condition is false, the loop terminates, and the program continues with the code following the loop.

Flowchart:



4 b) Write a program to convert binary to decimal number.

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

// function prototype
int convert(long long);

int main() {

  long long n;

  printf("Enter a binary number: ");
  scanf("%lld", &n);

  printf("%lld in binary = %d in decimal", n, convert(n));

  return 0;
}

// function definition
int convert(long long n) {
```

```
  int dec = 0, i = 0, rem;

  while (n != 0) {

    // get remainder of n divided by 10
    rem = n % 10;

    // divide n by 10
    n /= 10;

    // multiply rem by (2 ^ i)
    // add the product to dec
    dec += rem * pow(2, i);

    // increment i
    ++i;
  }

  return dec;
}
```

5a) Can function return an array to the calling function? Explain with example

Yes, in C, a function can return an array to the calling function. However, C does not allow direct return of entire arrays from functions. Instead, you can achieve this by returning a pointer to the first element of the array. This is because arrays in C are represented as pointers to the first element in memory.

Here's an example to illustrate how to return an array from a function:

```
#include <stdio.h>

// Function to create and return an array
int* createArray(int size) {
    int *arr = malloc(size * sizeof(int)); // Allocate memory for the array
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1; // Fill the array with values
    }
    return arr; // Return a pointer to the array
}

int main() {
    int size = 5;
    int *resultArray = createArray(size);

    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", resultArray[i]);
    }
```

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

5b) Write a Program To Find the largest elements in rows in a mxn matrix and store it in an array

```
#include <stdio.h>

int main()
{
    int m,n;                //Matrix Size Declaration
    printf("Enter the number of rows and column: \n");
    scanf("%d %d",&m,&n);   //Matrix Size Initialization
    int arr[10][10];        //Matrix Size Declaration
    printf("\nEnter the elements of the matrix: \n");
    for(int i=0;i<m;i++)    //Matrix Initialization
    {
        for(int j=0;j<n;j++)
        {
            scanf("%d",&arr[i][j]);
        }
    }
    printf("\nThe elements in the matrix are: \n");
    for(int i=0;i<m;i++)     //Print the matrix
    {
        for(int j=0;j<n;j++)
        {
            printf("%d ",arr[i][j]);
        }
        printf("\n");
    }
    int i = 0, j;
    int max = 0;
    int res[m];
    while (i < m)   //Check for the largest element in an array
    {
       for ( j = 0; j < n; j++)
       {
          if (arr[i][j] > max)
          {
             max = arr[i][j];
          }
       }
       res[i] = max;
       max = 0;
       i++;
    }
    for(int i = 0; i < n; i++)      //Print the largest element in an array
    {
```

```
        printf("Largest element in row %d is %d \n", i, res[i]);
    }

    return 0;
}
```

6a) How does a structure differ from array and what are the different ways to access struct member?

A structure (struct) and an array are both ways of organizing and storing data in C, but they have different characteristics and use cases.

Structures (struct):

Purpose: A structure is a composite data type that groups together variables of different data types under a single name. It allows you to create a custom data structure to represent a collection of related data.
Data Types: A struct can contain variables of different data types, which makes it suitable for grouping heterogeneous data.
Access: The variables within a struct are called members, and they can have meaningful names. Members are accessed using the dot (.) operator.
Memory: The memory for a struct is allocated based on the sum of the memory requirements of its members. Each member is stored at a separate memory location.
Size: The size of a struct is determined by the sum of the sizes of its members, potentially with padding for alignment.

Arrays:

Purpose: An array is a collection of elements of the same data type, stored in contiguous memory locations. It is used to represent a sequence of related data.
Data Types: All elements of an array must have the same data type.
Access: Elements of an array are accessed using the index within square brackets ([]).
Memory: The memory for an array is allocated as a contiguous block, ensuring efficient access to elements.
Size: The size of an array is determined by the number of elements it contains, multiplied by the size of each element.

Accessing Struct Members:
There are a few ways to access members of a struct:

Dot Operator (.): Members of a struct can be accessed using the dot operator followed by the member's name.

```
    struct Student {
    char name[50];
    int age;
};
struct Student student1;
strcpy(student1.name, "Alice");
student1.age = 20;
```

Pointer to Struct (->): When working with a pointer to a struct, you use the arrow operator (->) to access members instead of the dot operator.

```
    struct Student student2;
struct Student *ptr = &student2;
strcpy(ptr->name, "Bob");
ptr->age = 22;
```

Array of Structs: If you have an array of structs, you can use an index to access individual struct members.

```
  struct Student studentArray[3];
strcpy(studentArray[0].name, "Charlie");
studentArray[0].age = 19;
```

Structures allow you to group related data together with meaningful names, making the code more readable and maintainable. Arrays, on the other hand, are used to store sequences of data elements of the same type. Both structures and arrays have their own use cases, and choosing the appropriate one depends on the nature of the data you're working with and the specific requirements of your program.

6.b) Create a structure named student that has name, r:oii and marks as members. Assume appropriate types and size of members. Use this structure to read and display records of 10 students. Create two functions: One is to read information of students and other to display the information.

```
#include <stdio.h>
// Define the structure
struct student {
    char name[50];
    int roll_number;
    float marks;
};
// Function to read information of students
void readStudentInfo(struct student *s) {
    printf("Enter student name: ");
    scanf("%s", s->name);
    printf("Enter roll number: ");
    scanf("%d", &s->roll_number);
    printf("Enter marks: ");
    scanf("%f", &s->marks);
}
// Function to display information of students
void displayStudentInfo(const struct student *s) {
    printf("Name: %s\n", s->name);
    printf("Roll Number: %d\n", s->roll_number);
    printf("Marks: %.2f\n", s->marks);
    printf("----------------------\n");
}
```

```
int main() {
   struct student students[10]; // Array of 10 students
    printf("Enter information for 10 students:\n");
   for (int i = 0; i < 10; ++i) {
      printf("Student %d:\n", i + 1);
      readStudentInfo(&students[i]);
   }
   printf("\nStudent Information:\n");
   for (int i = 0; i < 10; ++i) {
      printf("Student %d:\n", i + 1);
      displayStudentInfo(&students[i]);
   }
   return 0;
}
```

Q) 7(a) Define :

a)**int *p**
The declaration int *p in C is used to declare a pointer to an integer. It means that the variable p is a pointer that can hold the memory address of an integer variable. Pointers are variables that store memory addresses, allowing you to indirectly access the value stored at that memory location.

b) **int p(char *a)**
The declaration int p(char *a) in C is a function declaration. It declares a function named p that takes a single parameter, which is a pointer to a character (a C-style string), and returns an integer.

c) **int (*p(char *a))**
The declaration int (*p(char *a)) in C is a function declaration that specifies a function named p which takes a parameter a that is a pointer to a character (string) and returns a pointer to an integer.

d) **int *p(void)**
The declaration int *p(void) in C is a function declaration that specifies a function named p which takes no parameters (receives no arguments) and returns a pointer to an integer.

e) **int (*p[10])(char a[])**
int (*p[10]): This part declares an array of 10 pointers to functions returning an integer.
(char a[]): This specifies the parameter of the function, which is a character array.
So, the declaration int (*p[10])(char a[]) declares an array of 10 function pointers, where each function takes a character array as a parameter and returns an integer.

Qn 7(b) Explain the advantage of using pointer in C programming.

Pointers are a fundamental and powerful feature of the C programming language that provide several advantages and capabilities. Here are some of the key advantages of using pointers in C programming:

Dynamic Memory Allocation: Pointers enable dynamic memory allocation using functions like malloc, calloc, and realloc. This allows you to allocate memory at runtime and manage data structures efficiently.
Efficient Memory Usage: Pointers allow you to work directly with memory addresses, which can lead to more efficient memory usage and reduced overhead. You can create complex data structures without needing to copy data, improving performance and minimizing memory consumption.
Passing Data by Reference: Pointers allow you to pass data by reference to functions, enabling functions to modify the original data. This is particularly useful when you want to update values in the calling function or when dealing with large data structures.
Dynamic Data Structures: Pointers facilitate the creation of dynamic data structures such as linked lists, trees, and graphs. These structures can grow or shrink as needed, providing flexibility in handling complex data.
String Manipulation: C-style strings (null-terminated character arrays) are often manipulated using pointers. Pointers simplify string operations like concatenation, copying, and searching.
Pointer Arithmetic: You can perform arithmetic operations on pointers, allowing you to navigate through arrays and data structures efficiently. Pointer arithmetic is crucial for iterating over arrays and accessing individual elements.
Function Pointers: C supports function pointers, allowing you to store and call functions dynamically. This feature is useful for implementing callback mechanisms, event handling, and dynamic dispatch.
Low-Level Hardware Interaction: Pointers enable low-level interaction with hardware and memory-mapped devices, which is essential for systems programming, device drivers, and embedded systems development.
Memory Efficiency: By working directly with memory addresses, pointers can help you design algorithms and data structures that use memory more efficiently, reducing overhead and optimizing performance.

8. Write a program to copy content of one file to another destination.txt.

```
#include <stdio.h>
int main() {
   FILE *sourceFile, *destinationFile;
   char character;
   // Open the source file for reading
   sourceFile = fopen("source.txt", "r");
   if (sourceFile == NULL) {
      perror("Error opening source file");
      return 1;
   }
   // Open the destination file for writing
   destinationFile = fopen("destination.txt", "w");
   if (destinationFile == NULL) {
      perror("Error opening destination file");
      fclose(sourceFile);  // Close the source file if the destination file failed to open
      return 1;
   }
   // Copy content from source file to destination file
   while ((character = fgetc(sourceFile)) != EOF) {
      fputc(character, destinationFile);
   }
```

```
    // Close both files
    fclose(sourceFile);
    fclose(destinationFile);
    printf("File copied successfully.\n");
    return 0;
}
```

9. Explain the FORTRAN structure. What are different types of FORTRAN?

A FORTRAN program generally consists of a main program and subprograms (procedures of subroutines or functions). The subprograms will be illustrated in a later part. The structure of a nam program is:

- Program Name
- Declaration Section
- Statements
- Stop
- End

Here are the primary data types available in FORTRAN:

Integer: Represents whole numbers. The size of an integer can vary based on the system, but it's typically a 4-byte or 8-byte value.

Real: Represents floating-point numbers with single or double precision. Single precision uses 4 bytes, and double precision uses 8 bytes.

Complex: Represents complex numbers with real and imaginary parts. Complex numbers use double precision.

Character: Represents individual characters or strings of characters.

Logical: Represents logical (Boolean) values, either `.True` or `.False`

Derived Types: Fortran also supports creating custom data types using derived types. These are similar to structures or classes in other languages.

Arrays: Fortran allows you to declare arrays of any of the above data types. Arrays can be one-dimensional, multi-dimensional, or even assumed-size arrays.

Pointers: Fortran 90 and later versions introduced pointer data types that allow dynamic memory allocation and manipulation.