ABES Engineering College, Ghaziabad.

Affiliated to Dr. A.P.J Abdul Kalam Technical University, Lucknow.

Department of CSE-Data Science

Title	**Lecture Notes**
Subject	**Programming for Problem Solving**
Topics	**Function, Recursion, Pointer, Structure, Union and Enum.**
Lecture Date	**July 2024**
Faculty Name	**Mr. Dilip Kr. Bharti**
Section	**First Year Section-B**

Function

Ques:

Explain function prototype? Why is it required.

Ans:

Function Prototype

A function prototype is a declaration of a function that specifies the function's name, its return type, and its parameters, without providing the actual body of the function. It serves as a forward declaration and informs the compiler about the function's existence before its actual implementation appears in the code.

Syntax of a Function Prototype

return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...);

Example

```
#include <stdio.h>

// Function prototype
int add(int, int);

int main() {
   int a = 5, b = 3;
   int result = add(a, b);  // Function call
   printf("Result: %d\n", result);
   return 0;
}

// Function definition
int add(int x, int y) {
   return x + y;
}
```

In this example, int add(int, int); is the function prototype. It tells the compiler that there is a function named add that takes two integers as parameters and returns an integer.

Why is a Function Prototype Required?

1. **Compiler Awareness**:
   - The compiler needs to know about the function before it is called. This allows the compiler to ensure that the function is called with the correct number and types of arguments.
2. **Type Checking**:
   - The prototype provides a way to check that the arguments passed to the function are of the correct type. This helps in catching errors at compile time rather than at runtime.
3. **Code Organization**:
   - In larger programs, function definitions are often placed after the main function or even in separate files. Prototypes allow you to declare functions at the beginning or in a header file, improving code organization and readability.
4. **Forward Declaration**:
   - If two or more functions call each other, their prototypes need to be declared before any function definition to avoid compilation errors.

Example Without Function Prototype

```
#include <stdio.h>
int main() {
   int result = add(5, 3);  // Error: Implicit declaration of function 'add'
   printf("Result: %d\n", result);
   return 0;
}

int add(int x, int y) {
   return x + y;
}
```

In this case, the compiler will throw an error because it encounters a call to the add function without knowing its return type or parameter types. This demonstrates the importance of having a function prototype.

Example With Function Prototype

```
#include <stdio.h>

// Function prototype

int add(int, int);

int main() {
   int result = add(5, 3);
   printf("Result: %d\n", result);
   return 0;
}

int add(int x, int y) {
   return x + y;
}
```

Here, the function prototype int add(int, int); informs the compiler about the add function, allowing the function call in main to be correctly recognized and type-checked.

Recursion In C

Recursion:

- When a function calls a copy of itself then the process is recursion.
- Recursion can be used in case of similar subtasks like sorting, searching and traversal problem
- While using recursion will have to define an exit condition on that function if not then it will go into infinite loop.

OR

Recursion is a technique in programming where a function calls itself directly or indirectly to solve a smaller instance of the same problem.

Syntax:

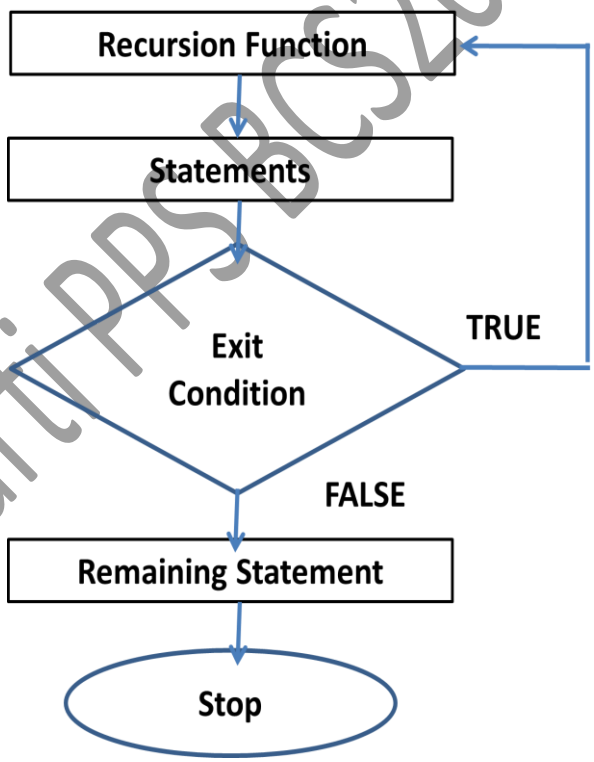
//Recursive Function:

```
int do_recursion()
{……..
do_recursion();
……..}
```

//Main function:

```
int main()
{……..
do_recursion();
……..
return 0;}
```

Recursion Path:



Type Of Recursion:

1. Direct Recursion
2. Indirect Recursion

Direct Recursion:

Direct recursion occurs when a function calls itself directly within its own definition. It typically involves a function that, during its execution, makes one or more calls to itself.

Example:

```
void count(int n)
{
   if (n <= 0)
   {
      printf(“Off!\n");
   }
else
  {
      printf("%d\n", n);
      count(n - 1); // Recursive call to count
   }
}

int main()
{
   count(5);
// Start the count from 5
   return 0;
}
```

Indirect Recursion:

Indirect recursion occurs when two or more functions call each other in a circular manner, forming a chain of function calls. Each function may not call itself directly, but it participates in a cycle of function calls.

Example:

```
void function1(int n);
void function2(int n)
{
        if (n <= 0)
        {
        return;
        }
        else
        {
      printf("%d ", n);
      function1(n - 1); // Call function1
   }
}
void function1(int n)
{
        if (n <= 0)
        {
                return;
        }
        else
```

```
        {
        printf("%d ", n);
        function2(n - 1); // Call function2
        }
}
int main()
{
   function1(5);
        // Start the recursion from     function1
   return 0;
}
```

Write a program in C to print the first 20 natural numbers using recursion.

Expected Output:

The natural numbers are:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 1819 20

Program 01:

```
#include<stdio.h>
int  print_Number(int n)
{
   if(n<=20)
   {
       printf(" %d ",n);
       print_Number (n+1);
   }
}
int main()
{
        int n = 1;
        printf("\n\n Recursion : print first 20 natural numbers :\n");
        printf("-----------------------------------------------\n");
        printf(" The natural numbers are :");
        print_Number (n);
        printf("\n\n");
        return 0;
}
```

Program 02:

Factorial of a Number

For instance,
5!=5×4×3×2×1=120

```
#include <stdio.h>
// Function to calculate factorial using recursion
int factorial(int n)
 {
   if (n == 0) {
```

```
        return 1; // Base case: 0! is 1
    }
else {
        return n * factorial(n - 1); // Recursive case
    }
}

int main() {
    int number;
    printf("Enter a positive integer: ");
    scanf("%d", &number);

    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    }
else {
        printf("Factorial of %d is %d\n", number, factorial(number));
    }

    return 0;
}
```

Sample Output

Enter a positive integer: 5
Factorial of 5 is 120

Write a program in C to print the Fibonacci Series using recursion.

Test Data :

Input number of terms for the Series (< 20) : 10

Expected Output:

Input number of terms for the Series (< 20) : 10

The Series are : 0 1 1 2 3 5 8 13 21 34

Program 03:

```
int fibo(int num)
{
        if (num==1)
        return 0;
        else if(num==2)
        return 1;
        else
        return fibo(num-1) + fibo(num-2);
}
int main()
{
        int term,i;
```

```
        printf("\n\n Recursion : Print Fibonacci Series :\n");
        printf("----------------------------------------\n");
        printf(" Input number of terms for the Series (< 20) : ");
        scanf("%d", &term);
        printf(" The Series are :\n");
        for(i=0;i<=n;i++)
        printf("%d\t",fibo(i));
        printf("\n\n");
        return 0;
}
```

Program: 04

Illustrate recursion. Write a program in C to find GCD (Greatest Common Divisor) of two numbers using recursion.

Ans:

```
#include <stdio.h>
// Function to find GCD using recursion
int gcd(int a, int b) {
   if (b == 0) {
      return a;
   }
   return gcd(b, a % b);
}

int main() {
   int num1, num2;

   printf("Enter two integers: ");
   scanf("%d %d", &num1, &num2);

   printf("GCD of %d and %d is %d\n", num1, num2, gcd(num1, num2));

   return 0;
}
```

Pointer In C

A pointer is a variable that stores the memory address of another variable. Pointers are powerful technique that enable direct memory access and manipulation, which can be crucial for tasks like dynamic memory allocation and efficient array handling.

Key Pointes of Pointers

1. **Pointer Declaration**: To declare a pointer, you use the * operator. For example, to declare a pointer to an integer, you would write int *ptr;.
2. **Pointer Initialization**: You can initialize a pointer by assigning it the address of a variable using the address-of operator &.
   For example, ptr = &var;.
3. **Dereferencing**: Dereferencing a pointer means accessing the value stored at the memory address the pointer is pointing to. This is done using the * operator.
   For example, *ptr gives you the value of the variable var if ptr is pointing to var.
4. **Pointer Arithmetic**: Pointers can be incremented or decremented, which is useful when iterating through arrays.
   For example, ptr++ moves the pointer to the next memory location of the type it points to.

Example: Using Pointers

```
#include <stdio.h>

  int main()
 {
  int var = 10;      // Declare an integer variable
  int *ptr;         // Declare a pointer to an integer

  ptr = &var;       // Assign the address of var to ptr

  // Print the value of var using the pointer
  printf("Value of var: %d\n", var);         // Output: 10
  printf("Value of var using pointer: %d\n", *ptr);  // Output: 10

  // Print the address of var
  printf("Address of var: %p\n", (void*)&var); // Output: address of var
  printf("Address stored in pointer: %p\n", (void*)ptr); // Output: address of var

  // Modify the value of var using the pointer
  *ptr = 20;
  printf("New value of var: %d\n", var);      // Output: 20

  return 0;
}
```

Explanation

1. **Pointer Declaration**: int *ptr; declares a pointer to an integer.
2. **Pointer Initialization**: ptr = &var; assigns the address of var to ptr.
3. **Dereferencing**: *ptr accesses the value stored at the address ptr is pointing to.
4. **Modifying Value**: *ptr = 20; changes the value of var via the pointer.

Using Pointers with Arrays

Pointers are especially useful with arrays, as the name of an array is a pointer to its first element.

```
#include <stdio.h>

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

   // Accessing array elements using the pointer
   for (int i = 0; i < 5; i++) {
      printf("arr[%d] = %d\n", i, *(ptr + i)); // Output: arr[0] = 10, arr[1] = 20, ...
   }

   return 0;
}
```

Explanation

1. int *ptr = arr; sets the pointer to the first element of the array.
2. *(ptr + i) accesses the ith element of the array using pointer arithmetic.

Call by Value Vs Call by Reference

- **Call by Value**: A copy of the actual parameter's value is passed to the function. Changes made to the parameter inside the function do not affect the actual parameter.

- **Call by Reference**: The address of the actual parameter is passed to the function. Changes made to the parameter inside the function affect the actual parameter.
  Here are examples demonstrating both concepts:

Call by Value:

In call by value, the actual parameter's value is copied to the function's parameter. Modifications within the function do not affect the actual parameter.

```
#include <stdio.h>
// Function prototype
void swapByValue(int a, int b);

int main() {
   int x = 10, y = 20;

   printf("Before swap (call by value): x = %d, y = %d\n", x, y);
   swapByValue(x, y);
   printf("After swap (call by value): x = %d, y = %d\n", x, y);

   return 0;
}

// Function definition
```

```
void swapByValue(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
   printf("Inside swapByValue: a = %d, b = %d\n", a, b);
}
```

Output

```
Before swap (call by value): x = 10, y = 20
Inside swapByValue: a = 20, b = 10
After swap (call by value): x = 10, y = 20
```

Explanation

- The swapByValue function swaps the values of a and b, but since a and b are copies of x and y, the original variables x and y remain unchanged.

Call by Reference

In call by reference, the address of the actual parameter is passed to the function. Modifications within the function affect the actual parameter.

```
#include <stdio.h>
// Function prototype
void swapByReference(int *a, int *b);

int main() {
   int x = 10, y = 20;

   printf("Before swap (call by reference): x = %d, y = %d\n", x, y);
   swapByReference(&x, &y);
   printf("After swap (call by reference): x = %d, y = %d\n", x, y);

   return 0;
}

// Function definition
void swapByReference(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
   printf("Inside swapByReference: *a = %d, *b = %d\n", *a, *b);
}
```

Output

```
Before swap (call by reference): x = 10, y = 20
Inside swapByReference: *a = 20, *b = 10
After swap (call by reference): x = 20, y = 10
```

Explanation

- The swapByReference function takes pointers to x and y as parameters.
- Inside the function, the values at the addresses pointed to by a and b are swapped, which directly affects x and y in the main function.

Summary

- **Call by Value**: Function receives a copy of the variable's value. Changes do not affect the original variable.

- **Call by Reference**: Function receives a reference (address) to the variable. Changes affect the original variable.
  These concepts are fundamental for understanding how functions interact with variables and how to manage memory and variable states in C programming effectively.

Ques: 01

Differentiate between call by value and call by reference. Write a program in C that computes the area and circumference of a circle with radius as input using call by reference in functions.

Ans:

Call by Value:

- **Definition**: When a function is called, the actual value of the argument is passed to the function. The called function creates its own copy of the argument, and any changes made to the parameter inside the function do not affect the original argument.
- **Example**:

```
void func(int x) {
   x = 10; // changes x only within this function
}

int main() {
   int a = 5;
   func(a);
   printf("%d", a); // Output will still be 5
   return 0;
}
```

Call by Reference:

- **Definition**: When a function is called, the address of the argument is passed to the function. The called function works with the original argument via its address, so any changes made to the parameter affect the actual argument.
- **Example**:

```
void func(int *x) {
   *x = 10; // changes the value at the address of x
}

int main() {
   int a = 5;
   func(&a);
   printf("%d", a); // Output will be 10
   return 0;
}
```

Ques: 02
Program to Compute Area and Circumference of a Circle using Call by Reference:
Ans:

```
#include <stdio.h>
#define PI 3.14

// Function to compute area and circumference
void computeCircle(float radius, float *area, float *circumference) {
   *area = PI * radius * radius;
   *circumference = 2 * PI * radius;
}

int main() {
   float radius, area, circumference;

   printf("Enter the radius of the circle: ");
   scanf("%f", &radius);

   computeCircle(radius, &area, &circumference);

   printf("Area of the circle: %.2f\n", area);
   printf("Circumference of the circle: %.2f\n", circumference);

   return 0;
}
```

Explanation:

1. **Function Definition**:
   - The computeCircle function takes three arguments: the radius of the circle, and pointers to area and circumference.
   - It calculates the area using the formula area=π×radius2\text{area} = \pi \times \text{radius}^2area=π×radius2.
   - It calculates the circumference using the formula circumference=2×π×radius\text{circumference} = 2 \times \pi \times \text{radius}circumference=2×π×radius.
   - The results are stored in the memory locations pointed to by area and circumference.
2. **Main Function**:
   - It prompts the user to enter the radius of the circle.
   - It reads the radius and stores it in radius.
   - It calls the computeCircle function with radius, and addresses of area and circumference.
   - It prints the computed area and circumference.

Structures, Unions, and Enumerated data types

1. Structures

A structure in C is a user-defined data type that groups different data types under a single name. It allows combining variables of different types into a single logical unit.

Example: Structure

```
#include <stdio.h>
// Define a structure named 'Person'
struct Person {
   char name[50];
   int age;
   float height;
};

int main() {
   // Declare and initialize a structure variable
   struct Person person1 = {"Deepak", 25, 5.7};

   // Access and print structure members
   printf("Name: %s\n", person1.name);
   printf("Age: %d\n", person1.age);
   printf("Height: %.2f\n", person1.height);

   return 0;
}
```

Explanation

- struct Person defines a structure with three members: name, age, and height.
- person1 is a variable of type struct Person and is initialized with values.
- Structure members are accessed using the dot operator (.).

Program:

To find the topper student information using a structure, we can define a structure to represent a student, store the details of multiple students, and then iterate through the list to determine the student with the highest marks.

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

// Define a structure named 'Student'
struct Student {
   char name[50];
   int rollNumber;
```

```
    float marks;
};

// Function to find the topper student
struct Student findTopper(struct Student students[], int n) {
    struct Student topper = students[0];

    for (int i = 1; i < n; i++) {
        if (students[i].marks > topper.marks) {
            topper = students[i];
        }
    }

    return topper;
}

int main() {
    int n;

    printf("Enter the number of students: ");
    scanf("%d", &n);

    struct Student students[n];

    // Input student details
    for (int i = 0; i < n; i++) {
        printf("\nEnter details for student %d:\n", i + 1);
        printf("Name: ");
        scanf("%s", students[i].name);
        printf("Roll Number: ");
        scanf("%d", &students[i].rollNumber);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }

    // Find the topper student
    struct Student topper = findTopper(students, n);

    // Print the topper student details
    printf("\nTopper Student Information:\n");
    printf("Name: %s\n", topper.name);
    printf("Roll Number: %d\n", topper.rollNumber);
    printf("Marks: %.2f\n", topper.marks);

    return 0;
}
```

Explanation

1. **Structure Definition**:

- The Student structure contains three members: name (string), rollNumber (integer), and marks (float).

2. **Input**:
- The user is prompted to enter the number of students.
- Details for each student (name, roll number, and marks) are collected and stored in an array of Student structures.

3. **Finding the Topper**:
- The findTopper function iterates through the array of students and finds the student with the highest marks.
- The topper student's information is returned to the main function.

4. **Output**:
- The details of the topper student are printed.

Sample Output

Enter the number of students: 3

Enter details for student 1:
Name: Deepak
Roll Number: 101
Marks: 85.5

Enter details for student 2:
Name: Rohit
Roll Number: 102
Marks: 92.3

Enter details for student 3:
Name: Samar
Roll Number: 103
Marks: 78.4

Topper Student Information:
Name: Rohit
Roll Number: 102
Marks: 92.30

This program demonstrates how to use structures to store and process complex data and how to find specific information based on given criteria.

2. Unions

A union in C is similar to a structure, but it allows storing different data types in the same memory location. Only one member can hold a value at any given time, and the memory allocated is the size of the largest member.

Example: Union

c

Copy code

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

```
// Define a union named 'Data'
union Data {
   int i;
   float f;
   char str[20];
};

int main() {
   // Declare a union variable
   union Data data;

   // Assign and print integer value
   data.i = 10;
   printf("data.i: %d\n", data.i);

   // Assign and print float value
   data.f = 220.5;
   printf("data.f: %.2f\n", data.f);

   // Assign and print string value
   strcpy(data.str, "Hello");
   printf("data.str: %s\n", data.str);

   return 0;
}
```

Explanation

- union Data defines a union with three members: i, f, and str.
- The same memory location is used to store different types of data.
- Only one member can be accessed at a time, and assigning a new value overwrites the previous value.

3. Enumerated Data Types (enum)

An enumerated data type (enum) in C is a user-defined type that consists of integral constants. It assigns names to the integral constants to make a program more readable.

Example: Enumerated Data Types

c

Copy code

```
#include <stdio.h>

// Define an enumeration named 'Day'
enum Day {
   SUNDAY,
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY
};
```

```
int main() {
    // Declare a variable of type enum Day
    enum Day today;

    // Assign a value to the variable
    today = WEDNESDAY;

    // Print the value
    printf("Today is day number %d\n", today);

    return 0;
}
```

Explanation

- enum Day defines an enumeration with named integral constants representing days of the week.
- The variable today is of type enum Day and is assigned the value WEDNESDAY.
- Enumerations help make the code more readable and maintainable by using meaningful names instead of numeric values.

Summary

- **Structures** group different types of data under a single name, allowing the management of related data items.
- **Unions** allow storing different data types in the same memory location, with only one member holding a value at a time.
- **Enumerated Data Types (enum)** assign names to integral constants, enhancing code readability and maintainability.