Dr. SNS RAJALAKSHMI COLLEGE OF ARTS AND SCIENCE

(AUTONOMOUS)

Accredited by NAAC (Cycle- III) with 'A+' Grade

DEPARTMENT OF B.SC CS (GCD)

23UCU401 – PROGRAMMING IN C

UNIT- III

Subject : Programming in C
Department : GCD
Faculty : Mrs.K.Sangeetha

Unit 3 :

What is a Function - Passing Values between Functions - Scope Rule of Functions - Calling Convention - Function Declaration and Prototypes - Call by Value and Call by Reference - An Introduction to Pointers -Pointer Notation -Back to Function Calls - Recursion - Recursion and Stack.

1. What is a Function

- A function is a self-contained block of code that performs a specific task.
- Functions make your code modular and easier to maintain.

Example:

```
// A simple function that adds two
numbers
int add(int a, int b) {
    return a + b;
}
```

Explanation:

- The `add` function takes two integers as input, adds them, and returns the result.
- This function can be called from other parts of the program to perform addition.

2.Passing Values between Functions

- You can pass data to a function as arguments or parameters.
- Functions can return values back to the calling code.

Example:

```
// Function that calculates the square of a number
int square(int x) {
    return x * x;
}

int main() {
    int num = 5;
    int result = square(num);
    // 'result' now holds the square of 'num'
    return 0;
}
```

Explanation:

- In the `main` function, we pass the value of `num` to the `square` function.
- The `square` function calculates the square and returns the result to the `main` function.

3.Scope Rule of Functions

- Scope defines where a variable is accessible.
- Variables declared within a function have local scope, while those declared outside have global scope.

Example:

```
int globalVar = 10;  // Global variable

int main() {
    int localVar = 5;  // Local variable
    return 0;
}
```

Explanation:

- `globalVar` is accessible throughout the program.
- `localVar` is only accessible within the `main` function.

4.Calling Convention

A calling convention in C is a set of rules that define how a function is called and how arguments are passed to and returned from the function. The calling convention specifies the following:

- How the function arguments are pushed onto the stack
- How the return value is returned to the caller
- How the function registers are saved and restored

```
#include <stdio.h>

int my_function(int a, int b) {
  return a + b;
}

int main() {
  int result = my_function(1, 2);

  printf("The result is %d.\n", result);

  return 0;
}
```

5.Function Declaration and Prototypes

- Function prototypes are declarations that specify the function's name, return type, and parameters.
- Prototypes help the compiler verify function calls.

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

int main() {
    int result = add(3, 4);
    return 0;
}

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

Explanation:

- The function prototype `int add(int a, int b);` informs the compiler about the `add` function's signature before it's defined.
- This allows the `main` function to call `add` even before its actual implementation.

6. Call by Value and Call by Reference

- Call by Value passes a copy of the argument to the function, so changes don't affect the original.
- Call by Reference passes a reference or address, allowing changes to affect the original data.

Example of Call by Value:

```
void modifyValue(int x) {
    x = x * 2;
}

int main() {
    int num = 5;
    modifyValue(num);
    // 'num' remains 5; no change occurred
    return 0;
}
```

Example of Call by Reference (using pointers):

```
void modifyValue(int* x) {
    *x = *x * 2;
}

int main() {
    int num = 5;
    modifyValue(&num);
    // 'num' is now 10; it was changed within the function
    return 0;
}
```

Explanation:

- In Call by Value, the function receives a copy of the argument, so any changes are local to the function.
- In Call by Reference, the function receives the address of the variable, allowing changes to affect the original data.

7. An Introduction to Pointers

- Pointers are variables that store memory addresses.
- They are used to manipulate data indirectly and can be powerful but require careful handling.

Example:

```
int num = 10;
int* ptr = &num; // 'ptr' stores the address of 'num'
*ptr = 20;       // Changes the value of 'num' through the pointer
```

Explanation:

- `ptr` is a pointer that stores the address of the `num` variable.
- `*ptr` is used to access and modify the value of `num` indirectly through the pointer.

8. Pointer Notation

- Pointer notation allows you to access data through pointers.
- `*` is used to dereference a pointer and obtain the value it points to.

```
int num = 5;
int* ptr = &num;
int value = *ptr; // 'value' now holds the value of 'num'
```

Explanation:

- `*ptr` retrieves the value that `ptr` points to, which is the value of `num`.

9. Back to Function Calls

- When a function is called, control jumps to the function, and local variables are created.
- When the function returns, control and data return to the calling code.

Explanation:

- When a function is called, a new stack frame is created with local variables.
- When the function returns, the stack frame is removed, and control returns to the calling code.

10. Recursion - Recursion and Stack

- Recursion is a technique where a function calls itself.
- Each recursive call uses its own stack frame, and the stack keeps track of multiple function calls.

Example:

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Explanation:

- The `factorial` function calculates the factorial of a number using recursion.
- Each recursive call creates a new stack frame, and the results are combined to compute the final result.