Unit -3

C-Program

Topic: Functions

What is Function?

A **function** is a self-contained block of code that performs a specific task. It helps in **dividing a large program** into smaller, manageable parts, making the program easier to understand, debug, and reuse.

C programs are usually made up of many functions, one of which must be main().
Execution of every C program **starts from the main() function**.

Why use functions?

- Reusability: Write once, use many times.
- Modularity: Divide complex problems into simple parts.
- Easier maintenance: Fix one function without touching the whole program.

Syntax:

```
return_type function_name (parameter_list) {
   // Body of the function
   return value;  // Optional, if return_type is void, no return needed
}
```

Function Declaration and Definition:

1) Function Declaration (Prototype):

A **function declaration** tells the compiler about the **function name**, **return type**, and **parameters** before it is used.
It helps the compiler understand how to handle function calls.

Syntax:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

Example:

```
#include <stdio.h>


int add(int, int);   // Function Declaration
int main() {
   int result = add(5, 10);   // Function Call
   printf("Sum = %d", result);
   return 0;
}
int add(int a, int b) {   // Function Definition
  return a + b;
}
```

2) Function Prototype:

A prototype is simply another name for function declaration. It ensures type checking during compilation.

```
float area(float radius); // Function prototype
```

Types of Functions?

There are two main types:

a) Library Functions
b) User-Defined Functions

a) Library Function:

These are pre-built functions provided by the C standard library. You don't write them; you just include the appropriate header file (using #include) and use them.

Here are some common library function:

Function	Purpose	Header File
printf()	Output to screen	<stdio.h>
scanf()	Input from user	<stdio.h>
strlen()	Finds length of a string	<string.h>
sqrt()	Square root	<math.h>
pow()	Power calculation	<math.h>
toupper()	Converts character to uppercase	<ctype.h>

Example:

```
#include <stdio.h>

#include <math.h>

int main() {

printf("Square root of 16 is: %.2f", sqrt(16));

return 0;

}
```

b) User Defined Function:

These are **custom functions** created by the programmer to perform specific tasks.

Key Components:

- **Declaration/Prototype:** Specifies the function's name, return type, and parameters (optional).
- **Definition:** Contains the actual code (body) of the function.
- **Call:** Invokes the function from another part of the program (e.g., main())

Advantages:

- Code reusability: Call the function multiple times.

- Modularity: Easier debugging and testing.
- Abstraction: Hide implementation details.

Disadvantages:

- Overhead: Slight performance cost due to function calls.
- Potential for errors: If not properly declared or defined.

Syntax:

```
return_type function_name(parameter_list) {
   // body of the function
}
```

Example:

```
#include <stdio.h>
int juce() {
   printf("JAMIA BCA STUDENT");
}
int main() {
   juce();  // function call
   return 0;
}
```

Inline and Macro Functions:

Macro Function: A macro is a preprocessor directive (starting with #define) that performs textual substitution before the code is compiled. The preprocessor replaces every occurrence of the macro name with its defined body, effectively inlining the code at the call site.

Syntax:

```
#define MACRO_NAME(parameters) body
```

Advantages

- **Zero Runtime Overhead**: Ideal for simple, frequently called operations.
- **Flexible**: Can generate code dynamically (e.g., using __LINE__ or __FILE__ for debugging).
- **Conditional Compilation**: Supports #ifdef, #ifndef for platform-specific code.
- **Constants**: Efficient for compile-time constants (e.g., #define BUFFER_SIZE 1024).

Disadvantages

- **No Type Safety**: Treats everything as text; no compile-time type checking. E.g., passing a float to MAX(int, int) could cause issues.
- **Side Effects and Precedence Issues**: Arguments are evaluated multiple times (e.g., MAX(i++, j) increments i twice). Operator precedence can break if parentheses are missing.
- **Debugging Nightmares**: Expanded code bloats the binary; stepping through in a debugger shows raw expansions, not the macro name.
- **Name Collisions**: Global scope; can clash with other identifiers.
- **Undefined Behavior**: Relies on programmer discipline; errors surface late (at link/runtime).

Example:

Example 1: #define PI 3.14159

Example 2: #define SQUARE(x) ((x) * (x))

Example 3: #include <stdio.h>

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int main() {
   int x = 5, y = 10;
```

```
    printf("Max of %d and %d is %d\n", x, y, MAX(x, y));  // Expands to: ((x) > (y) ? (x) : (y))
    return 0;
}
```

Inline Function: An inline function is a regular function marked with the inline keyword, which serves as a hint to the compiler to insert the function's body directly at the call site, similar to a macro. Unlike macros, it's a true function with type checking, and the compiler decides whether to inline based on heuristics (e.g., function size, call frequency).

Syntax:

```
inline return_type function_name(parameters) {
    // body
}
```

Example:

```
#include <stdio.h>
inline int max(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
    int x = 5, y = 10;
    printf("Max of %d and %d is %d\n", x, y, max(x, y));  // Compiler may inline to: (x > y ? x : y)
    return 0;
}
```

Types of Arguments: Actual and Formal Arguments:

Arguments are the values passed to a function. There are two type:

a) **Formal Arguments (Parameters):** These are the variables defined in the function's prototype and definition. They're placeholders inside the function.

Example: In int add(int x, int y), x and y are formal arguments.

b) Actual Arguments (Arguments):

These are the real values passed when calling the function.

Example: In add(5, 3), 5 and 3 are actual arguments.

Example of Addition and substraction two numbers using function

```
#include <stdio.h>
int add(int a, int b) {     // a and b are formal arguments
   return a + b;
}
int subtract(int a, int b) {
   return a - b;
}
int main() {
   int num1, num2, sum, difference;
   printf("Enter first number: ");
```

```
  scanf("%d", &num1);


  printf("Enter second number: ");
  scanf("%d", &num2);
  sum = add(num1, num2);
  difference = subtract(num1, num2);    //num1 and num2 are actual arguments


  printf("\nAddition of %d and %d = %d", num1, num2, sum);
  printf("\nSubtraction of %d and %d = %d\n", num1, num2, difference);
  return 0; }
```

Passing an Array as Parameters:

In **C programming**, when we pass an array to a function, we are **actually passing the address (reference) of its first element**, not a copy of the whole array.
That means:

- The function can **access and modify** the elements of the original array.
- Arrays are always **passed by reference**, not by value.

```
#include <stdio.h>
// Function that takes an array and its size
int BCA(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);  // accessing array elements
   }
   printf("\n");
}
```

```
int main() {
   int numbers[] = {10, 20, 30, 40, 50};
   int size = 5;
   BCA(numbers, size);   // Passing array to function
   return 0;
}
```

Methods to Call a Function: (Call by value and Call by reference)

a) Call by Value:

- A **copy** of the actual argument is passed to the function.
- Changes made inside the function **do not affect** the original value.

Example:

```
#include <stdio.h>

int modify(int x) {
   x = x + 10;
}

int main() {
   int a = 5;
   modify(a);
   printf("Value of a = %d", a);
   return 0;
}
```

Output: Value of a = 5

b) Call by Value:

- • **Address** of the actual argument is passed.
- • Changes made inside the function **affect the original value**.

Example:

```
#include <stdio.h>

int modify(int *x) {
   *x = *x + 10;
}

int main() {
   int a = 5;
   modify(&a);
   printf("Value of a = %d", a);
   return 0;
}
```

Output: Value of a = 15

Extras:

You can write function with these methods:

-No Arguments and No Return Value	-With Arguments but No Return Value

int functionName() {
 	
}
 	
int main() {
functionName();
}
	int functionName(dataType arg1, dataType arg2) {
	}
	
	int main() {
	functionName(value1, value2);
	}

-With Arguments and With Return Value	-No Arguments but With Return Value

Int functionName(dataType arg1, dataType arg2) {
return value;
}
 	
int main() {
dataType result = functionName(value1, value2);
}
	dataType functionName() {
	
	return value;
	}
	
	int main() {
	dataType result = functionName();
	}