C FUNCTIONS

Monolithic vs Modular Programming

- Monolithic: Single function for large program
- No division → difficult to maintain
- Error checking is hard
- Modular: Divides program into small modules
- Each module tested independently
- Linker combines modules

Monolithic vs Modular (Pros & Cons)

- Disadvantages of Monolithic:
- - Difficult to debug, maintain
- - Code not reusable
- Advantages of Modular:
- - Easy to code & debug
- - Reduces program size
- - Reusable code
- - Errors localized to modules

What is a Function?

- Group of statements performing a task
- Every C program has at least one function: main()
- Two types: User Defined & Standard Library

Function Declaration (Prototype)

- Informs compiler about function name, arguments, return type
- Syntax: return_type function_name(type1 arg1, type2 arg2);

Function Definition

- Contains actual code of function
- Parts: Function header + body
- Syntax:

  return_type function_name(type1 arg1, type2 arg2) { ... }

Function Categories

- 1. No arguments, No return value
- 2. No arguments, With return value
- 3. With arguments, No return value
- 4. With arguments, With return value

Example – No Arguments & No Return

```
void link(void);
int main()
 {
    link();
 }
void link(void)
{
printf("Link the file");
}
```

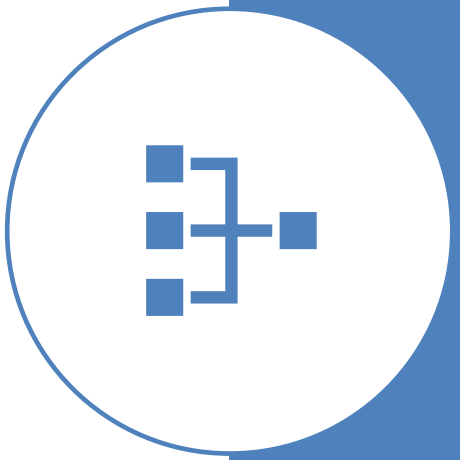
Example – No Arguments & Return Value

```
int addfn(void);
int main()
{
 int s = addfn();
 printf("Sum = %d", s);
}
int addfn(void)
{
int a,b,sum;
sum=a+b;
return sum;
}
```

Example – With Arguments & No Return

```
void msg(int a, int b);
int main() { int a=2,b=3; msg(a,b);}
void msg(int a,int b){ int s=a+b; printf("Sum=%d",s);}
```

Example – With Arguments & Return Value



- int msg(int a, int b);
- int main(){ int a=2,b=3; int s=msg(a,b); printf("Sum=%d",s);}
- int msg(int a,int b){ return a+b;}

Actual vs Formal Arguments

Actual: Values passed in function call e.g., fun(6,9)

Formal: Variables in function definition e.g., int fun(int x,int y)

Formal arguments act as local variables

Parameter Passing Techniques

Call by Value: Copy of variable passed, no effect on caller

Call by Reference: Address passed using pointers, changes affect original

Example – Call by Value

Void	void swap(int a,int b){int t=a;a=b;b=t;}
Swap	int main(){int k=50,m=25; swap(k,m); printf("%d %d",k,m);}
Output	Output: 50 25

Example – Call by Reference

```
void add(int *n)
{ *n=*n+10;
printf("In function=%d",*n);}
int main()
{ int num=2;
printf("Before=%d",num);
add(&num); printf("After=%d",num);}
```

Output: Before=2, In function=12, After=12

Example – Largest of Two Numbers

```
int large(int p, int q)
{                                   if(p>q)
return p;
else
return q;
}
```

Example – Factorial

```
int factorial(int n)
        {
          int i,p=1;
          for(i=n;i>1;i--)
           p*=i;
          return p;
        }


int main()
   {
      int a;
      scanf("%d",&a);
      printf("Factorial=%d",factorial(a));
   }
```

Summary

- Functions support modular programming
- Easier debugging & maintenance
- Categories: 4 types of functions
- Parameter passing: Call by Value & Call by Reference