Structured Programming-functions

Modular programming and Functions

L 2 2 - L 2 3

Objectives:

To learn and appreciate the following concepts

- To understand scope of variables
- Modularization and importance of modularization
- Understand how to define and invoke a function
- Understand the flow of control in a program involving function call
- Understand the different categories of functions
- Write programs using functions

Session outcome:

At the end of session one will be able to

- Understand modularization and function
- Write simple programs using functions

Programming Scenario . . .

Lengthier programs

- Prone to errors
- tedious to locate and correct the errors

To overcome this

Programs broken into a number of smaller logical components, each of which serves a specific task.

Modularization

- Process of splitting the lengthier and complex programs into a number of smaller units is called **Modularization.**

- Programming with such an approach is called **Modular programming**

Advantages of modularization

- Reusability
- Debugging is easier
- Build library
- Makes programs easier to understand

Functions

- A **function** is a set of instructions to carryout a particular task.

- Using functions we can structure our programs in a **more modular** way.

Functions

- Standard functions

  (library functions or built in functions)

- User-defined functions

  Written by the user(programmer)

General form of function definition

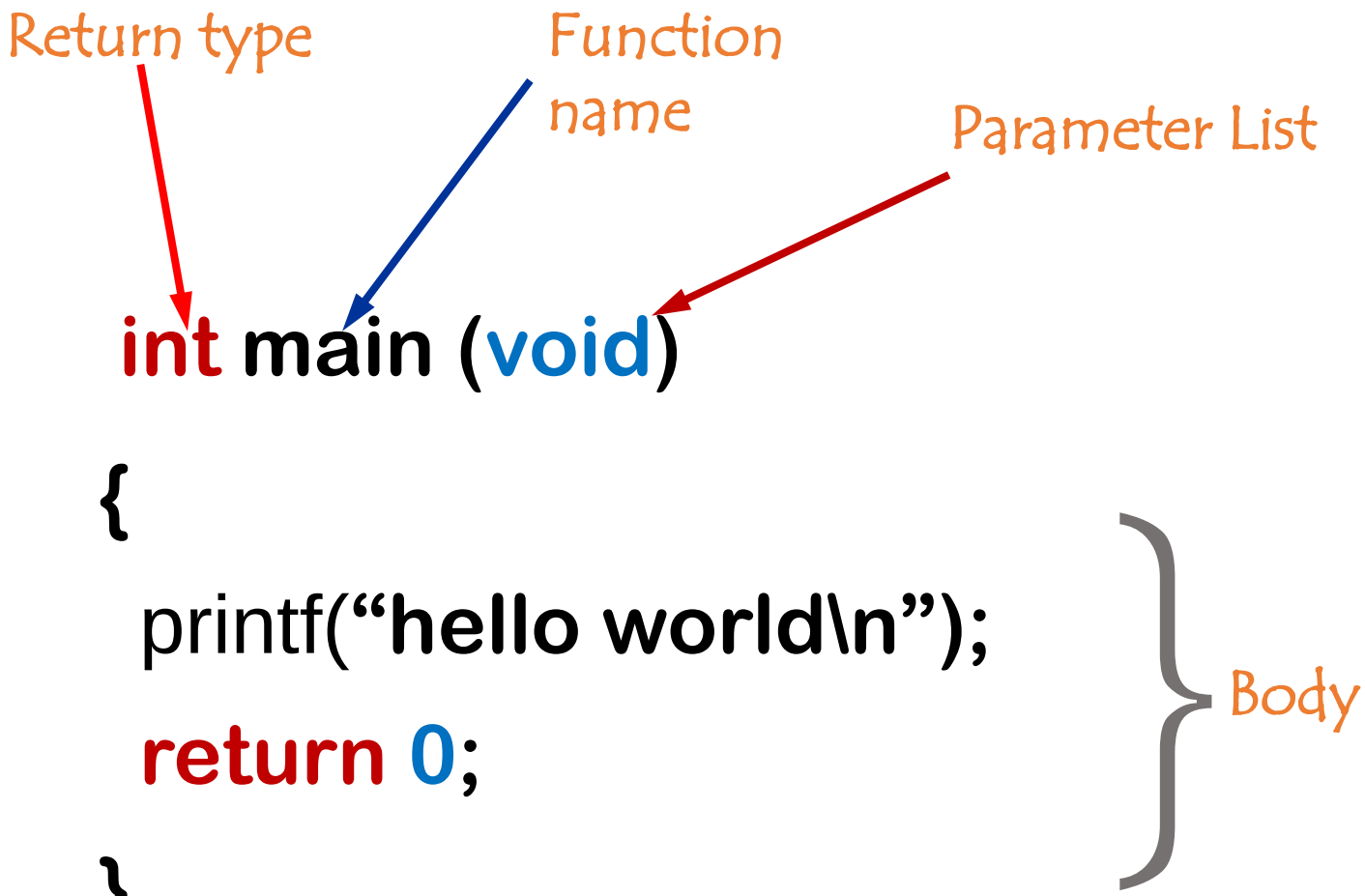
```
return_type  function_name(parameter_definition)
 {
    variable declaration;

    statement1;
    statement2;
            .
            .
            .
    return(value_computed);
 }
```

Defining a Function

- ✓Name (function name)
  - You should give functions descriptive names
  - Same rules as variable names, generally
- ✓Return type
  - Data type of the value returned to the part of the program that activated (called) the function.
- ✓Parameter list (parameter_definition)
  - A list of variables that hold the values being passed to the function
- ✓Body
  - Statements enclosed in curly braces that perform the function's operations(tasks)

Understanding `main( )` function



Function Definition and Call

```
// FUNCTION DEFINITION
```

Return type Function name Parameter List

```
void DisplayMessage(void)
{
        printf("Hello from function DisplayMessage\n");
}
int main()
{
        printf("Hello from main \n");
        DisplayMessage();   // FUNCTION CALL
        printf("Back in function main again.\n");
        return 0;
 }
```

Multiple Functions- An example

```
void First (void){          // FUNCTION DEFINITION
        printf("I am now inside function First\n");
}
 void Second (void){        // FUNCTION DEFINITION
        printf( "I am now inside function Second\n");
        First();            // FUNCTION CALL
        printf("Back to Second\n");
}
int main (){
        printf( "I am starting in function main\n");
        First ();           // FUNCTION CALL
         printf( "Back to main function \n");
        Second ();          // FUNCTION CALL
        printf( "Back to main function \n");
        return 0;
}
```

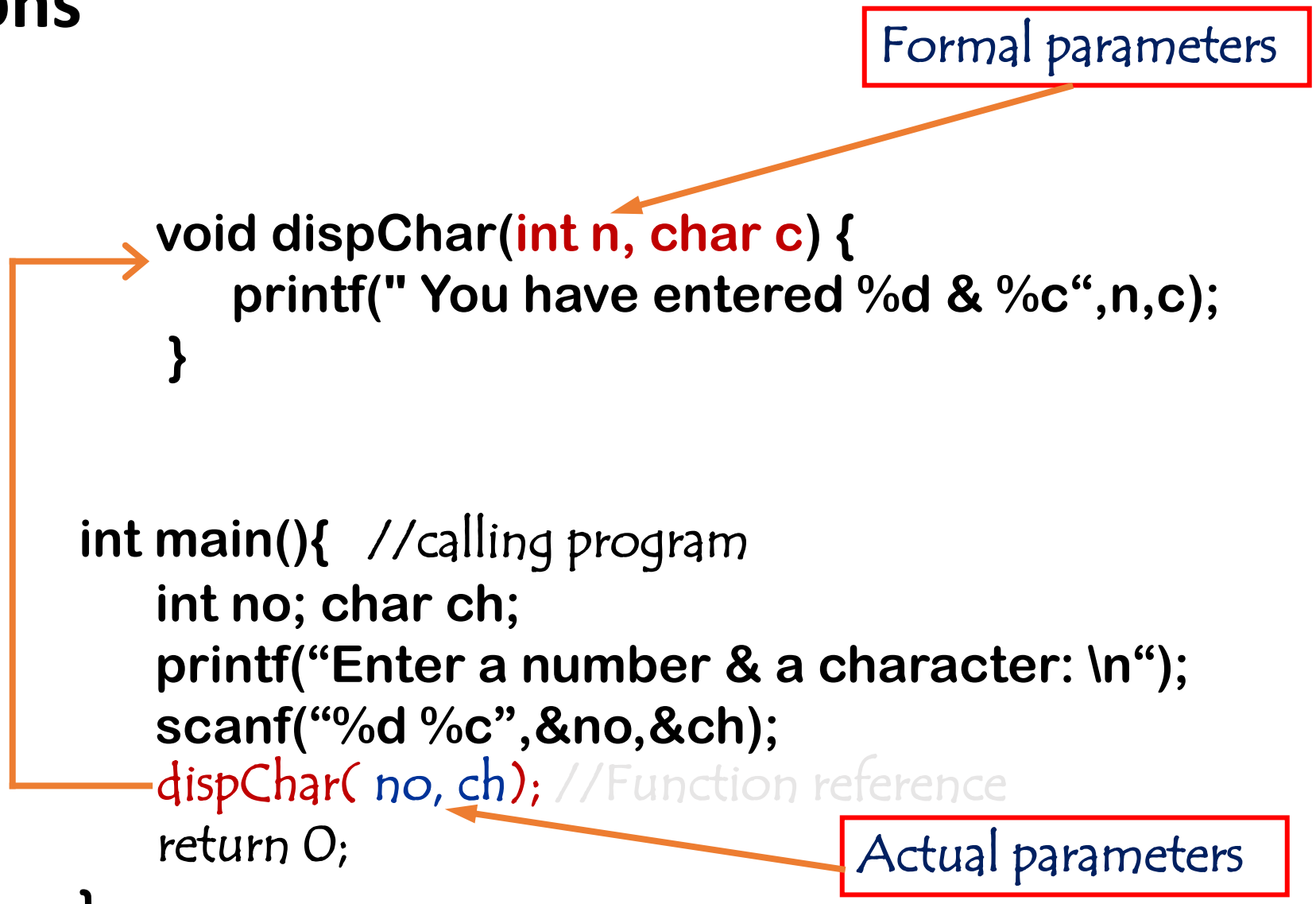
Arguments and Parameters

- Both arguments and parameters are variables used in a **program** & **function.**

- Variables used in the *function reference* or *function call* are called as **arguments**. These are written within the parenthesis followed by the name of the function. They are also called actual parameters.

- Variables used in *function definition* are called **parameters**, They are also referred to as formal parameters.

Functions


Formal parameters
void dispChar(int n, char c) {
printf(" You have entered %d & %c“,n,c);
}
int main(){ //calling program
int no; char ch;
printf(“Enter a number & a character: \n“);
scanf(“%d %c”,&no,&ch);
dispChar(no, ch); //Function reference
return 0;
}
Actual parameters

Function Prototypes

- Must be included for each function that will be defined, (required by Standards for C++ but optional for C) if not directly defined before main().
- In most cases it is recommended to include a function prototype in your C/C++ program to avoid ambiguity.
- **Identical** to the function header, with semicolon (;) added at the end.

- Function prototype (declaration) includes
  - Function name
  - Parameters – what the function takes in and their type
  - Return type – data type function returns (default **int**)
- Parameter names are **Optional**.

Function Prototypes

- Function prototype provides the compiler the name and arguments of the functions and must appear before the function is used or defined.
- It is a model for a function that will appear later, somewhere in the program.
- General form of the function prototype:

fn_return_type fn_name(type par1, type par2, …, type parN);

- Example:

int maximum(int, int, int);

  - Takes in 3 **int**s
  - Returns an **int**

Scope of Variables

- A scope is a region of the program where a defined variable can have its existence and beyond that it cannot be accessed.

- The two types of variables are

  1) **local** variables

  2) **global** variables

Local Variables

- Variables that are declared inside a function are called local variables.
- They can be used only by statements that are inside that function.
- In the following example all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
 int main () {
   /* local variable declaration */
    int a, b, c;
    a = 10;  b = 20; c = a + b;
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
    return 0;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

```
#include <stdio.h>
int g; /* global variable declaration */
int main () {
 int a, b; /* local variable declaration */
 a = 10; b = 20; g = a + b;
 printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
 return 0;
}
```

Functions- points to note

1. The parameter list must be separated by commas.

```
dispChar( int n, char c);
```

2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.

```
void dispChar(int n, char c); //proto-type
void dispChar(int num, char ch){
    printf(" You have entered %d &%c", num,ch);
 }
```

4. Use of parameter names in the declaration(prototype) is optional but parameter type is a must.

```
void dispChar(int , char); //proto-type
```

Functions- points to note

5. If the function has no formal parameters, the list can be written as (void) or simply ()
6. The return type is optional, when the function returns **integer** type data.
7. The return type must be **void** if no value is returned.
8. When the declared types do not match with the types in the function definition, compiler will produce error.

Functions- Categories

Categorization based on the arguments and return values

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and one return value.
4. Functions with no arguments but return a value.
5. Functions that return multiple values

   (will see later with parameter passing techniques).

Function with No Arguments/parameters & No return values

```
void dispPattern(void); // prototype

int main(){
        printf("fn to display a line of stars\n");
        dispPattern();
        return 0;
}
void dispPattern(void ){
        int i;
        for (i=1;i<=20 ; i++)
                printf( "*");
}
```

Function with No Arguments but A return value

```
int readNum(void); // prototype

int main(){
        int c;
        printf("Enter a number \n");
        c=readNum();
        printf("The number read is %d",c);
        return 0;
}
int readNum(){
        int z;
        scanf("%d",&z);
        return(z);
}
```

Fn with Arguments/parameters & No return values

```
void dispPattern(char ch); // prototype

int main(){
        printf("fn to display a line of patterns\n");
        dispPattern('#');
        dispPattern('*');
        dispPattern('@');
        return 0;
}
void dispPattern(char ch ){
        int i;
        for (i=1;i<=20 ; i++)
                printf("%c",ch);
        }
```

Function with Arguments/parameters & One return value

```
int main(){
        int a,b,c;
        printf("\nEnter numbers to be added\n");
        scanf("%d %d",&a,&b);
        c=fnAdd(a,b);
        printf("Sum is %d ", c);
        return 0;
}
int  fnAdd(int x, int y ){
        int z;
        z=x+y;
        return(z);
}
```

Problems…

Write appropriate functions to

1. Find the factorial of a number ‘n’.
2. Reverse a number ‘n’.
3. Check whether the number ‘n’ is a palindrome.
4. Generate the Fibonacci series for given limit ‘n’.
5. Check whether the number ‘n’ is prime.
6. Generate the prime series using the function written for prime check, for a given limit.

Factorial of a given number ‘n’

```
long  factFn(int); //prototype

int main() {
   int n, f;

    printf("Enter a number :“);
    scanf(“%d”,&n);
    f =factFn(n);
    printf(“Fact= %ld“,f);

    return 0;
}
```

```
//function definition
long factFn(int num) {
int i;
    long fact=1;

    //factorial computation
    for (i=1; i<=num; i++)
         fact=fact * i;

    // return the result
    return (fact);
}
```

Reversing a given number ‘n’

```
int Reverse(int); //prototype

int main()
{
  int n,r;
  printf("Enter a number : \n“);
  scanf(“%d”, &n);

 r=  Reverse(n);
 printf(“ reversed no=%d“,r)

 return 0;
}
```

```
int Reverse(int num)
{
    int rev=0;
    int digit;

    while(num!=0)
    {
       digit = num % 10;
       rev = (10 * rev) + digit;
       num = num/10;
    }
    return (rev);
}
```

Check whether given number is prime or not

```
int IsPrime(int); //prototype

int main() {
  int n;

  printf("Enter a number : ");
  scanf("%d",&n);
  if (IsPrime(n))
     Printf("%d is a prime no",n);
  else
     Printf("%d is not a prime no",n);
  return 0;
}
```

```
int IsPrime(int num) //prime check
{
    int p=1;
    for(int j=2;j<=num/2;j++) //
change
    {
        if(num%j==0)
        {
            p=0;
            break;
        }
    }
    return p;
}
```

First n Fibonacci number generation

```
void fibFn(int); //prototype

int main() {

    int n;

    printf("Enter the limit  ”);

    scanf(“%d”,&n);

    fibFn(n); //function call

    return 0;

}
```

```
void fibFn(int lim) { //fib generation
    int  i, first, sec, next;
    if (lim<=0)
      printf("limit should be +ve.\n“);
    else {
      printf("\nFibonacci nos\n“);
      first = 0, sec = 1;
      for (i=1; i<=lim; i++) {
         printf(“%d”, first)
         next = first + sec;
         first = sec;
         sec = next;
      }
    }
}
```

Summary

- Modularization and importance of modularization
- Defining and invoking a function
- Flow of control of a program involving function call
- Different categories of functions
- Simple programs using functions