SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.
An Autonomous Institution

COURSE NAME : 19CST101 PROGRAMMING FOR PROBLEM SOLVING

I YEAR/ I SEMESTER

UNIT-IV FUNCTIONS AND POINTERS

Topic: Functions

Ms. Sumathi B
Assistant Professor
Department of Computer Science and Engineering

Functions

User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

The execution of a C program begins from the `main()` function.

When the compiler encounters `functionName();`, control of the program jumps to

```
void functionName()
```

And, the compiler starts executing the codes inside `functionName()`.

The control of the program jumps back to the `main()` function once code inside the function definition is executed.

Functions

How user-defined function works?

```
#include <stdio.h>
void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Functions

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

Functions

ELEMENTS OF USER-DEFINED FUNCTIONS

- **Function declaration or prototype** – informs compiler about the function name, function parameters and return value's data type.
- **Function call** – This calls the actual function
- **Function definition** – This contains all the statements to be executed.

Sno	C Function aspects	Syntax
1	Function definition	return_type function_name(arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

Functions

ELEMENTS OF USER–DEFINED FUNCTIONS

- There are **three elements** related to functions
  - Function definition
  - Function call
  - Function declaration
- **The function definition** is an independent program module that is specially written to implement the requirements of the function
- To use this function we need to invoke it at a required place in the program. This is known as **the function call.**
- The program that calls the function is referred to as the **calling program or calling function.**
- The calling program should declare any function that is to be used later in the program. This is known as the **function declaration or function prototype.**

FUNCTION DECLARATION

- Like variables, all functions in a C program must be declared, before they are invoked.
- A function declaration (also known as function prototype) consists of **four parts**.
    - Function type (return type).
    - Function name.
    - Parameter list.
    - Terminating semicolon.
- They are coded in the following format:
    - Function-type function-name (parameter list);
- This is very similar to the function header line except the terminating semicolon.
- For example, mul function defined in the previous section will be declared as:
    - int mul (int m, int n); /* Function prototype */

FUNCTION DECLARATION

Points to Note

1. The parameter list must be separated by commas.
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.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns int type data.
7. The retype 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 an error.

FUNCTION DECLARATION

A prototype declaration may be placed in **two places** in a program.

1. Above all the functions (including main).
2. Inside a function definition.

When we place the declaration above all the functions (**in the global declaration section**), the prototype is referred to as a global prototype.

Such declarations are available **for all the functions** in the program.

When we place it in a function definition (**in the local declaration section**), the prototype is called a local prototype.

Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions.

This region is known as the scope of the function.

It is a good programming style to declare prototypes in the global declaration section before main.

It adds flexibility, provides an excellent quick reference to the functions used in the program, and enhances documentation.

FUNCTION DECLARATION

Prototypes: Yes or No

Prototype declarations are not essential.

If a function has not been declared before it is used, C will assume that its details available at the time of linking.

Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions.

If these assumptions are wrong, the linker will fail and we will have to change the program.

The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in following three places:

1. in declaration (prototypes),
2. in function call, and
3. in function definition.

The parameters used in prototypes and function definitions are called **formal parameters** and those used in **function calls are called actual parameters**.

Actual parameters used in a **calling statement** may be simple constants, variables, or expressions.

The formal and actual parameters must match exactly in type, order and number.

Their names however, do not need to match.

CATEGORY OF FUNCTIONS

- A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

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

No Arguments and No Return Values

- When a function has no arguments, it does not receive any data from the calling function.
- Similarly, when it does not return a value, the calling function does not receive any data from the called function.
- In effect, there is no data transfer between the calling function and the called function.
- This is depicted in Fig.
- The dotted lines indicate that there is onlv a transfer of control but not data.

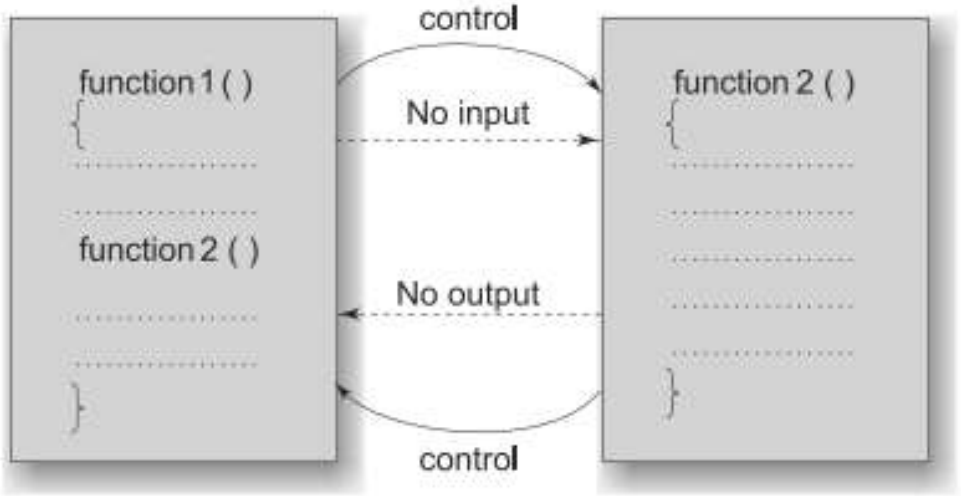
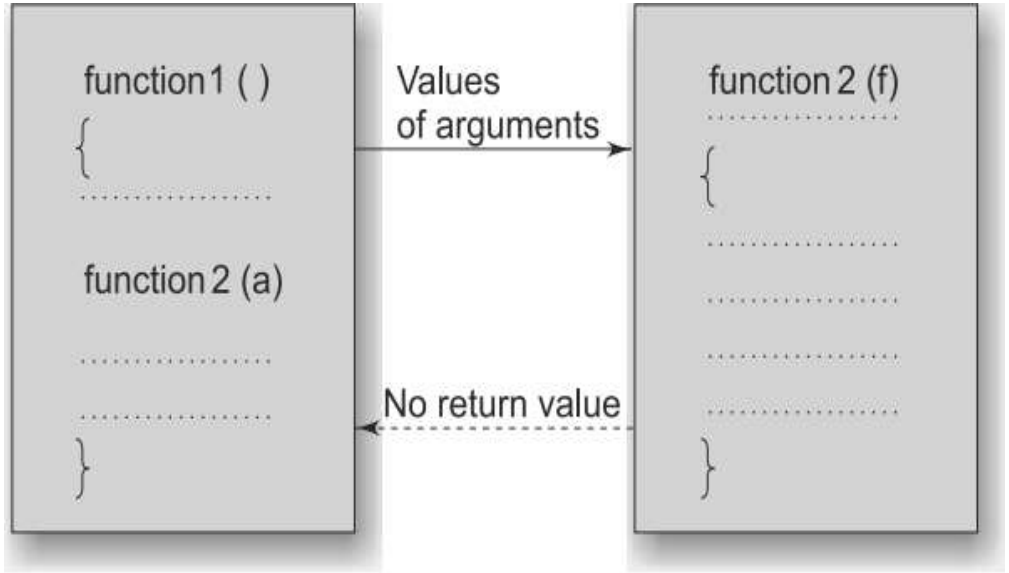


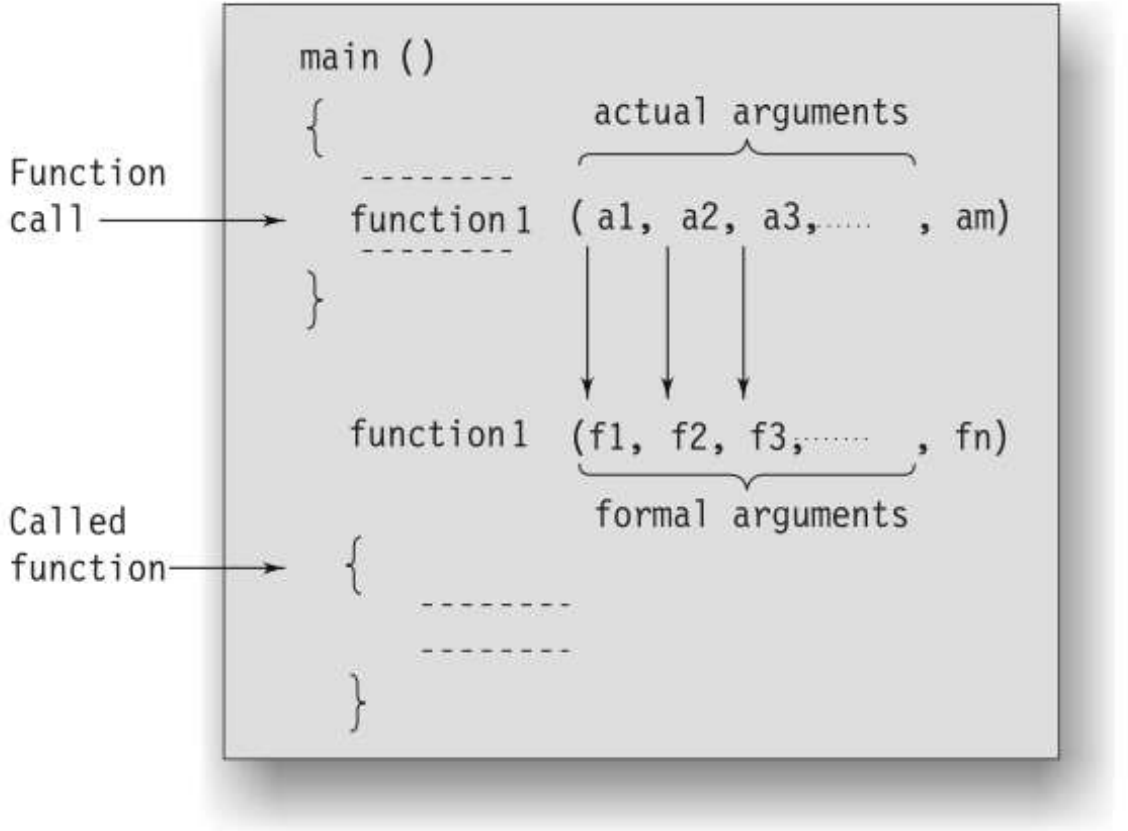
Fig. 11.3 *No data communication between functions*

Arguments But No Return Values

- The actual and formal arguments should match in number, type, and order.
- The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument as shown in Fig

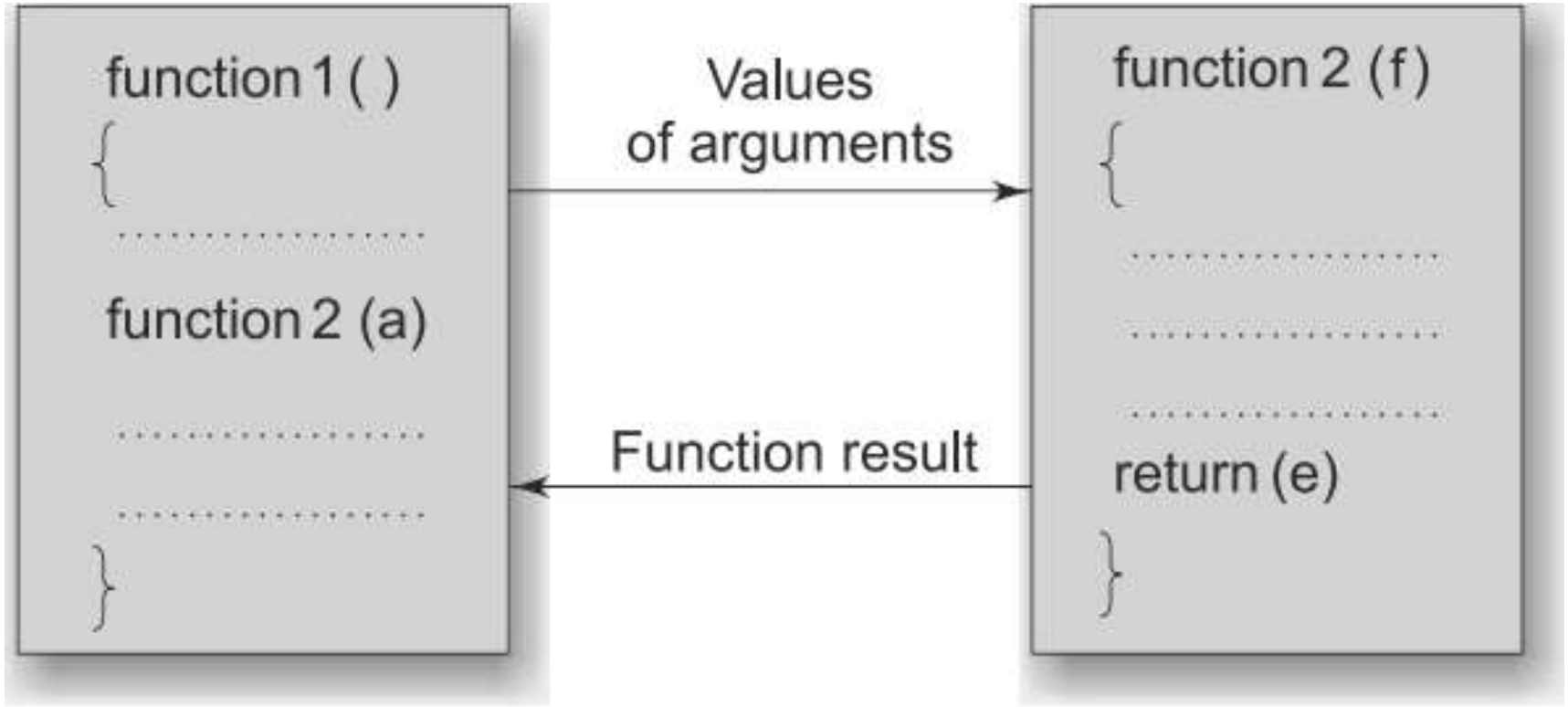


One-way data communication



Arguments matching between the function call and the called function

Arguments with Return Values



Two-way data communication between functions

No Arguments But Returns a Value

1. There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function.
2. A typical example is the **getchar** function declared in the header file <stdio.h>.
3. We have used this function earlier in a number of places.
4. The getchar function has no parameters but it returns an integer type data that represents a character.

- We can design similar functions and use n our programs.
- Example

```
int get_number(void); main
        {
        int m = get_number( ); printf("%d",m);
        }
 int get_number(void)
        {
         int number;  scanf("%d", &number); return(number);
        }
```

NESTING OF FUNCTIONS

- C permits nesting of functions freely.
- main can call function1, which calls function2, which calls function3, ……. and so on.
- There is in principle no limit as to how deeply functions can be nested.

```
float ratio (int x, int y, int z);
int difference (int x, int y);
main( )
{
   int a, b, c;
   scanf("%d %d %d", &a, &b, &c);
   printf("%f \n", ratio(a,b,c));
}

float ratio(int x, int y, int z)
{
   if(difference(y, z))
      return(x/(y-z));
   else
      return(0.0);
}
int difference(int p, int q)
{
      if(p != q)
         return (1);
      else
         return(0);
}
```


Thank You!