SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF AIML

PROGRAMMING FOR PROBLEM SOLVING

I YEAR - I SEM

UNIT 4 – FUNCTIONS AND POINTERS

TOPIC 2 – Function Calls

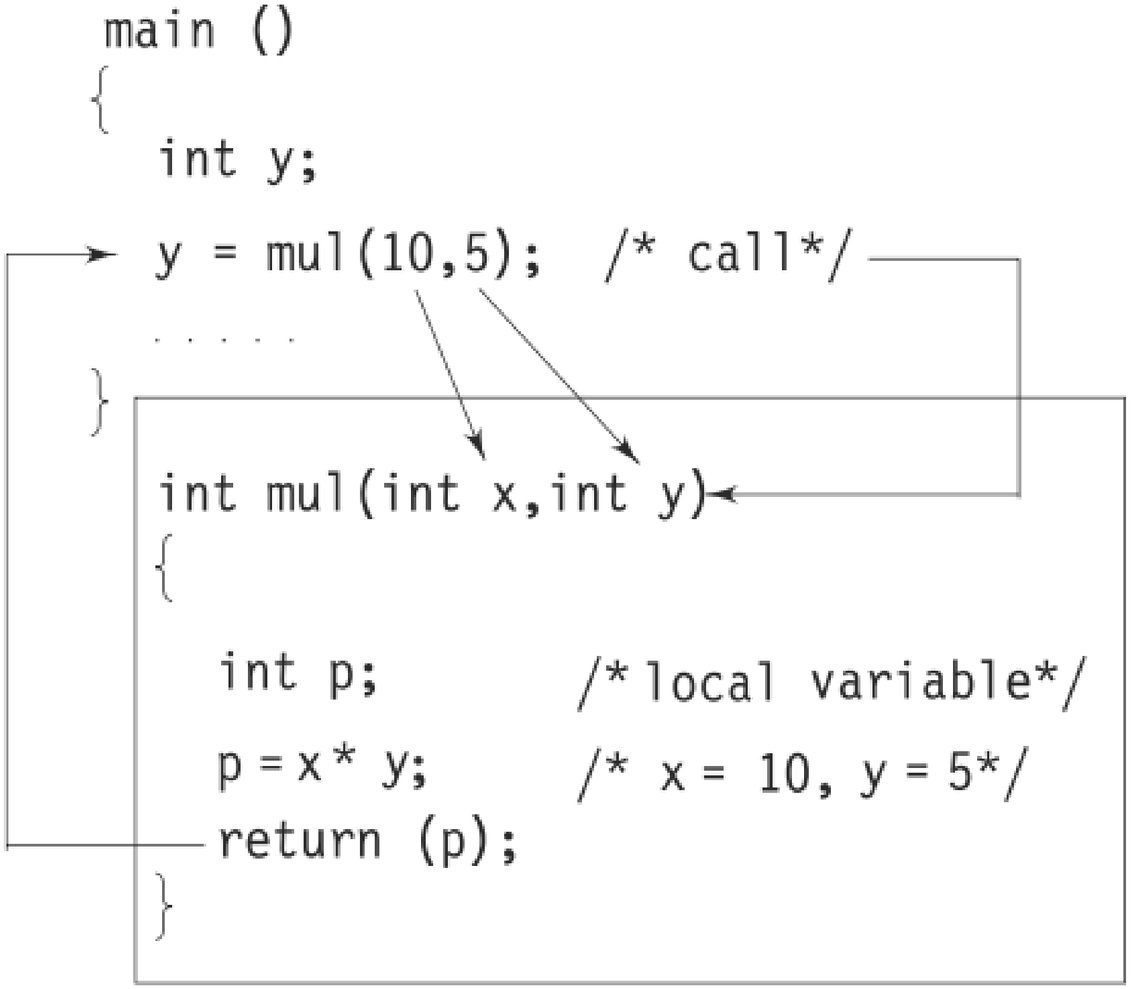
FUNCTION CALLS

- A function can be called by simply using the function name followed by a list of actual parameters (or arguments), if any, enclosed in parentheses.
- Example:

```
main( )
{
    int y;
    y = mul(10,5);     /* Function call */
    printf("%d\n", y);
}
```

- When the compiler encounters a **function call**, the control is transferred to the function mul().
- This function is then executed line by line as described and a value is returned when a return statement is encountered.
- This value is assigned to y.
- This is illustrated below and shown as figure.

FUNCTION CALLS



FUNCTION CALLS

- The function call sends two integer values 10 and 5 to the function.
  - int mul(int x, int y)
- which are assigned to x and y respectively.
- The function computes the product x and y, assigns the result to the local variable p, and then returns the value 25 to the main where it is assigned to y again.
- There are many different ways to call a function.
- Listed below are some of the ways the function **mul** can be invoked.

```
mul (10, 5)
mul (m, 5)
mul (10, n)
mul (m, n)
mul (m + 5, 10)
mul (10, mul(m,n))
mul (expression1, expression2)
```

- Note that the **sixth call** uses its own call as its one of the parameters.
- When we use expressions, they should be evaluated to single values that can be passed as actual parameters.

FUNCTION CALLS

- A function which returns a value can be used in expressions like any other variable.
- Each of the following statements is valid:

```
printf("%d\n", mul(p,q));
y = mul(p,q) / (p+q);
if (mul(m,n)>total) printf("large");
```

- However, a function cannot be used on the right side of an assignment statement.
- For instance, **mul(a,b) = 15;** is invalid.
- A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function.
- The function printline( ) discussed belongs to this category.
- Such functions may be called in by simply stating their names as independent statements.
- Example:

```
main( )
{
printline( );
}
```

- Note the presence of a semicolon at the end.

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.**

FUNCTION DECLARATION

- **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:

- Category 1: Functions with **no** arguments and **no** return values.
- Category 2: Functions with arguments and **no** return values.
- Category 3: Functions with arguments and **one** return value.
- Category 4: Functions with **no** arguments but return a value.
- 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 only a transfer of control but not data.

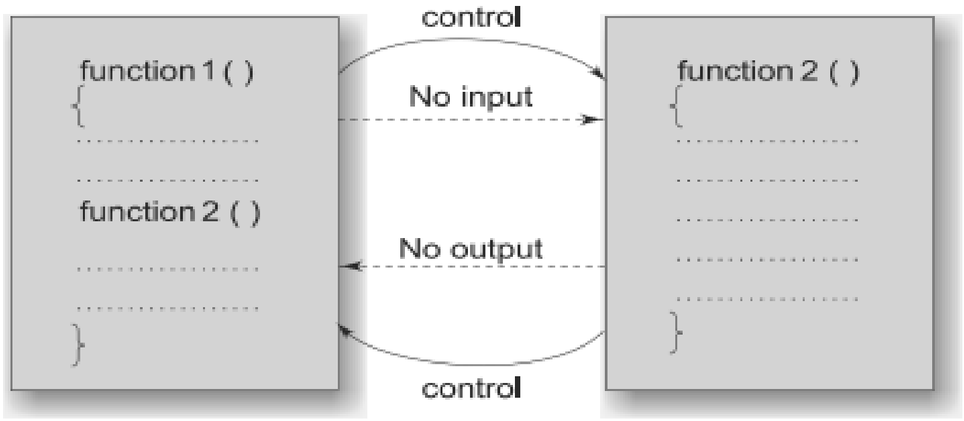
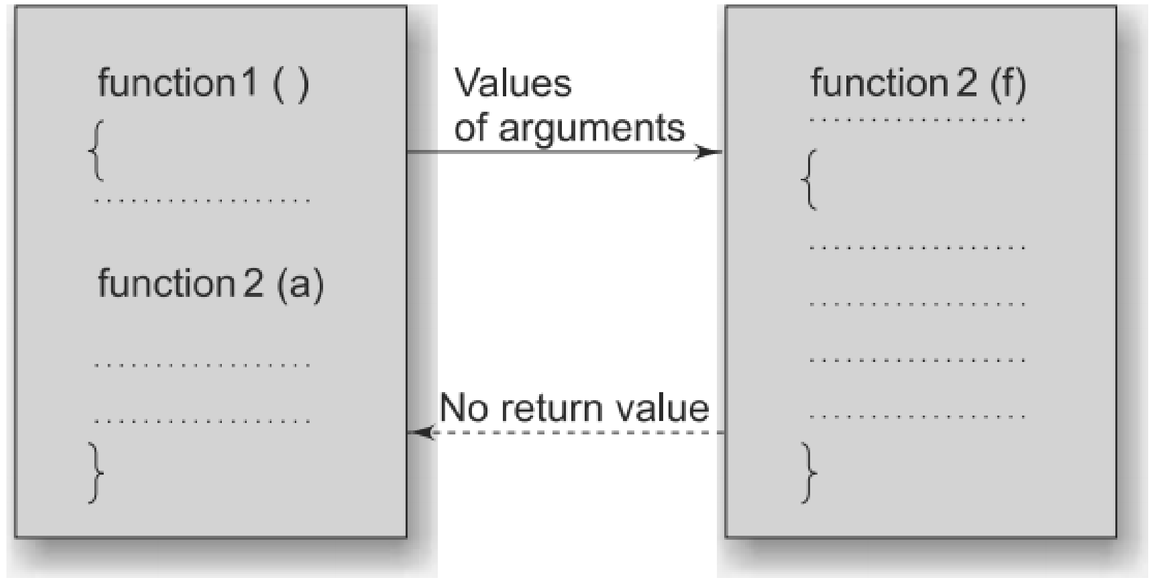


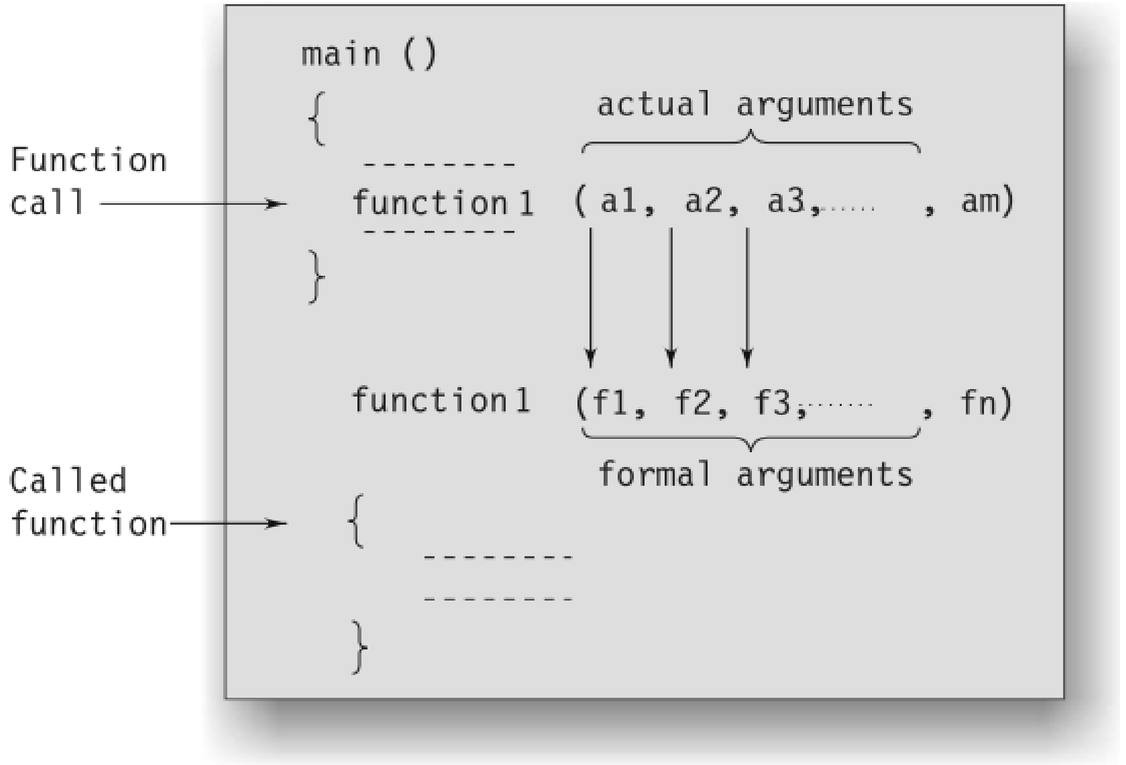
Fig. 11.3 *No data communication between functions*

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

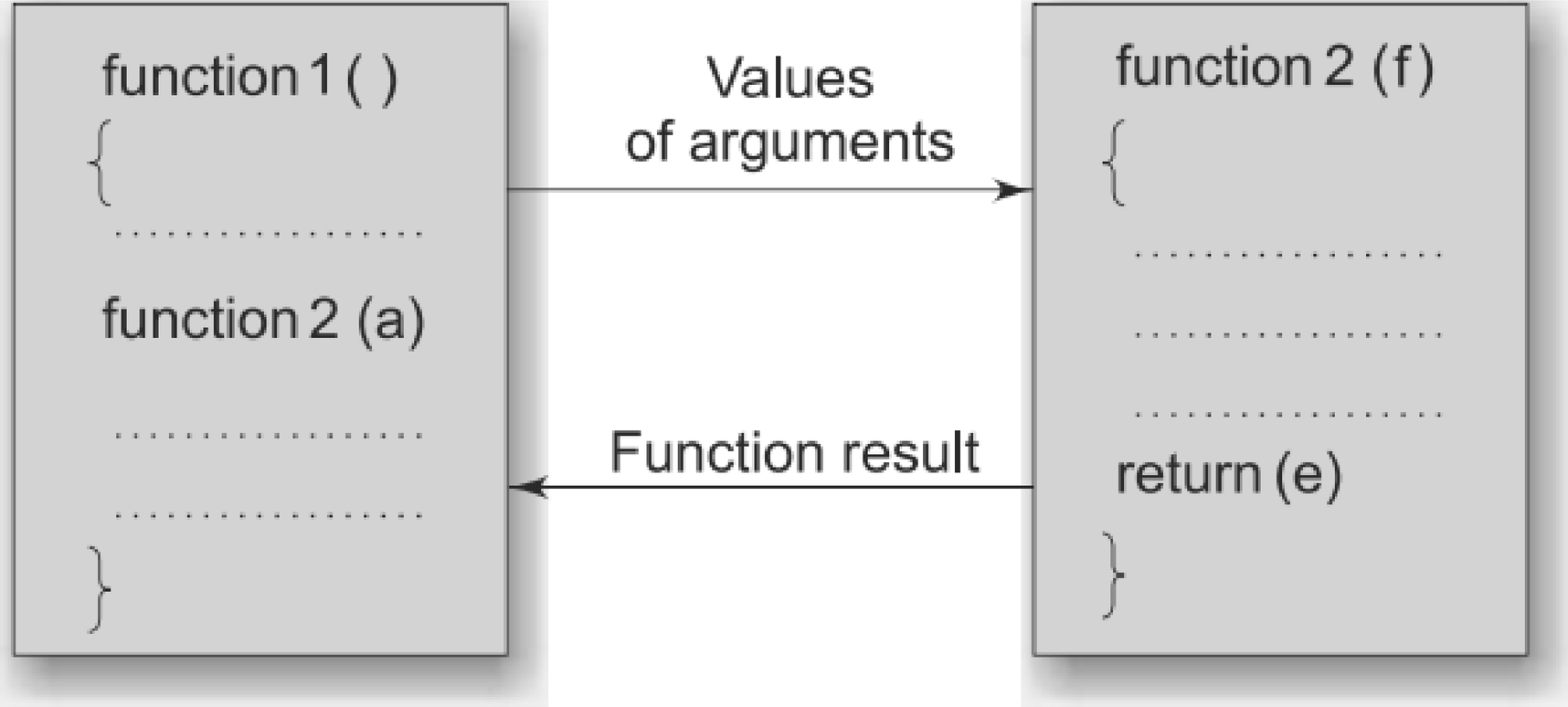


One-way data communication



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

- 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.
- A typical example is the **getchar** function declared in the header file <stdio.h>.
- We have used this function earlier in a number of places.
- The getchar function has no parameters but it returns an integer type data that represents a character.
- We can design similar functions and use in 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);
}
```

RECURSION

- When a called function in turn calls another function a process of ‘chaining’ occurs.
- Recursion is a special case of this process, where a function calls itself.
- A very simple example of recursion is presented below:

```
main( )
{
printf(“This is an example of recursion\n”)
main( );
}
```

- When executed, this program will produce an output something like this:

```
This is an example of recursion
This is an example of recursion
This is an example of recursion
This is an ex
```

- Execution is terminated abruptly; otherwise the execution will continue indefinitely.

RECURSION

- Another useful example of recursion is the evaluation of factorials of a given number.
- The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

  factorial of n = n(n–1)(n–2).....1.
- For example,

  factorial of 4 = 4 3 2 1 = 24
- A function to evaluate factorial of n is as follows:

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

RECURSION

- Let us see how the recursion works.
- Assume n = 3.
- Since the value of n is not 1, the statement
  - fact = n * factorial(n–1);
- will be executed with n = 3.
- That is, fact = 3 * factorial(2); will be evaluated.
- The expression on the right-hand side includes a call to factorial with n = 2.
- This call will return the following value:
  - 2 * factorial(1)
- Once again, factorial is called with n = 1.
- This time, the function returns 1.
- The sequence of operations can be summarized as follows:
- fact = 3 * factorial(2)
- = 3 * 2 * factorial(1)
- = 3 * 2 * 1
- = 6

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

PASS BY VALUE VERSUS PASS BY POINTERS

- The technique used to pass data from one function to another is known as parameter passing.
- Parameter passing can be done in following **two** ways:
  - Pass by value (also known as call by value).
  - Pass by pointers (also known as call by pointers).
- In pass by value, **values of actual parameters are copied** to the variables in the parameter list of the called function.
- The called function **works on the copy and not on the original** values of the actual parameters.
- This ensures that the original data in the calling function cannot be changed accidentally.
- In pass by pointers (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function.
- In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.
- Pass by pointers method is often used when manipulating arrays and strings.
- This method is also used when we require multiple values to be returned by the called function.

THE SCOPE, VISIBILITY, AND LIFETIME OF VARIABLES

- Variables in C differ in behaviour from those in most other languages.
- For example, in a BASIC program, a variable retains its value throughout the program.
- It is not always the case in C.
- It all depends on the ‘storage’ class a variable may assume.
- In C not only do all variables have a data type, they also have a **storage class**.
- The following variable storage classes are most relevant to functions:
  - **1. Automatic variables.**
  - **2. External variables.**
  - **3. Static variables.**
  - **4. Register variables.**
- We shall briefly discuss the scope, visibility, and longevity of each of the above class of variables.

THE SCOPE, VISIBILITY, AND LIFETIME OF VARIABLES

- **Scope**
- The scope of variable determines over what region of the program a variable is actually available for use(‘active’).
- **Longevity**
- Longevity refers to the period during which a variable retains a given value during execution of a program (‘alive’).
- So longevity has a direct effect on the utility of a given variable.
- **Visibility**
- The visibility refers to the accessibility of a variable from the memory.
- The variables may also be broadly categorized, depending on the place of their declaration, as **internal (local) or external (global).**
- Internal variables are those which are declared within a particular function,
- while external variables are declared outside of any function.