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

23ITT101-PROGRAMMING IN C AND DATA STRUCTURES

I YEAR - II SEM

UNIT 2 – DECISIONS STATEMENTS AND FUNCTIONS

TOPIC 3 – Functions

INTRODUCTION

- The **strengths** of C language is C functions.
- They **are easy to define and use**.
- We have used functions in every program that we have discussed so far.
- However, they have been primarily limited to the **three** functions, namely
  - main, printf, and scanf.
- C functions can be classified into **two categories**, namely, **library functions and user-defined functions**.
- **main** is an example of user-defined functions.
- printf and scanf belong to the category of library functions.
- The main distinction between these two categories is that library functions are not required to be written by us.
- Whereas a user-defined function has to be developed by the user at the time of writing a program.
- However, a user-defined function can later become a part of the C program library.
- In fact, this is one of the strengths of C language.

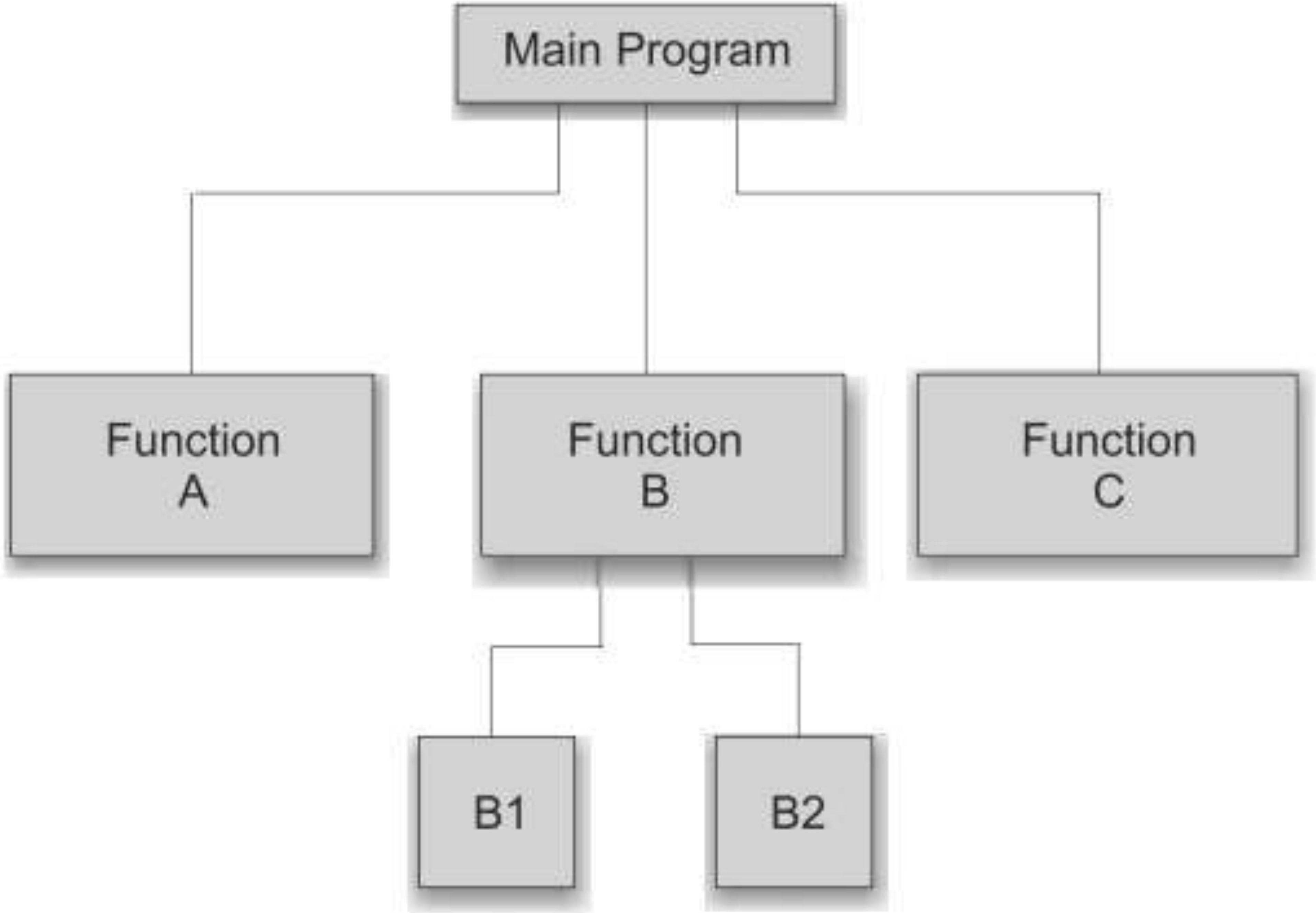
NEED FOR USER-DEFINED FUNCTIONS

- **Every program must have a main function** to indicate where the program has to begin its execution.
- While it is possible to code any program utilizing only main function, it leads to a number of problems.
- The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult.
- If a program is divided into **functional parts**, then each part may be independently coded and later combined into a single unit.
- These independently coded programs are called **subprograms** that are much easier to understand, debug, and test.
- In C, such subprograms are referred to as **‘functions’.**

NEED FOR USER-DEFINED FUNCTIONS

- There are times when certain type of operations or calculations are **repeated** at many points throughout a program.
- For instance, we might use the factorial of a number at several points in the program.
- In such situations, we may **repeat the program statements** wherever they are needed.
- Another approach is to design a function that can be called and used whenever required.
- This saves both time and space.

MODULAR DIVISION



Top-down modular programming using functions

NEED FOR USER-DEFINED FUNCTIONS

- This “division” approach clearly results in a number of advantages.

- 1. It **facilitates top-down modular** programming as shown in Fig.
- In this programming style, the **high level logic** of the overall problem is **solved first** while the details of each **lower-level** function are addressed later.

- 2. The **length of a source program can be reduced** by using functions at appropriate places.

- 3. It is **easy to locate and isolate** a faulty function for further investigations.

- 4. **A function may be used by many other programs**. This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

A MULTI-FUNCTION PROGRAM

- A function is a self-contained block of code that performs a particular task.

- Once a function has been designed and packed, it can be treated as a ‘black box’ that takes some data from the main program and returns a value.

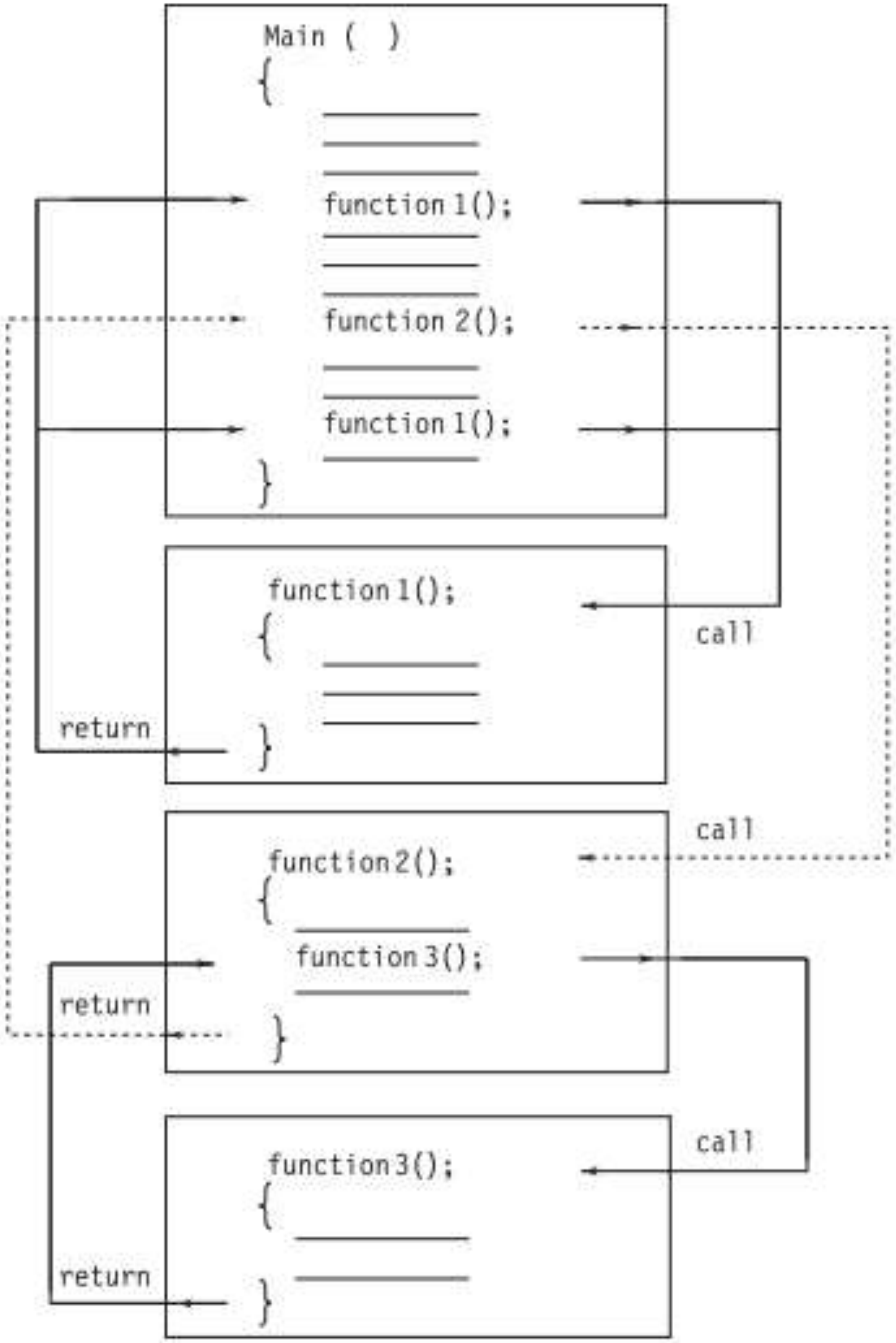
- The inner details of operation are **invisible** to the rest of the program.

- All that the program knows about a function is: What goes in and what comes out.

- Every C program can be designed using a collection of these black boxes known as **functions**.

FUNCTIONS - Example

```
int main(){
  int a,b;
   printf("enter two values");
 scanf("%d %d",&a,&b);
   add(a,b); // calling function
  sub(a,b);
}
add(int x,int y)//called function
{    int c;
  c=x+y;
   printf("%d",c);
    }
sub(int x,int y)
{    int d;
  d=x-y;
   printf("\n%d",d);
}
```



Flow of control in a multi-function program

MODULAR PROGRAMMING

- Any function can call any other function.
- In fact, it can call itself.
- A ‘**called function**’ can also call another function.
- A function can be called more than once.
- In fact, this is one of the main features of using functions.
- Figure illustrates the flow of control in a multi-function program.
- Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the functions that make up a complete program.
- The functions can be placed in any order.
- A **called function** can be placed either before or after the calling function.
- However, it is the usual practice to put all the called functions at the end.

MODULAR PROGRAMMING

- Modular programming is a strategy applied to the design and development of software systems.
- It is defined as organizing a large program into small, independent program segments called **modules** that are separately named and individually callable program units.
- These modules are carefully integrated to become a software system that satisfies the system requirements.
- It is basically a **“divide-and-conquer”** approach to problem solving.
- Modules are identified and designed such that they can be organized into a **top-down** hierarchical structure (similar to an organization chart).
- **In C, each module refers to a function that is responsible for a single task.**

CHARACTERISTICS OF MODULAR PROGRAMMING

- 1. Each module should do only one thing.
- 2. Communication between modules is allowed only by a **calling module**.
- 3. A module can be called by one and only one higher module.
- 4. No communication can take place directly between modules that do not have calling – called relationship.
- 5. All modules are designed as single-entry, single-exit systems using control structures.

ELEMENTS OF USER-DEFINED FUNCTIONS

- We have discussed and used a variety of data types and variables in our programs so far.
- However, declaration and use of these variables were primarily done inside the main function.
- As mentioned, functions are classified as one of the **derived data types in C**.
- We can therefore define functions and use them like any other variables in C programs.
- It is therefore not a surprise to note that there exist some similarities between functions and variables in C. They are
- Both function names and variable names are considered identifiers and therefore, they must adhere to the rules for identifiers.
- Like variables, functions have types (such as int) associated with them.
- Like variables, function names and their types must be declared and defined before they are used in a program

ELEMENTS OF USER-DEFINED FUNCTIONS

- In order to make use of a user-defined function, we need to establish three elements that are related to functions.
- **1. Function definition.**
- **2. Function call.**
- **3. Function declaration.**
- The **function definition** is an independent program module that is specially written to implement the requirements of the function.
- In order 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 (or a function) that calls the function is referred to as the **calling program or calling function**.
- The calling program should declare any function (like declaration of a variable) that is to be used later in the program.
- This is known as the **function declaration or function prototype**.

DEFINITION OF FUNCTIONS

- A **function definition**, also known as **function implementation** shall include the following elements:
    - 1. function name;
    - 2. function type;
    - 3. list of parameters;
    - 4. local variable declarations;
    - 5. function statements; and
    - 6. a return statement.
- All the six elements are grouped into **two parts**, namely,
    - function header (First three elements); and
    - function body (Second three elements).

DEFINITION OF FUNCTIONS

- A general format of a function definition to implement these two parts is given below:

```
function_type function_name(parameter list)
{
          local variable declaration;
         executable statement1;
         executable statement2;
          . . . . .
          . . . . .
         return statement;
}
```

- The first line function_type function_name(parameter list) is known as the **function header** and the statements within the opening and closing braces constitute the **function body**, which is a compound statement.

DEFINITION OF FUNCTIONS

- **Function Header**
- The function header consists of **three** parts:
  - the function type (also known as return type)
  - the function name
  - the formal parameter list.
- Note that a semicolon is not used at the end of the function header.
- **Name and Type**
- The **function type** specifies the type of value (like float or double) that the function is expected to return to the program **calling the function**.
- If the return type is not explicitly specified, C will assume that it is an integer type.
- If the function is not returning anything, then we need to specify the return type as void.
- The value returned is the output produced by the function.
- The **function name** is any valid C identifier and therefore must follow the same rules of formation as other variable names in C.
- The name should be **appropriate** to the task performed by the function.

DEFINITION OF FUNCTIONS

- **Formal Parameter List**
- The parameter list declares the variables that will receive the data sent by the calling program.
- They serve as input data to the function to carry out the specified task.
- Since they represent the actual input values, they are often referred to as **formal parameters**.
- These parameters can also be used to **send values to the calling programs**.
- The parameters are also known as **arguments**.
- The parameter list contains declaration of variables separated by commas and surrounded by parentheses.
- Examples:
  - float quadratic (int a, int b, int c) {. . . . }
  - double power (double x, int n) {. . . ..}
  - float mul (float x, float y) {. . . . }
  - int sum (int a, int b) {. . . . }
- Remember, there is no semicolon after the closing parenthesis.

DEFINITION OF FUNCTIONS

- Note that the declaration of parameter variables cannot be combined.
- That is, int sum (int a,b) **is illegal**.
- A function need not always receive values from the calling program.
- In such cases, functions have no formal parameters.
- To indicate that the parameter list is empty, we use the keyword void between the parentheses as in **void printline (void)**

```
void printline (void)
{
 . . . .
 }
```

- This function neither receives any input values nor returns back any value.
- Many compilers accept an empty set of parentheses, without specifying anything

DEFINITION OF FUNCTIONS

- **Function Body**
- The function body contains the **declarations** and statements necessary for performing the required task.
- The body enclosed in braces, contains **three parts**, in the order given below:
  - 1. Local declarations that specify the variables needed by the function.
  - 2. Function statements that perform the task of the function.
  - 3. A return statement that returns the value evaluated by the function.
- If a function does not return any value (like the printline function), we can omit the return statement.
- However, note that its return type should be specified as void.
- Again, it is nice to have a return statement even for void functions.
- Some examples of typical function definitions are:

FUNCTION DEFINITION - Example

```
(a)    float mul (float x, float y)//function header
       {
        float result;          /* local variable */
       result = x * y;     /* computes the product */
       return (result);    /* returns the result */
       }
(b)    void sum (int a, int b)
       {
       printf (“sum = %d”, a + b);          /* no local variables */
       return;                              /* optional */
       }
 (c)   void display ( )
        {
       /* no local variables */
       printf (“No type, no parameters”);
       /* no return statement */
       }
```

RETURN VALUES AND THEIR TYPES

- As pointed out earlier, a function may or may not send back any value to the calling function.
- If it does, it is done through the **return** statement.
- While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most.
- The return statement can take one of the following forms:

```
return;
 or
return(expression);
```

- The first, the ‘plain’ return does not return any value; it acts much as the closing brace of the function.
- When a return is encountered, the control is immediately passed back to the **calling function**.
- An example of the use of a simple return is as follows:

```
 if(error)
return;
```

RETURN VALUES AND THEIR TYPES

- The second form of return with an expression returns the value of the expression.
- For example, the function

```
int mul (int x, int y)
{
    int p;
    p = x*y;
     return(p);
}
```

- returns the value of p which is the product of the values of x and y.
- The last two statements can be combined into one statement as follows:
    - return (x*y);
- A function may have more than one return statements

RETURN VALUES AND THEIR TYPES

- The above situation arises when the value returned is based on certain conditions.
- For example:

```
 if( x <= 0 )
return(0);
 else
return(1);
```

- What type of data does a function return? All functions by default return int type data.
- But what happens if a function must return some other type? We can force a function to return a particular type of data by using a type specifier in the function header as discussed earlier.
- When a value is returned, it is automatically cast to the function’s type.
- In functions that do computations using doubles, yet return ints, the returned value will be truncated to an integer.
- For instance, the function will return the value 7, only the integer part of the result.

```
 int product (void)
{
return (2.5 * 3.0);
}
```