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.

Some characteristics of modular programming are:

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.

9.4 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 we mentioned in Chapter 4, 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.

- 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.

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.*

9.5 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
6. A return statement

All the six elements are grouped into two parts, namely,

- Function header (First three elements)
- Function body (Second three elements)

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.

Function Header

The function header consists of three parts: The function type (also known as *return* type, the function name and 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**. Remember, **void** is one of the fundamental data types in C. It is a good programming practice to code explicitly the return type, even when it is an integer. (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. However, care must be exercised to avoid duplicating library routine names or operating system commands.

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 actual input values, they are often referred to as *formal* parameters. These parameters can also be used to send values to the calling programs. This aspect will be covered later when we discuss more about functions. 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. 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)
{
    . . . .
}
```

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

```
void printline ( )
```

But, it is a good programming style to use **void** to indicate a nill parameter list.

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;

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

Note:

1. When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a *void return.*
2. A *local variable* is a variable that is defined inside a function and used without having any role in the communication between functions.

9.6 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;
```

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. This 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

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

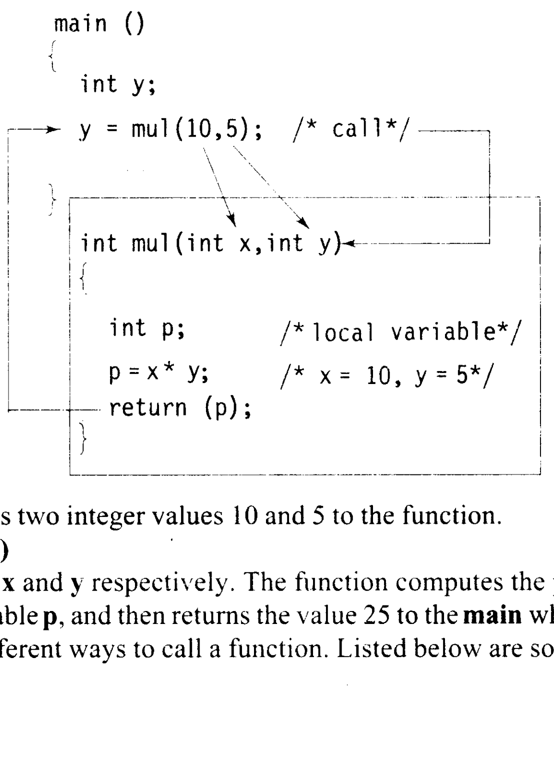
will return the value 7, only the integer part of the result.

9.7 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:



```
main ()
{
   int y;
   y = mul(10,5);  /* call*/
}

int mul(int x,int y)
{
   int p;          /*local variable*/
   p = x* y;       /* x = 10, y = 5*/
   return (p);
}
```

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.

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 in Section 9.3 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 Call

A function call is a postfix expression. The operator (. .) is at a very high level of precedence. (See Table 3.8) Therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parentheses set (. .) which contains the *actual parameters* is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note:

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

9.8 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 */

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.

Equally acceptable forms of declaration of **mul** function are:

```
int mul  (int, int);
    mul  (int a, int b);
    mul  (int, int);
```

When a function does not take any parameters and does not return any value, its prototype is written as:

```
void display (void);
```

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. (Scope is discussed later in this chapter.) 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.

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 three places;

1. In declaration (prototypes)
2. In function call
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.

9.9 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.

In the sections to follow, we shall discuss these categories with examples. Note that, from now on, we shall use the term arguments (rather than parameters) more frequently.

9.10 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. 9.3. The dotted lines indicate that there is only a transfer of control but not data.

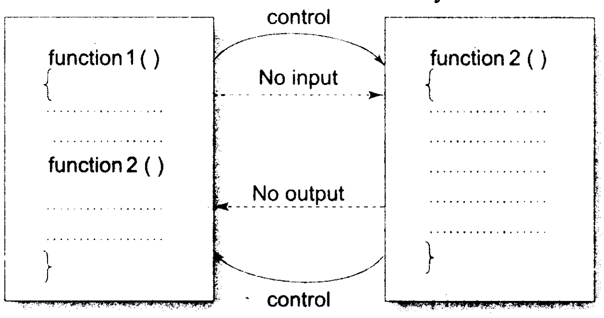


Fig. 9.3 *No data communication between functions*

As pointed out earlier, a function that does not return any value cannot be used in an expression. It can only be used as an independent statement.

Example 9.1 Write a program with multiple functions that do not communicate any data between them.

A program with three user-defined functions is given in Fig. 9.4. **main** is the calling function that calls **printline** and **value** functions. Since both the called functions contain no arguments, there are no argument declarations. The **printline** function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The **value** function calculates the value of principal amount after a certain period of years and prints the results. The following equation is evaluated repeatedly:

value = principal(1+interest-rate)

Program

```
/* Function declaration */
void printline (void);
void value (void);

main()
{
    printline();
    value();
    printline();
}
/*        Function1: printline( )        */

void printline(void)    /* contains no arguments */
{
    int i ;

    for(i=1; i <= 35; i++)
       printf("%c",'-');
    printf("\n");
}
/*        Function2: value( )            */
void value(void)        /* contains no arguments */
{
    int    year, period;
    float  inrate, sum, principal;

    printf("Principal amount?");
    scanf("%f", &principal);
    printf("Interest rate?   ");
    scanf("%f", &inrate);
    printf("Period?          ");
```

```
        scanf("%d", &period);

        sum = principal;
        year = 1;
        while(year <= period)
        {
           sum = sum *(1+inrate);
           year = year +1;
        }
        printf("\n%8.2f %5.2f %5d %12.2f\n",
                principal,inrate,period,sum);
    }
```

Output

```
- - - - - - - - - - - - - - - - - - - - - - - -
Principal amount?   5000
Interest rate?      0.12
Period?             5

5000.00  0.12       5      8811.71
- - - - - - - - - - - - - - - - - - - - - - - -
```

Fig. 9.4 *Functions with no arguments and no return values*

It is important to note that the function **value** receives its data directly from the terminal. The input data include principal amount, interest rate and the period for which the final value is to be calculated. The **while** loop calculates the final value and the results are printed by the library function **printf.** When the closing brace of **value( )** is reached, the control is transferred back to the calling function **main.** Since everything is done by the value itself there is in fact nothing left to be sent back to the called function. Return types of both **printline** and **value** are declared as **void.**

Note that no **return** statement is employed. When there is nothing to be returned, the **return** statement is optional. The closing brace of the function signals the end of execution of the function, thus returning the control, back to the calling function.

9.11 ARGUMENTS BUT NO RETURN VALUES

In Fig. 9.4 the **main** function has no control over the way the functions receive input data. For example, the function **printline** will print the same line each time it is called. Same is the case with the function **value.** We could make the calling function to read data from the terminal and pass it on to the called function. This approach seems to be wiser because the calling function can check for the validity of data, if necessary, before it is handed over to the called function.

The nature of data communication between the calling function and the called function with arguments but no return value is shown in Fig. 9.5.

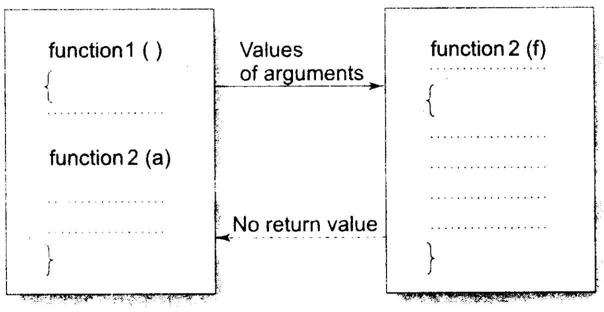


Fig. 9.5 *One-way data communication*

We shall modify the definitions of both the called functions to include arguments as follows:

void printline(char ch)
void value(float p, float r, int n)

The arguments **ch, p, r,** and **n** are called the *formal arguments*. The calling function can now send values to these arguments using function calls containing appropriate arguments. For example, the function call

value(500,0.12,5)

would send the values 500,0.12 and 5 to the function

void value(float p, float r, int n)

and assign 500 to **p**, 0.12 to **r** and 5 to **n**. The values 500, 0.12 and 5 are the *actual arguments, which* become the values of the *formal arguments* inside the called function.

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. 9.6.

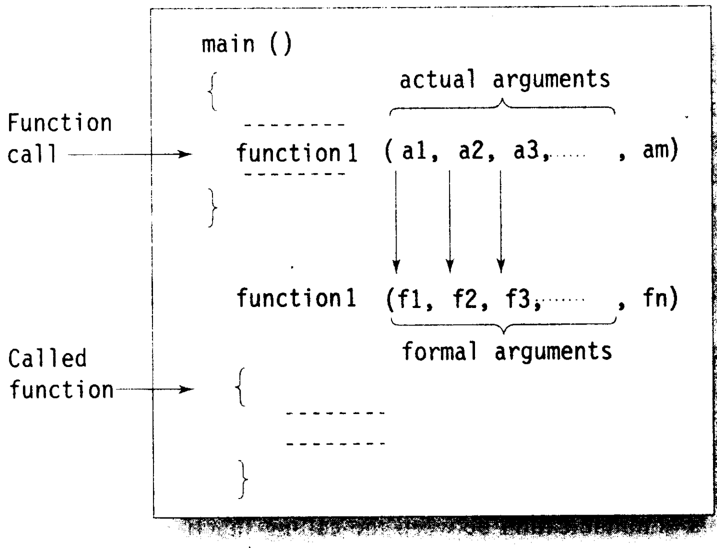


Fig. 9.6 *Arguments matching between the function call and the called function*

We should ensure that the function call has matching arguments. In case, the actual arguments are more than the formal arguments (m > n), the extra actual arguments are discarded. On the other hand, if the actual arguments are less than the formal arguments, the unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values. Remember, no error message will be generated.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions, or constants. The variables used in actual arguments must be assigned values before the function call is made.

Remember that, when a function call is made, only *a copy of the values of actual arguments is passed into the called function*. What occurs inside the function will have no effect on the variables used in the actual argument list.

Example 9.2 Modify the program of Example 9.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig. 9.7. Most of the program is identical to the program in Fig. 9.4. The input prompt and **scanf** assignment statement have been moved from **value** function to **main.** The variables **principal, inrate,** and **period** are declared in **main** because they are used in main to receive data. The function call

```
value(principal, inrate, period);
```

passes information it contains to the function **value.**

The function header of **value** has three formal arguments **p,r,** and **n** which correspond to the actual arguments in the function call, namely, **principal, inrate,** and **period.** On execution of the function call, the values of the actual arguments are assigned to the corresponding formal arguments. In fact, the following assignments are accomplished across the function boundaries:

```
p = principal;
r = inrate;
n = period;
```

Program

```
/* prototypes */
void printline (char c);
void value (float, float, int);

main( )
{
     float principal, inrate;
     int period;

     printf("Enter principal amount, interest");
     printf(" rate, and period \n");
     scanf("%f %f %d",&principal, &inrate, &period);
     printline('Z');
     value(principal,inrate,period);
```

```
        println('C');
    }
    void printline(char ch)
    {
        int i ;
        for(i=1; i <= 52; i++)
            printf("%c",ch);
        printf("\n");
    }

    void value(float p, float r, int n)
    {
        int year ;
        float sum ;
        sum = p ;
        year = 1;
        while(year <= n)
        {
            sum = sum * (1+r);
            year = year +1;
        }
        printf("%f\t%f\t%d\t%f\n",p,r,n,sum);
    }
```

Output

```
Enter principal amount, interest rate, and period
5000 0.12  5
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
5000.000000      0.120000        5       8811.708984
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Fig. 9.7 *Functions with arguments but no return values*

The variables declared inside a function are known as *local variables* and therefore their values are local to the function and cannot be accessed by any other function. We shall discuss more about this later in the chapter.

The function **value** calculates the final amount for a given period and prints the results as before. Control is transferred back on reaching the closing brace of the function. Note that the function does not return any value.

The function **printline** is called twice. The first call passes the character 'Z', while the second passes the character 'C' to the function. These are assigned to the formal argument **ch** for printing lines (see the output).

Variable Number of Arguments

Some functions have a variable number of arguments and data types which cannot be known at compile time. The **printf** and **scanf** functions are typical examples. The ANSI standard proposes new symbol called the *ellipsis* to handle such functions. The *ellipsis* consists of three periods (...) and used as shown below:

double area(float d,...)

Both the function declaration and definition should use ellipsis to indicate that the arguments are arbitrary both in number and type.

9.12 ARGUMENTS WITH RETURN VALUES

The function **value** in Fig. 9.7 receives data from the calling function through arguments, but does not send back any value. Rather, it displays the results of calculations at the terminal. However, we may not always wish to have the result of a function displayed. We may use it in the calling function for further processing. Moreover, to assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in Fig. 9.8

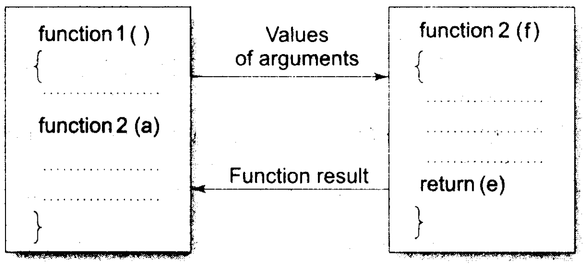


Fig. 9.8 *Two-way data communication between functions*

We shall modify the program in Fig. 9.7 to illustrate the use of two-way data communication between the calling and the called functions.

Example 9.3 In the program presented in Fig. 9.7 modify the function **value,** to return the final amount calculated to the **main,** which will display the required output at the terminal. Also extend the versatility of the function **printline** by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig. 9.9. One major change is the movement of the **printf** statement from **value** to **main.**

Program

```
    void printline (char ch, int len);
    value (float, float, int);

      main( )
      {
        float principal, inrate, amount;
        int period;
        printf("Enter principal amount, interest");
        printf("rate, and period\n");
        scanf(%f %f %d", &principal, &inrate, &period);
        printline ('*' , 52);
        amount = value (principal, inrate, period);
        printf("\n%f\t%f\t%d\t%f\n\n",principal,
          inrate,period,amount);
        printline('=',52);
      }
      void printline(char ch, int len)
      {
        int i;
        for (i=1;i<=len;i++) printf("%c",ch);
        printf("\n");
      }
      value(float p, float r, int n) /* default return type */
      {
        int year;
        float sum;
        sum = p; year = 1;
        while(year <=n)
        {
          sum = sum * (1+r);
          year = year +1;
        }
        return(sum);      /* returns int part of sum */
      }
```

Output

```
Enter principal amount, interest rate, and period
5000   0.12    5
**************************************************

5000.000000   0.1200000   5   8811.000000

= = = = = = = = = = = = = = = = = = = = = = = = = =
```

Fig. 9.9 *Functions with arguments and return values*

The calculated value is passed on to **main** through statement:

```
return(sum);
```

Since, by default, the return type of **value** function is **int**, the 'integer' value of **sum** at this point is returned to **main** and assigned to the variable **amount** by the functional call

```
amount = value (principal, inrate, period);
```

The following events occur, in order, when the above function call is executed:

1. The function call transfers the control along with copies of the values of the actual arguments to the function **value** where the formal arguments **p**, **r**, and **n** are assigned the actual values of **principal, inrate** and **period** respectively.
2. The called function **value** is executed line by line in a normal fashion until the **return(sum);** statement is encountered. At this point, the integer value of **sum** is passed back to the function-call in the **main** and the following indirect assignment occurs:

   ```
   value(principal, inrate, period) = sum;
   ```
3. The calling statement is executed normally and the returned value is thus assigned to **amount**, a **float** variable.
4. Since **amount** is a **float** variable, the returned integer part of sum is converted to floating-point value. See the output.

Another important change is the inclusion of second argument to **printline** function to receive the value of length of the line from the calling function. Thus, the function call

```
printline('*', 52);
```

will transfer the control to the function **printline** and assign the following values to the formal arguments **ch**, and **len**;

```
ch = '*' ;
len = 52;
```

Returning Float Values

We mentioned earlier that a C function returns a value of the type **int** as the default case when no other type is specified explicitly. For example, the function **value** of Example 9.3 does all calculations using **floats** but the return statement

```
return(sum);
```

returns only the integer part of **sum.** This is due to the absence of the *type-specifier* in the function header. In this case, we can accept the integer value of **sum** because the truncated decimal part is insignificant compared to the integer part. However, there will be times when we may find it necessary to receive the **float** or **double** type of data. For example, a function that calculates the mean or standard deviation of a set of values should return the function value in either **float** or **double.**

In all such cases, we must explicitly specify the *return type* in both the function definition and the prototype declaration.

If we have a mismatch between the type of data that the called function returns and the type of data that the calling function expects, we will have unpredictable results. We must, therefore, be very careful to make sure that both types are compatible.

Example 9.4 Write a function **power** that computes x raised to the power y for integers x and y and returns double-type value.

Fig. 9.10 shows a **power** function that returns a **double.** The prototype declaration

double power(int, int);

appears in **main**, before **power** is called.

Program

```
main( )
{
  int x,y;         /*input data */
  double power(int, int);/* prototype declaration*/
  printf("Enter x,y:");
  scanf("%d %d" , &x,&y);
  printf("%d to power %d is %f\n", x,y,power (x,y));
}
double power (int x, int y);
{
  double p;
  p = 1.0 ;      /* x to power zero */

  if(y >=0)
    while(y--)    /* computes positive powers */
     p *= x;
  else
    while (y++)  /* computes negative powers */
     p /= x;
  return(p);     /* returns double type */

}
```

Output

```
Enter x,y:16 2
16 to power 2 is 256.000000

Enter x,y:16 -2
16 to power -2 is 0.003906
```

Fig. 9.10 *Power fuctions: Illustration of return of float values*

Another way to guarantee that **power**'s type is declared before it is called in **main** is to define the **power** function before we define **main. Power**'s type is then known from its definition, so we no longer need its type declaration in **main.**

9.13 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);
}
```

9.14 FUNCTIONS THAT RETURN MULTIPLE VALUES

Up to now, we have illustrated functions that return just one value using a return statement. That is because, a return statement can return only value. Suppose, however, that we want to get more information from a function. We can achieve this in C using the arguments not only to receive information but also to send back information to the calling function. The arguments that are used to "send out" information are called *output parameters.*

The mechanism of sending back information through arguments is achieved using what are known as the *address operator* (&) and *indirection operator* (*). Let us consider an example to illustrate this.

```
void mathoperation (int x, int y, int *s, int *d);
main( )
```

```
{
  int x = 20, y = 10, s, d;
  mathoperation(x,y, &s, &d);

  printf("s=%d\n d=%d\n", s,d);
}
void mathoperation (int a, int b, int *sum, int *diff)
{
  *sum  = a+b;
  *diff = a-b;
}
```

The actual arguments **x** and **y** are input arguments, **s** and **d** are output arguments. In the function call, while we pass the actual values of **x** and **y** to the function, we pass the addresses of locations where the values of **s** and **d** are stored in the memory. (That is why, the operator & is called the address operator.) When the function is called the following assignments occur:

value of x to a
value of y to b
address of s to sum
address of d to diff

Note that indirection operator * in the declaration of **sum** and **diff** in the header indicates these variables are to store addresses, not actual values of variables. Now, the variables **sum** and **diff** point to the memory locations of **s** and **d** respectively.

(The operator * is known as indirection operator because it gives an indirect reference to a variable through its address.)

In the body of the function, we have two statements:

```
* sum  = a+b;
* diff = a-b;
```

The first one adds the values **a** and **b** and the result is stored in the memory location pointed to by **sum.** Remember, this memory location is the same as the memory location of **s.** Therefore, the value stored in the location pointed to by **sum** is the value of **s.**

Similarly, the value of a–b is stored in the location pointed to by **diff,** which is the same as the location **d.** After the function call is implemented, the value of **s** is a+b and the value of **d** is a–b. Output will be:

```
s = 30
d = 10
```

The variables ***sum** and ***diff** are known as *pointers* and **sum** and **diff** as *pointer* variables. Since they are declared as **int**, they can point to locations of **int** type data.

The use of pointer variables as actual parameters for communicating data between functions is called "pass by pointers" or "call by address or reference". (Pointers and their applications are discussed in detail in Chapter 11).

Rules for Pass by Pointers

1. The types of the actual and formal arguments must be same.
2. The actual arguments (in the function call) must be the addresses of variables that are local to the calling function.
3. The formal arguments in the function header must be prefixed by the indirection operatior *.
4. In the prototype, the arguments must be prefixed by the symbol *.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator *.

9.15 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.

Consider the following program:

```
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);
}
```

The above program calculates the ratio

$$\frac{a}{b-c}$$

and prints the result. We have the following three functions:

main( )
ratio( )
difference( )

main reads the values of a, b and c and calls the function **ratio** to calculate the value a/(b–c). This ratio cannot be evaluated if (b–c) = 0. Therefore, **ratio** calls another function **difference** to test whether the difference (b–c) is zero or not; **difference** returns 1, if b is not equal to c; otherwise returns zero to the function **ratio**. In turn, **ratio** calculates the value a/(b–c) if it receives 1 and returns the result in **float.** In case, **ratio** receives zero from **difference,** it sends back 0.0 to **main** indicating that (b–c) = 0.

Nesting of function calls is also possible. For example, a statement like

```
P = mul(mul(5,2),6);
```

is valid. This represents two sequential function calls. The inner function call is evaluated first and the returned value is again used as an actual argument in the outer function call. If **mul** returns the product of its arguments, then the value of **p** would be 60 (= 5×2×6).

Note that the nesting does not mean defining one function within another. Doing this is illegal.

9.16 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.

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);
}
```

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
```

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an **if** statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

9.17 PASSING ARRAYS TO FUNCTIONS

One-Dimensional Arrays

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, *without any subscripts,* and the size of the array as arguments. For example, the call

largest(a,n)

will pass the whole array **a** to the called function. The called function expecting this call must be appropriately defined. The **largest** function header might look like:

float largest(float array[ ], int size)

The function **largest** is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[ ];
```

The pair of brackets informs the compiler that the argument **array** is an array of numbers. It is not necessary to specify the size of the **array** here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

```
main( )
{
    float largest(float a[ ], int n);
    float value[4] = {2.5,-4.75,1.2,3.67};
    printf("%f\n", largest(value,4));
}
float largest(float a[], int n)
{
    int i;
    float max;
    max = a[0];
    for(i = 1; i < n; i++)
        if(max < a[i])
        max = a[i];
    return(max);
}
```

When the function call **largest**(value,4) is made, the values of all elements of array **value** become the corresponding elements of array **a** in the called function. The **largest** function finds the largest value in the array and returns the result to the **main.**

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

Example 9.5 Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

$$S.D = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{x} - x_i)^2}$$

Where $\bar{x}$ is the mean of the values.

Program

```
#include     <math.h>
#define SIZE    5
float std_dev(float a[], int n);
float mean (float a[], int n);

main( )
{
     float value[SIZE];
     int i;

     printf("Enter %d float values\n", SIZE);
     for (i=0 ;i < SIZE ; i++)
         scanf("%f", &value[i]);
     printf("Std.deviation is %f\n", std_dev(value,SIZE));
}
float std_dev(float a[], int n)
{    int i;

     float x, sum = 0.0;
     x = mean (a,n);
     for(i=0; i < n; i++)
      sum += (x-a[i])*(x-a[i]);
     return(sqrt(sum/(float)n));
}
float mean(float a[],int n)
{
     int i ;
     float sum = 0.0;
     for(i=0 ; i < n ; i++)
        sum = sum + a[i];
     return(sum/(float)n);
}
```

Output

```
Enter 5 float values
35.0 67.0 79.5 14.20 55.75

Std.deviation is 23.231582
```

Fig. 9.11 *Passing of arrays to a function*

A multifunction program consisting of **main, std_dev,** and **mean** functions is shown in Fig. 9.11. **main** reads the elements of the array **value** from the terminal and calls the function **std_dev** to print the standard deviation of the array elements. **Std_dev,** in turn, calls another function **mean** to supply the average value of the array elements.

Both **std_dev** and **mean** are defined as **floats** and therefore they are declared as **floats** in the global section of the program.

Three Rules to Pass an Array to a Function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of the elements of an array, then these changes will be made to the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements are passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument. Example 9.6 highlights these concepts.

Example 9.6 Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function **sort()** is given in Fig. 9.12. Its output clearly shows that a function can change the values in an array passed as an argument.

Program

```
void sort(int m, int x[ ]);
main()
{
    int i;
    int marks[5] = {40, 90, 73, 81, 35};

    printf("Marks before sorting\n");
    for(i = 0; i < 5; i++)
       printf("%d ", marks[i]);
    printf("\n\n");

    sort (5, marks);

    printf("Marks after sorting\n");
    for(i = 0; i < 5; i++)
       printf("%4d", marks[i]);
    printf("\n");
```

```
            }
            void sort(int m, int x[ ])
            {
               int i, j, t;

               for(i = 1; i <= m-1; i++)
                  for(j = 1; j <= m-i; j++)
                     if(x[j-1] >= x[j])
                     {
                        t = x[j-1];
                        x[j-1] = x[j];
                        x[j] = t;
                     }
            }
          Output
                  Marks before sorting
                  40 90 73 81 35

                  Marks after sorting
                  35  40  73  81  90
```

Fig. 9.12 *Sorting of array elements using a function*

Two-Dimensional Arrays

Like simple arrays, we can also pass multidimensional arrays to functions. The approach is similar to the one we did with one-dimensional arrays. The rules are simple.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

The function given below calculates the average of the values in a two-dimensional matrix.

```
double average(int x[][N], int M, int N)
{
    int i, j;
    double sum = 0.0;
    for (i=0; i<M; i++)
         for(j=1; j<N; j++)
       sum += x[i][j];
    return(sum/(M*N));
}
```

This function can be used in a main function as illustrated below:

```
        main( )
        {
              int M=3, N=2;
              double average(int [ ] [N], int, int);
              double mean;
              int matrix [M][N]=
                      {
                        {1,2},
                        {3,4},
                        {5,6}
                      };

              mean = average(matrix, M, N);
              . . . . . .
              . . . . . .
        }
```

9.18 PASSING STRINGS TO FUNCTIONS

Because the strings are treated as character arrays in C, the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined. Example:

```
void display(char item_name[ ])
{
    . . . . . .
    . . . . . .
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

3. A call to the function must have a string array name without subscripts as its actual argument. Example:

```
display (names);
```

where **names** is a properly declared string array in the calling function.

We must note here that, like arrays, strings in C cannot be passed by value to functions.

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 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.

9.19 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* of variable determines over what region of the program a variable is actually available for use ('active'). *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. 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.

It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are *created* when the function is called and *destroyed* automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as *local* or *internal* variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable **number** in the example below is automatic.

```
main( )
{
     int number;
     -----
     -----
}
```

We may also use the keyword **auto** to declare automatic variables explicitly.

```
main( )
{
     auto int number;
     -----
     -----
}
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Example 9.7 Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig. 9.13. **m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100, and 1000 in function1, function2, and **main** respectively.

When executed, **main** calls **function2** which in turn calls **function1**. When **main** is active, m = 1000; but when **function2** is called, the **main**'s **m** is temporarily put on the shelf and the new local **m** = 100 becomes active. Similarly, when **function1** is called, both the previous values of **m** are put on the shelf and the latest value of **m** (=10) becomes active. As soon as **function1** (m=10) is finished, **function2** (m=100) takes over again. As soon it is done, **main** (m=1000) takes over. The output clearly shows that the value assigned to **m** in one function does not affect its value in the other functions; and the local value of **m** is destroyed when it leaves a function.

Program

```
void function1(void);
void function2(void);
main( )
{
        int m = 1000;
        function2();

        printf("%d\n",m); /* Third output */
}
 void function1(void)
{
        int m = 10;

        printf("%d\n",m); /* First output */
}


 void function2(void)
{
        int m = 100;

        function1();
        printf("%d\n",m); /* Second output */
}
```

```
Output
   10
   100
   1000
```

Fig. 9.13 *Working of automatic variables*

There are two consequences of the scope and longevity of **auto** variables worth remembering. First, any variable local to **main** will normally *alive* throughout the whole program, although it is *active* only in **main**. Secondly, during recursion, the nested variables are unique **auto** variables, a situation similar to function-nested **auto** variables with identical names.

External Variables

Variables that are both *alive* and *active* throughout the entire program are known as *external* variables. They are also known as *global* variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer **number** and float **length** might appear as:

```
int number;
float length = 7.5;
main( )
{
   --------
   --------
}
function1( )
{
  --------
  --------
}
function2( )
{
  --------
  --------
}
```

The variables **number** and **length** are available for use in all the three functions. In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared. Consider the following example:

```
int count;
main( )
{
    count = 10;
    -----
    -----
}
function( )
{
    int count = 0;
    ------
    ------
    count = count+1;
}
```

When the **function** references the variable **count**, it will be referencing only its local variable, not the global one. The value of **count** in **main** will not be affected.

Example 9.8 Write a multifunction program to illustrate the properties of global variables.

A program to illustrate the properties of global variables is presented in Fig. 9.14. Note that variable **x** is used in all functions but none except **fun2**, has a definition for **x**. Because **x** has been declared 'above' all the functions, it is available to each function without having to pass x as a function argument. Further, since the value of **x** is directly available, we need not use **return(x)** statements in **fun1** and **fun3**. However, since **fun2** has a definition of **x**, it returns its local value of **x** and therefore uses a **return** statement. In **fun2**, the global **x** is not visible. The local **x** hides its visibility here.

Program

```
int fun1(void);
int fun2(void);
int fun3(void);
int x ;       /* global */
main( )
{
     x = 10 ;   /* global x */
     printf("x = %d\n", x);
     printf("x = %d\n", funl());
     printf("x = %d\n", fun2());
     printf("x = %d\n", fun3());
}
funl(void)
{
     x = x + 10 ;
}
int fun2(void)
{
     int x ;          /* local */
     x = 1 ;
     return (x);
}
fun3(void)
{
     x = x + 10 ;     /* global x */
}
```

Output

```
x = 10
x = 20
x = 1
x = 30
```

Fig. 9.14 *Illustration of properties of global variables*

Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value.

Global Variables as Parameters

Since all functions in a program source file can access global variables, they can be used for passing values between the functions. However, using global variables as parameters for passing values poses certain problems.

- The values of global variables which are sent to the called function may be changed inadvertently by the called function.
- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main( )
{
    y = 5;
    . . . .
    . . . .
}
int y;      /* global declaration */
func1( )
{
      y = y+1;
}
```

We have a problem here. As far as **main** is concerned, **y** is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement

```
y = y+1;
```

in **fun1** will, therefore, assign 1 to y.

External Declaration

In the program segment above, the **main** cannot access the variable y as it has been declared after the **main** function. This problem can be solved by declaring the variable with the storage class **extern.**

For example:

```
main( )
{
    extern int y;   /* external declaration */
    . . . . .
    . . . . .
}
func1( )
{
    extern int y;   /* external declaration */
    . . . . .
    . . . . .
}
int y;              /* definition */
```

Although the variable **y** has been defined after both the functions, the *external declaration* of **y** inside the functions informs the compiler that y is an integer type defined somewhere else in the program. Note that **extern** declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.
Example:

```
main( )
{    int i;
     void print_out(void);
     extern float height [ ];
     . . . . .
     . . . . .
     print_out( );
}
void print_out(void)
{
     extern float height [ ];
     int i;
     . . . . .
     . . . . .
}
float height[SIZE];
```

An **extern** within a function provides the type information to just that one function. We can provide type information to all functions within a file by placing external declarations before any of them.

Example:

```
extern float height[ ];
main( )
{
     int i;
     void print_out(void);
     . . . . .
     . . . . .
     print_out( );
}
void print_out(void)
{
     int i;
     . . . . .
     . . . . .
}
float height[SIZE];
```

The distinction between definition and declaration also applies to functions. A function is defined when its parameters and function body are specified. This tells the compiler to allocate space for the function code and provides type information for the parameters. Since functions are external by default, we declare them (in the calling functions) without the qualifier **extern.** Therefore, the declaration

```
void print_out(void);
```

is equivalent to

```
extern void print_out(void);
```

Function declarations outside of any function behave the same way as variable declarations.

Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared *static* using the keyword **static** like

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend up to the end of the function in which they are defined. Therefore, internal **static** variables are similar to **auto** variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal **static** variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Example 9.9 Write a program to illustrate the properties of a static variable.

The program in Fig. 9.15 explains the behaviour of a static variable.

Program

```
void stat(void);
main ( )
{
   int i;
   for(i=1; i<=3; i++)
   stat( );
}
void stat(void)
{
   static int x = 0;

   x = x+1;
   printf("x = %d\n", x);
}
```

Output

```
x = 1
x = 2
x = 3
```

Fig. 9.15 *Illustration of static variable*

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to **stat**, **x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of **x** becomes three when the third call is made.

Had we declared **x** as an **auto** variable, the output would have been:

```
x = 1
x = 1
x = 1
```

This is because each time **stat** is called, the auto variable x is initialized to zero. When the function terminates, its value of 1 is lost.

An external **static** variable is declared outside of all functions and is available to all the functions in that program. The difference between a **static** external variable and a simple external variable is that the **static** external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class **static**.

Register Variables

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g., loop control variables) in the register will lead to faster execution of programs. This is done as follows:

```
register int count;
```

Although, ANSI standard does not restrict its application to any particular data type, most compilers allow only **int** or **char** variables to be placed in the register.

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert **register** variables into non-register variables once the limit is reached.

Table 9.1 summarizes the information on the visibility and lifetime of variables in functions and files.

Table 9.1 *Scope and Lifetime of Variables*

Storage Class	*Where declared*	*Visibility (Active)*	*Lifetime (Alive)*
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with **extern**	Entire program (Global)

(Contd.)

Storage Class	*Where declared*	*Visibility (Active)*	*Lifetime (Alive)*
extern	Before all functions in a file (cannot be initialized)	Entire file plus other files where variable is declared **extern** and the file where originally declared as global.	Global
static	Before all functions in a file	Only in that file	Global
None or **auto**	Inside a function (or a block)	Only in that function or block	Until end of function or block
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Inside a function	Only in that function	Global

Nested Blocks

A set of statements enclosed in a set of braces is known a *block* or a *compound* statement. Note that all functions including the **main** use compound *statement.* A block can have its own declarations and other statements. It is also possible to have a block of such statements inside the body of a function or another block, thus creating what is known as *nested blocks* as shown below:

```
main()
{
        int a = 20;
        int b = 10;
        .....                                     Outer
        {                                         block
                int a = 0;            Inner
                int c = a + b;        block
                .....
        }
        b = a;
}
```

When this program is executed, the value c will be 10, not 30. The statement b = a; assigns a value of 20 to **b** and not zero. Although the scope of **a** extends up to the end of **main** it is not "visible" inside the inner block where the variable **a** has been declared again. The inner **a** hides the visibility of the outer **a** in the inner block. However, when we leave the inner block, the inner **a** is no longer in scope and the outer **a** becomes visible again.

Remember, the variable **b** is not re-declared in the inner block and therefore it is visible in both the blocks. That is why when the statement

```
int c = a + b;
```

is evaluated, **a** assumes a values of 0 and **b** assumes a value of 10.

Although main's variables are visible inside the nested block, the reverse is not true.

Scope Rules

Scope

The region of a program in which a variable is available for use.

Visibility

The program's ability to access a variable from the memory.

Lifetime

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an **auto** variable declared in **main** is the entire program execution time, although its scope is only the **main** function.
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A **static** local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.
7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

9.20 MULTIFILE PROGRAMS

So far we have been assuming that all the functions (including the **main**) are defined in one file. However, in real-life programming environment, we may use more than one source files which may be compiled separately and linked later to form an executable object code. This approach is very useful because any change in one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must

- Where more functions are used, they may be placed in any order.
- A global variable used in a function will retain its value for future use.
- A local variable defined inside a function is known only to that function. It is destroyed when the function is exited.
- A global variable is visible only from the point of its declaration to the end of the program.
- When a variable is redeclared within its scope either in a function or in a block, the original variable is not visible within the scope of the redeclared variable.
- A local variable declared **static** retains its value even after the function is exited.
- Static variables are initialized at compile time and therefore they are initialized only once.
- Use parameter passing by values as far as possible to avoid inadvertent changes to variables of calling function in the called function.
- Although not essential, include parameter names in the prototype declarations for documentation purposes.
- Avoid the use of names that hide names in outer scope.

CASE STUDY

Calculation of Area under a Curve

One of the applications of computers in numerical analysis is computing the area under a curve. One simple method of calculating the area under a curve is to divide the area into a number of trapezoids of same width and summing up the area of individual trapezoids. The area of a trapezoid is given by

Area = 0.5 * (h1 + h2) * b

where h1 and h2 are the heights of two sides and b is the width as shown in Fig. 9.18.

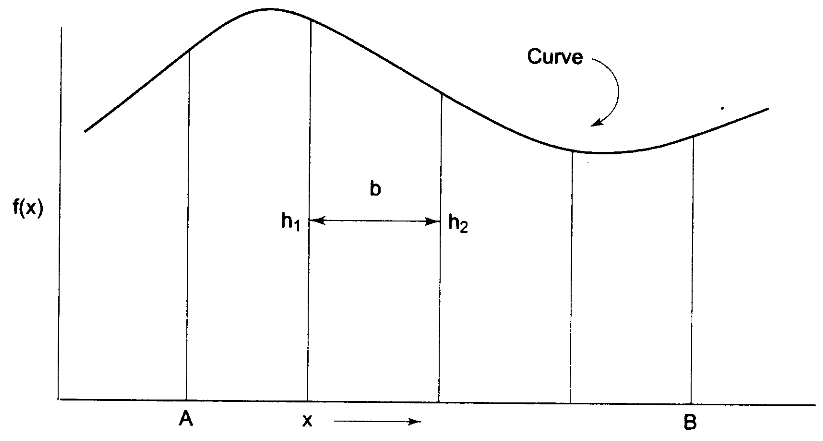


Fig. 9.18 *Area under a curve*

The program in Fig. 9.20 calculates the area for a curve of the function

$$f(x) = x^2 + 1$$

between any two given limits, say, A and B.

Input

Lower limit (A)
Upper limit (B)
Number of trapezoids

Output

Total area under the curve between the given limits.

Algorithm

1. Input the lower and upper limits and the number of trapezoids.
2. Calculate the width of trapezoids.
3. Initialize the total area.
4. Calculate the area of trapezoid and add to the total area.
5. Repeat step-4 until all the trapezoids are completed.
6. Print total area.

The algorithm is implemented in top-down modular form as in Fig. 9.19.

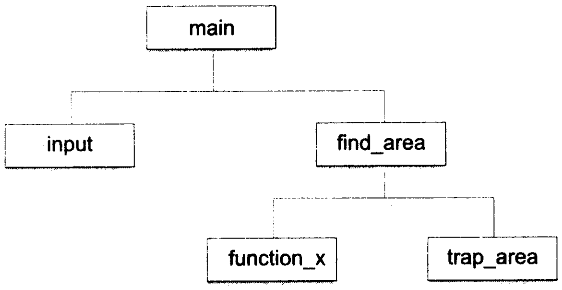


Fig. 9.19 *Modular chart*

The evaluation of f(x) has been done using a separate function so that it can be easily modified to allow other functions to be evaluated.

The output for two runs shows that better accuracy is achieved with larger number of trapezoids. The actual area for the limits 0 and 3 is 12 units (by analytical method).

Program
```
    #include <stdio.h>
    float  start_point,           /* GLOBAL VARIABLES */
           end_point,
           total_area;
    int    numtraps;
    main( )
    {
        void   input(void);
        float  find_area(float a,float b,int n); /* prototype */
    print("AREA UNDER A CURVE");
    input( );
    total_area = find_area(start_point, end_point, numtraps);
    printf("TOTAL AREA = %f", total_area);
}
void input(void)
{
    printf("\n Enter lower limit:");
    scanf("%f", &start_point);
    printf("Enter upper limit:");
    scanf("%f", &end_point);
    printf("Enter number of trapezoids:");
    scanf("%d", &numtraps);
}
float find_area(float a, float b, int n)
{
    float base, lower, h1, h2;  /* LOCAL VARIABLES */
    float function_x(float x);  /* prototype */
    float trap_area(float h1,float h2,float base);/*prototype*/

    base = (b-1)/n;
    lower = a;

     for(lower =a; lower <= b-base; lower = lower + base)
    {
        h1  = function_x(lower);
        h1  = function_x(lower + base);
        total_area += trap_area(h1, h2, base);
    }
        return(total_area);
float trap_area(float height_1,float height_2,float base)
{
  float area;     /* LOCAL VARIABLE */
  area = 0.5 * (height_1 + height_2) * base;
  return(area);
}
float function_x(float x)
{
    /* F(X) = X * X + 1 */

    return(x*x + 1);
}
```

Output

```
    AREA UNDER A CURVE
```

```
Enter lower limit: 0
Enter upper limit: 3
Enter number of trapezoids: 30
TOTAL AREA = 12.005000

AREA UNDER A CURVE
Enter lower limit: 0
Enter upper limit: 3
Enter number of trapezoids: 100
TOTAL AREA = 12.000438
```

Fig. 9.20 *Computing area under a curve*

REVIEW QUESTIONS

9.1 State whether the following statements are *true* or *false*.

(a) C functions can return only one value under their function name.
(b) A function in C should have at least one argument.
(c) A function can be defined and placed before the **main** function.
(d) A function can be defined within the **main** function.
(e) An user-defined function must be called at least once; otherwise a warning message will be issued.
(f) Any name can be used as a function name.
(g) Only a **void** type function can have **void** as its argument.
(h) When variable values are passed to functions, a copy of them are created in the memory.
(i) Program execution always begins in the main function irrespective of its location in the program.
(j) Global variables are visible in all blocks and functions in the program.
(k) A function can call itself.
(l) A function without a **return** statement is illegal.
(m) Global variables cannot be declared as **auto** variables.
(n) A function prototype must always be placed outside the calling function.
(o) The return type of a function is **int** by default.
(p) The variable names used in prototype should match those used in the function definition.
(q) In parameter passing by pointers, the formal parameters must be prefixed with the symbol * in their declarations.
(r) In parameter passing by pointers, the actual parameters in the function call may be variables or constants.
(s) In passing arrays to functions, the function call must have the name of the array to be passed without brackets.
(t) In passing strings to functions, the actual parameter must be name of the string post-fixed with size in brackets.

9.2 Fill in the blanks in the following statements.

(a) The parameters used in a function call are called ______.
(b) A variable declared inside a function is called _________.
(c) By default, ______ is the return type of a C function.
(d) In passing by pointers, the variables of the formal parameters must be prefixed with _______ in their declaration.
(e) In prototype declaration, specifying ____ is optional.
(f) _________ refers to the region where a variable is actually available for use.
(g) A function that calls itself is known as a _______ function.
(h) If a local variable has to retain its value between calls to the function, it must be declared as _________.
(i) A _______ aids the compiler to check the matching between the actual arguments and the formal ones.
(j) A variable declared inside a function by default assumes ______ storage class.

9.3 The **main** is a user-defined function. How does it differ from other user-defined functions?

9.4 Describe the two ways of passing parameters to functions. When do you prefer to use each of them?

9.5 What is prototyping? Why is it necessary?

9.6 Distinguish between the following:
(a) Actual and formal arguments
(b) Global and local variables
(c) Automatic and static variables
(d) Scope and visibility of variables
(e) & operator and * operator

9.7 Explain what is likely to happen when the following situations are encountered in a program.
(a) Actual arguments are less than the formal arguments in a function.
(b) Data type of one of the actual arguments does not match with the type of the corresponding formal argument.
(c) Data type of one of the arguments in a prototype does not match with the type of the corresponding formal parameter in the header line.
(d) The order of actual parameters in the function call is different from the order of formal parameters in a function where all the parameters are of the same type.
(e) The type of expression used in **return** statement does not match with the type of the function.

9.8 Which of the following prototype declarations are invalid? Why?
(a) `int (fun) void;`
(b) `double fun (void)`
(c) `float fun (x, y, n);`
(d) `void fun (void, void);`
(e) `int fun (int a, b);`
(f) `fun (int, float, char);`
(g) `void fun (int a, int &b);`

9.9 Which of the following header lines are invalid? Why?
(a) `float average (float x, float y, float z);`
(b) `double power (double a, int n - 1)`

(c) `int product (int m, 10)`
(d) `double minimum (double x; double y;)`
(e) `int mul (int x, y)`
(f) `exchange (int *a, int *b)`
(g) `void sum (int a, int b, int &c)`

9.10 Find errors, if any, in the following function definitions:

(a)
```
void abc (int a, int b)
{
        int c;
        . . . .
        return (c);
}
```
(b)
```
int abc (int a, int b)
{
        . . . .
        . . . .
}
```
(c)
```
int abc (int a, int b)
{
        double c = a + b;
        return (c);
}
```
(d)
```
void abc (void)
{
        . . . .
        . . . .
        return;
}
```
(e)
```
int abc(void)
{
        . . . .
        . . . .
        return;
}
```

9.11 Find errors in the following function calls:

(a) `void xyz ( );`
(b) `xyx ( void );`
(c) `xyx ( int x, int y);`
(d) `xyzz ( );`
(e) `xyz ( ) + xyz ( );`

9.12 A function to divide two floating point numbers is as follows:

```
divide (float x, float y)
{
    return (x / y);
}
```

What will be the value of the following function calls"
(a) divide (10, 2)
(b) divide (9, 2)
(c) divide (4.5, 1.5)
(d) divide (2.0, 3.0)

9.13 What will be the effect on the above function calls if we change the header line as follows:
(a) int divide (int x, int y)
(b) double divide (float x, float y)

9.14 Determine the output of the following program?

```
int prod( int m, int n);
main ( )
{
     int x = 10;
     int y = 20;
     int p, q;
     p = prod (x,y);
     q = prod (p, prod (x,z));
     printf ("%d %d\n", p,q);
}
int prod( int a, int b)
{
     return (a * b);
}
```

9.15 What will be the output of the following program?

```
void test (int *a);
main ( )
{
       int x = 50;
       test ( &x);
       printf("%d\n", x);
}
void test (int *a);
{
       *a = *a + 50;
}
```

9.16 The function **test** is coded as follows:

```
int test (int number)
{
          int m, n = 0;
          while (number)
          {
                 m = number % 10;
                 if (m % 2)
                   n = n + 1;
                 number = number /10;
          }
          return (n);
}
```

What will be the values of **x** and **y** when the following statements are executed?

```
    int x = test (135);
    int y = test (246);
```

PROGRAMMING EXERCISES

9.1 Write a function **exchange** to interchange the values of two variables, say **x** and **y.** Illustrate the use of this function, in a calling function. Assume that **x** and **y** are defined as global variables.

9.2 Write a function **space(x)** that can be used to provide a space of x positions between two output numbers. Demonstrate its application.

9.3 Use recursive function calls to evaluate

$$f(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots..$$

9.4 An n_order polynomial can be evaluated as follows:

$$P = (\ldots..(((a_0x+a_1)x+a_2)x+a_3)x+..+a_n)$$

Write a function to evaluate the polynomial, using an array variable. Test it using a main program.

9.5 The Fibonacci numbers are defined recursively as follows:

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1}+F_{n-2}, n > 2$

Write a function that will generate and print the first n Fibonacci numbers. Test the function for n = 5, 10, and 15.

9.6 Write a function that will round a floating-point number to an indicated decimal place. For example the number 17.457 would yield the value 17.46 when it is rounded off to two decimal places.

9.7 Write a function **prime** that returns 1 if its argument is a prime number and returns zero otherwise.

9.8 Write a function that will scan a character string passed as an argument and convert all lower-case characters into their uppercase equivalents.

9.9 Develop a top_down modular program to implement a calculator. The program should request the user to input two numbers and display one of the following as per the desire of the user:

(a) Sum of the numbers

(b) Difference of the numbers

(c) Product of the numbers

(d) Division of the numbers

Provide separate functions for performing various tasks such as reading, calculating and displaying. Calculating module should call second level modules to perform the individual mathematical operations. The main function should have only function calls.

9.10 Develop a modular interactive program using functions that reads the values of three sides of a triangle and displays either its area or its perimeter as per the request of the user. Given the three sides a, b and c.

$$\text{Perimeter} = a + b + c$$

$$\text{Area} = \sqrt{(s-a)(s-b)(s-c)}$$

$$\text{where } s = (a+b+c)/2$$

9.11 Write a function that can be called to find the largest element of an m by n matrix.

9.12 Write a function that can be called to compute the product of two matrices of size m by n and n by m. The main function provides the values for m and n and two matrices.

9.13 Design and code an interactive modular program that will use functions to a matrix of m by n size, compute column averages and row averages, and then print the entire matrix with averages shown in respective rows and columns.

9.14 Develop a top-down modular program that will perform the following tasks:
(a) Read two integer arrays with unsorted elements.
(b) Sort them in ascending order
(c) Merge the sorted arrays
(d) Print the sorted list
Use functions for carrying out each of the above tasks. The main function should have only function calls.

9.15 Develop your own functions for performing following operations on strings:
(a) Copying one string to another
(b) Comparing two strings
(c) Adding a string to the end of another string
Write a driver program to test your functions.