FUNCTIONS

- In addition to function **main**, a program source module may contain one or more other functions.
- The execution of the program always starts with function **main**.
- The statements of a function (other than function **main**) are executed only if that function is called in function **main** or any other function that is called in function **main**.
- After the statements of a function are executed, the next statement to be executed is the one that follows the statement in which that function is called.

Example

The statements of the program in figure F1 are executed in the following order: 23, 24, 26, 15, 16, 27, 28, 31, 32, 34, 15, 16, 35, 36, and 37.

- There are two types of functions in the C/C++ programming language:
  - Functions that do not return a value: ***void*** functions, and
  - Functions that return a value.
- You can also write a function with or without *parameters*:
  - An example of a **void** function without parameters is provided in figure F1, line 13 to line 17.
  - An example of a **void** function with parameters is provided in figure F2, line 8 to line 12.

Defining and Calling a *void* Function without Parameters

- You define a **void** function without parameters as follows:

```
void  <function-name>( )
{
     <Body-of-the-function>
}
```

<function-name>	is the **name of the function**
void <function-name>( )	is the **function header**. It may also be specified as follows: *void <function-name>(void)*
<Body-of-the-function>	is the **body of the function:** It consists of one or more statements that are executed each time the function is *called*.

- You **call** a *void* function without parameters by using the following statement:

  <*function-name*> ( );

- The program in figure F1 illustrates the definition and calls of a **void** function without parameters.

Figure F1 Defining and Calling a *void* Function without Parameters

```
1. /*----- Program to compute the area and the perimeter of rectangles ------*/
2.
3. #include   <iostream>
4. using namespace std;
5.
6. double  len,      // to hold the length of the rectangle
7.          width    // to hold the width of the rectangle
8.          area,    // to hold the area
9.          peri;    // to hold the perimeter
10.
11.   /*------------------------function computeAreaPeri1 ---------------------*/
12.   /*------- compute the area and the perimeter of a rectangle -----------*/
13.   void computeAreaPeri1( void )
14.   {
15.      area  = len * width;
16.      peri =  2 * ( len + width );
17.   }
18.
19.   int main ()
20.   {
21.      /*-------compute and print the area and the perimeter of a rectangle with
            length 20 and width 8  ---------------------------------------------*/
22.
23.      len = 20;
24.      width = 8;
25.
26.      computeAreaPeri1( );
27.      cout  << endl  << "the area of the rectangle is:\t"   << area;
28.      cout  << endl  << "Its perimeter is:\t"   <<    peri;
29.
30.      /*-------read the length and the width of a rectangle and compute and
            print its area and perimeter  ------------------------------------ */
31.      cout  << endl  << "enter the length and the width of the rectangle:\t";
32.      cin   >> len   >> width;
33.
34.      computeAreaPeri1( );
35.      cout  << endl  << "the area of the rectangle is:\t"   << area;
36.      cout  << endl  << "Its perimeter is:\t"   <<    peri;
37.         return ( 0 );
      }
```

Global Variables and Local Variables

- A **global variable** is a variable that is defined outside of the body of any function.

  Examples of global variables are variables *len*, *width*, *area*, and *peri* defined in figure F1, line 6 to line 9.

- A global variable can be accessed in the body of any function that appears after its definition.

Example

In the source module in figure F1,

len

is accessed in the body of function *computeAreaPeri1* in lines 15 and 16

and in the body of function *main* in lines 23 and 32.

- A function can use *global variables* to share information with the calling function.

Example

In the source module in figure F1,

- Function *main* stores 20 into variable *len* in line 23, and 8 into variable *width* in line 24 before it calls function *computeAreaPeri1* in line 26.
- In line 15, function *computeAreaPeri1* retrieves 20 from variable l*en* and 8 from variable *width*, multiplies 20 by 8, and stores the result, 160.0 in variable *area*.
- In line 16, function *computeAreaPeri1* retrieves 20 from variable *len* and 8 from variable *width*, computes 2 * (20 + 8), and stores the result, 56.0 in global variable *peri*.
- In line 27, function *main* retrieves 160.0 from global variable *area* and prints it.
- In line 28, function *main* retrieves 56.0 from global variable *peri* and prints it.

- A variable that is defined in the body of a function is a **local variable** of that function.
- A local variable can only be accessed in the body of the function in which it is defined.

Exercise F1*

Execute the following program and show its output for the input value 7:

```
#include   <iostream>
using namespace std;

int gnum1, gnum2;

void funct(void)
{
    int num =  gnum1 + 10;
    gnum1  + =  num;
    gnum2 = 2 * gnum1 + 5;
}
int main( )
{
    gnum1 = 15;
    funct( );
    cout  <<  endl  << "gnum1="  <<  gnum1  <<  "\tgnum2="  <<  gnum2;
    cin  >>  gnum1;
    funct( );
    cout  <<  endl  << "gnum1="  <<  gnum1  <<  "\tgnum2="  <<  gnum2;
    return 0;
}
```

Exercise F2*

Write a **void** function without parameters *computeAreaPeri* that computes the area and the perimeter of a circle which are output in function **main**.

a. Function **main** first calls function *computeAreaPeri* to compute the area and the perimeter of the circle with radius 5.43.

b. It then reads the radius of a circle and then calls function *computeAreaPeri* again to compute the area and the perimeter of this circle.

You must first determine and define the global variables of this program; then write function *computeAreaPeri*, and finally write function *main*.

Exercise F3

Execute the following program and show its output for the input value 15:

```
#include   <iostream>
    using namespace std;

    int gnum1, gnum2;
```

```
void funct(void)
        {
            int num = 2 * gnum1 + 5;
            gnum2 = gnum1 + num;
            gnum1 = gnum2 + 10;
        }

        int main( )
        {
            gnum1 = 25;
            funct( );
            cout << endl << "gnum1=" << gnum1 << "\tgnum2=" << gnum2;
            cin >> gnum1;
            funct( );
            cout << endl << "gnum1=" << gnum1 << "\tgnum2=" << gnum2;
            return 0;
        }
```

Exercise F4

Write a **void** function without parameters *computeTaxNet* that uses the gross pay of an individual to compute his tax deduction and net pay that are printed in function **main**. The tax deduction is computed as follows: if the gross pay is greater than or equal to $1000.00, then the tax deduction is 25% of the gross pay; otherwise, it is 18% of the gross pay. The net pay is the gross pay minus the tax deduction.

a. Function **main** first calls function *computeTaxNet* to compute the tax deduction and the net pay for the gross pay $1250.

b. It then reads the gross pay of an individual and then calls function *computeTaxNet* to compute his tax deduction and net pay.

c. You must first determine and define the global variables of this program; then write function *computeTaxNet*, and finally write function **main**.

Writing and Calling a void Function with Parameters

- You write a **void** function with parameters by specifying the parameters and their data types in the parentheses that follow the function name in the function header as follows:

```
void  <function-name> ( type1 parameter1, type2 parameter2, type3 parameter3, . . .  )
{
        <Body-of-the-function>
}
```

- There are two types of parameters in C++: **value parameters** and **reference parameters**.
- You distinguish a *value parameter* from a *reference parameter* by placing the ampersand character, **&** in front of reference parameters.

Example

```
void   computeAreaPeri2( double len, double wid, double &ar, double &peri )
{
    ar   =  len  *  wid;
    peri  =  2 * ( len + wid );
}
```

- *len* and *wid* are value parameters.
- *ar* and *peri* are reference parameters.

- Parameters are used in the body of a function in the same way that its local variables are used.

Calling a *void* Function with Parameters

- **You call of a *void* function with parameters** by using a statement consisting of the function name followed by the **arguments** between the parentheses as follows:
  - For each value parameter, the argument must be a value or an expression.
  - For each reference parameter, the argument must be a variable name.
  - You must also specify an argument for each parameter.
  - The data type of an argument must be compatible with the data type of the corresponding parameter.

Examples

Given the following definitions of variables:

```
double  length = 25.00,  width  =  10.00,
        area, perimeter;
```

The following calls of the function *computeAreaPeri2* are valid:

a. computeAreaPeri2(27.42, 16.25, area, perimeter);

b. computeAreaPeri2(length, width, area, perimeter);

c. computeAreaPeri2(length + 5.0, 8.50, area, perimeter);

d. computeAreaPeri2(length, width * 2, area, perimeter);

The following calls of the function *computeAreaPeri2* are invalid:

a. computeAreaPeri2(27.42, 16.25, 50.5, perimeter);

   the argument that corresponds to the reference parameter ar is 50.5 which is not a variable name.

b. computeAreaPeri2(length, width, area, perimeter + 4);

   the argument that corresponds to the reference parameter peri is perimeter + 5 which is not a variable name.

c. computeAreaPeri2(8.50, area, perimeter);

   only 3 arguments (instead of 4) are provided.

Exercise F5

Assume given the following definition of function *process*:

```
void   process(int vnum, int & rnum1, int & rnum2)
{
    vnum  =  vnum + 5;
    rnum1  = rnum1 + 10;
    rnum2 = vnum + rnum1;
}
```

And that local variables are defined in function *main* as follows:

```
int   num1 = 6,  num2 = 9,  num3 = 12;
```

Indicate whether each of the following calls of function *process* is valid or invalid. Also give the reason why a call is invalid.

a. process(15, num2, num3);
b. process(21, num1 + 3, num2);
c. process(num3, num1, num2);
d. process(num1, 10, num3);
e. process(3, num1, 25);
f. process(num1, num2, num3 + 4);
g. process(num2, num1);
h. process(num1 + 3, num2, num3);
i. process(5, num3);
j. process(num1, num1, num3);

Processing Arguments/Parameters in the Body of a Function

- The following actions take place when a function with parameters is called:
  - A memory location is created for each value parameter and the value of the corresponding argument is stored into it.
  - Each reference parameter is replaced in the body of the function with the corresponding argument.

Example

Given the following definition of function *computeAreaPeri2( )*:

```
void   computeAreaPeri2( double len, double wid, double &ar, double &peri )
{
    ar   =  len  *  wid;
    peri  =  2 * ( len + wid );
}
```

Assume that local variables are defined in function **main** as follows:

```
double  length = 25.00,  width  =  10.00,
        area, perimeter;
```

a. After the call statement: **computeAreaPeri2(80.00, 50.00, area, perimeter);**

The body of function *computeAreaPeri2* is executed as if it was written as follows:

```
{
    len  =  80.00;
    wid  =  50.00;
    area   =  len  *  wid;
    perimeter =  2 * ( len + wid );
}
```

After the execution of function *computeAreaPeri2*, local variables *area* and *perimeter* of function **main** will have the following values:

area: 4000.00, perimeter: 260.00

b. After the call statement: **computeAreaPeri2(length + 5, width, area, perimeter);**

The body of function *computeAreaPeri2* is executed as if it was written as follows:

```
{
    len  =  30.00;
    wid  =  10.00;
    area  =  len  *  wid;
    perimeter =  2 * ( len + wid );
}
```

After the execution of function *computeAreaPeri2*, local variables *area* and *perimeter* of function **main** will have the following values:

area: 300.00, perimeter: 80.00

Exercise F6

Assume given the following definition of function *process*:

```
void   process(int vnum, int & rnum1, int & rnum2)
{
    vnum  =  vnum + 5;
    rnum1  = rnum1 + 10;
    rnum2 = vnum + rnum1;
}
```

And that local variables are defined in function **main** as follows:

```
int   num1 = 6,  num2 = 9,  num3 = 12;
```

1. For each of the following calls of function *process*, show the body of function *process* in the way it is executed after the call.
2. For each of the following calls, execute function *process* and display the corresponding output.
   a. process(15, num2, num3);
      cout << endl << "num1=" << num1 << "num2=" << num2 << "num3=" << num3;
   b. process(num1, num2, num3);
      cout << endl << "num1=" << num1 << "num2=" << num2 << "num3=" << num3;
   c. process(num1 + 3, num2, num3);
      cout << endl << "num1=" << num1 << "num2=" << num2 << "num3=" << num3;

Exercise F7

Assume given the following definition of function *computeAreaPeri2* that computes the area and the perimeter of a rectangle given its length and width:

```
void   computeAreaPeri2( double len, double wid, double &ar, double &peri )
{
    ar   =  len  *  wid;
    peri  =  2 * ( len + wid );
}
```

a. Write the sequence of statements to compute and print the area and the perimeter of a rectangle with length 70 and width 45.
b. Write the sequence of statements to read the length and the width of a rectangle and to compute and print its area and perimeter.

Using Parameters to Share Values between a Function and the Calling Function

- When you write a **void** function with parameters, chose the parameters as follows:
  - Chose a value parameter for every value that is passed to the function.
  - Chose a reference parameter for every value that the function must return to the calling function.

Example

In the source module in figure F2, function *computeAreaPeri2* uses the parameters as follows:

— Value parameter *len* is used to get the length of the rectangle from function **main**.

— Value parameter *wid* is used to get the width of the rectangle from function **main**.

— Reference parameter *ar* is used to send back the area of the rectangle to function **main**.

— Reference parameter *peri* is used to send back the perimeter of the rectangle to function **main**.

Figure F2 Defining and Calling Functions with Parameters

```
1. /*----- Program to compute the area and the perimeter of rectangles ------*/
2.
3. #include   <iostream>
4. using namespace std;
5.
6. /*------------------------function computeAreaPeri2 ---------------------*/
7. /*------- compute the area and the perimeter of a rectangle-------------*/
8. void computeAreaPeri2( double len, double wid, double &ar, double &peri )
9. {
10.      ar  = len * wid;
11.      peri =  2 * ( len + wid );
12.   }
13.
14.   int main ()
15.   {
16.      double  length,            // to hold the length of the rectangle
17.           Width,                // to hold the width of the rectangle
18.           area,                 // to hold the area
19.           perimeter;            // to hold the perimeter
20.
21.      /*-------compute and print the area and the perimeter of a rectangle with
            length 45 and width 34 ------------------------------------------*/
22.
23.      computeAreaPeri2( 45, 34, area, perimeter );
24.      cout  << endl  << "the area of the rectangle is:\t"   << area;
25.      cout  << endl  << "Its perimeter is:\t"   <<    perimeter;
26.
27.      /*-------read the length and the width of a rectangle and compute and
            print its area and perimeter  ------------------------------------ */
28.      cout  << endl  << "enter the length and the width of the rectangle:\t";
29.      cin   >> length   >> width;
30.
31.      computeAreaPeri2( length, width, area, perimeter );
32.      cout  << endl  << "the area of the rectangle is:\t"   << area;
33.      cout  << endl  << "Its perimeter is:\t"   <<    perimeter;
34.      return ( 0 );
35.   }
36.
```

Exercise F8

Write a **void** function with parameters named *computeAreaPeri3* that gets the radius of a circle and then computes its area and perimeter and returns them to the calling function: first determine the number and the types of the parameters that are needed by the function before you write it.

Exercise F9

Assume given the following definition of function *process*:

```
void   process(int vnum, int & rnum1, int & rnum2)
{
    vnum  =  vnum + 5;
    rnum1  = rnum1 + 10;
    rnum2 = vnum + rnum1;
}
```

And that local variables are defined in function **main** as follows:

```
int   num1 = 10,  num2 = 15,  num3 = 20;
```

Indicate whether each of the following calls of function *process* is valid or invalid and:

— For each invalid call, provide the reason why it is invalid.

— Do the following for each valid call:

i. show the body of the function *process* in the way it is executed after the function call,

ii. then execute function *process* and show the contents of the variables *num1*, *num2*, and *num3* after the function is executed.

a. process(num1, num2, num3);
b. process(num2, num3, num1);
c. process(num1, 10, num3);
d. process(3, num1, num2);
e. process(num1, num2, num3 + 4);
f. process(num1 + 3, num2, num3);
g. process(5, num3);
h. process(num1, num1, num3);

Exercise F10

1. Write a **void** function named *computeProductSum* with parameters that gets two integer values and then computes their product and their sum and returns them to the calling function: first determine the number and the types of the parameters that are needed by the function before you write it.
2. Write the sequence of statements to compute the product and the sum of 15 and 79 (by calling function *computeProductSum*) and then print them.

Exercise F11

Write a **void** function named *swapper* with parameters that gets two variables as arguments and interchanges their values if the value of the first variable is less than that of the second variable.

Basic Properties of Functions

1. **Definition** A function is defined only once by writing its function header followed by its constituent statements enclosed between the left brace **{** and the right brace **}**.
2. **Execution** At any point in another function where the effect of a function is needed, use a call statement to pass the control of the execution of a program to that function. After the execution of the last statement in the body of a function, the control of the execution of the program returns to the statement that follows the call statement.
3. **Placement** A function may be called in a source module only if it has been written (defined) or ***declared*** before the call statement.

Example

In the source module in figure F1, function *computeAreaPeri1* is written before function **main**, and in the source module in figure F2 function *computeAreaPeri2* is written before function **main**.

- You **declare a function** by writing its function header followed by the semicolon. This statement is called **function prototype**.

Example

1. The function prototype of function *computeAreaPeri1* define in the source module in figure F1 is:

   void computeAreaPeri( ); or

   void computeAreaPeri(void);

2. The function prototype of function *computeAreaPeri2* defined in the source module in figure F2 is:

   void computeAreaPeri2(double len, double wid, double &ar, double &peri); or

   void computeAreaPeri2(double, double, double &, double &);

Notes:

1. The names of parameters may be omitted in the function prototype of a function with parameters.
2. It is a good programming practice to precede a function call with the function prototype of that function, and to write all the functions (in a source module) after function **main** (as it is done in the source modules in figures F3 and F4).

Figure F3 **Using the Function Prototype of a Function without Parameters**

```
1. /*----- Program to compute the area and the perimeter of rectangles ------*/
2.
3. #include   <iostream>
4. using namespace std;
5.
6. void  computeAreaPeri(   );   //to compute the area and perimeter of a rectangle
7.
8. double  len,                  // to hold the length of the rectangle
9.          width                // to hold the width of the rectangle
10.         area,                // to hold the area
11.         peri;                // to hold the perimeter
12.
13.
14.   int main ()
15.   {
16.      /*-------compute and print the area and the perimeter of a rectangle with
            length 45 and width 34 ----------------------------------------*/
17.
18.      len = 45;
19.      width = 34;
20.
21.      computeAreaPeri( );
22.      cout  << endl  << "the area of the rectangle is:\t"   << area;
23.      cout  << endl  << "Its perimeter is:\t"   <<    peri;
24.
25.      /*-------read the length and the width of a rectangle and compute and
            print its area and perimeter  ------------------------------------ */
26.      cout  << endl  << "enter the length and the width of the rectangle:\t";
27.      cin   >> len   >> width;
28.
29.      computeAreaPeri( );
30.      cout  << endl  << "the area of the rectangle is:\t"   << area;
31.      cout  << endl  << "Its perimeter is:\t"   <<    peri;
32.      return ( 0 );
33.   }
34.
35.   /*------------------------function computeAreaPeri ---------------------*/
36.   /*------- compute the area and the perimeter of the rectangle-----------*/
37.   void computeAreaPeri(  )
38.   {
39.      area  = len * width;
40.      peri =  2 * ( len + width );
41.   }
```

Figure F4 **Using the Function Prototype of a Function with Parameters**

```
1.    /*----- Program to compute the area and the perimeter of rectangles ------*/
2.
3.    #include   <iostream>
4.    using namespace std;
5.
6.    void  computeAreaPeri2( double len, double wid, double &ar, double &peri  );
7.       //to compute the area and perimeter of a rectangle
8.
9.    int main ()
10.   {
11.      double  length,            // to hold the length of the rectangle
12.           Width,                // to hold the width of the rectangle
13.           area,                 // to hold the area
14.           perimeter;            // to hold the perimeter
15.
16.      /*-------compute and print the area and the perimeter of a rectangle with
            length 45 and width 34 ------------------------------------------*/
17.
18.      computeAreaPeri2( 45, 34, area, perimeter );
19.      cout  << endl  << "the area of the rectangle is:\t"   << area;
20.      cout  << endl  << "Its perimeter is:\t"   <<    perimeter;
21.
22.      /*-------read the length and the width of a rectangle and compute and
            print its area and perimeter  ------------------------------------ */
23.      cout  << endl  << "enter the length and the width of the rectangle:\t";
24.      cin   >> length   >> width;
25.
26.      computeAreaPeri2( length, width, area, perimeter );
27.      cout  << endl  << "the area of the rectangle is:\t"   << area;
28.      cout  << endl  << "Its perimeter is:\t"   <<    perimeter;
29.      return ( 0 );
30.   }
31.
32.   /*------------------------function computeAreaPeri2 ---------------------*/
33.   /*------- compute the area and the perimeter of the rectangle-----------*/
34.   void computeAreaPeri2( double len, double wid, double &ar, double &peri )
35.   {
36.      ar  = len * wid;
37.      peri =  2 * ( len + wid );
38.   }
```

Exercise F12

Write the function prototypes of the following functions:

a. Function *computeTaxNet* that you wrote in exercise F4.

b. Function *computeAreaPeri3* that you wrote in exercise F8.

c. Function *computeProductSum* that you wrote in exercise F10.

d. Function *swapper* that you wrote in exercise F11.

Functions that Returns a Value

- You write a function that returns a value in the same way that you write a **void** function except for the following two conditions:

  1. The return type of a function that returns a value must be a valid C++ data type such as **bool**, **char**, **int**, **float**, or **double**.
  2. The body of a function that returns a value must include at least one ***return*** statement.

  Examples of functions that return a value follow: the first has no parameter whereas the second one does.

Example of a Function without Parameters that Returns a Value

```
/*--------- this function returns 5 whenever it is called ----------------*/
int   process( void )
{
     return 5;
}
```

Example of a Function with Parameters that Returns a Value

```
/*----this function gets two double precision values, computes and returns their average --*/
double  computeAverage( double num1, double num2 )
{
     double avg;
     avg  =  (num1 + num2) / 2;
     return( avg );
}
```

The return Statement

- The syntax of the **return** statement follows:

  return <expression>;

  or

  return (<expression>);

  where <*expression*> is an expression that evaluates to a value whose data type is compatible with the function's return type.

- <*expression*> is first evaluated and its value is supplied to the calling function when the control of the execution of the program returns to it.

Executing the return Statement

Given the following definitions of variables:

```
char ch = 'A';
int num = 5;
double dnum = 7.2;
```

The following return statements evaluate as follows:

return Statement	**Function Return Type**	**Value Returned**
return ch;	char	A
return num + 6;	int	11
return dnum – 3;	double/float	4.2
return dnum – 3;	int	4
return num +6;	double/float	11.0

- The body of a *void* function may (optionally) have a return statement without any expression as follows:

  return;

- The execution of a function terminates whenever a **return** statement is executed in that function.
- If a **return** statement is not met during the execution of a function, the execution of that function terminates with its last closing brace.

Note:

The return statement cannot be used to return more than one value.

Calling a Function that returns a Value

- **You call a function without parameters that returns a value** by writing the function name followed by the left and the right parentheses **function-name( )** in any expression in which the value returned by that function could be written.

Example

Given the following function *process( )*:

```
/*--------- this function returns 5 whenever it is called ----------------*/
int   process( void )
{
     return 5;
}
```

The following calls of function *process( )* will produce the specified output.

1. cout << endl << "Result1=\t" << (process( ) + 4);

   OUTPUT Result1 = 9

2. int num1, num2 = 12;
   num1 = num2 - process( ) ;
   cout << endl << "Result2 =\t" << num1;

   OUTPUT Result2 = 7

Note: If you call function *process( )* as a *void* function:

proess( );

nothing will happen.

- **You call a function with parameters that returns a value** by writing:

  <function-name> (<Arg1>, <Arg2>, . . ., <ArgN>)

  in an expression in which the value returned could be written.

- You specify the arguments <Arg1>, <Arg2>, . . ., <ArgN> in the same way that you specify the argument of a **void** functions with parameters.

Example

Given the following function *computeAverage( )*

```
/*----this function gets two double precision values, computes and returns their average --*/
    double  computeAverage( double num1, double num2 )
    {
        double avg;
        avg  =  (num1 + num2) / 2;
        return( avg );
    }
```

The following calls of function *computeAverage( )* will produce the specified output:

1. `cout << endl << "Average1=\t" << computeAverage( 10, 5 );`

   OUTPUT Average1 = 7.5

2. ```
   int num1, num2 = 12;
   num1 = num2 + computeAverage( num2, 4 ) ;
   cout << endl << "Average2 =\t" << num1;
   ```

   OUTPUT Average2 = 20.0

Function Prototype of a Function that Returns a Value

- You specify the function prototype of a function that returns a value in the same way that you specify the function prototype of a void function.

Example

a. The function prototype of function *process* defined above is:

   `int  process( );` or `int  process( void );`

b. The function prototype of function *computeAverage* defined above is:

   `double  computeAverage( double num1, double num2 );` or

   `double  computeAverage( double, double );`
   ```

Exercise F13

Given the following definitions of functions *process* and *computeAverage*:

```
int   process( void )
{
    return 5;
}
```

```
double  computeAverage( double num1, double num2 )
{
    double avg;
    avg  =  (num1 + num2) / 2;
    return( avg );
}
```

A. Show the output after the execution of each of the following sequences of statements:

   a. `cout << endl << "Result1=\t" << process( ) + 2;`

   b.
   ```
   int num1 = 3, num2;
   num2 = process( ) - num1;
   cout << endl << "Result2=\t" << num2;
   ```

   c. `cout << endl << "Average1=\t" << computeAverage( 20, 5 );`

   d.
   ```
   int num1 = 4, num2 = 6;
   cout << endl << "Average2=\t" << computeAverage( num1 + 3, num2 - 2 );
   ```

B. Write the sequence of statements to read two floating point values and to compute (using function *computeAverage*) their average and print it.

Exercise F14

a. Write a function with parameter *int computeProduct* that receives as arguments two integer values, computes their product and return it to the calling function.

b. Write the sequence of statements to compute (using function *computeProduct*) the product of 123 and 567 and print the result.

c. Write the sequence of statements to read two integer values and to compute (using function *computeProduct*) their product and print it.

d. Write the function prototype of function *computeProduct*.

Local Variables and Program Blocks

- A C++ program block is any sequence of statements enclosed between the left brace { and the right brace }.

Example

```
/*---------- Program to read a certain number of values and to compute their average-------*/
int  main(   )
{
    int  total = 0, count;
    { // beginning of the block
        int  num;        // variable num is a local variable of this program block

        /*---------------------read the number of values to be read ---------------------------------------*/
        cout  <<  endl  <<  "Enter the number of values:\t"
        cin  >>  count;

        /*---------------------- read all the values and compute their sum ----------------------------*/
        for ( int  i = 0;  i  <  count ;  i ++ )  // variable i is a local variable of  the for-loop block
        {
            cin  >>  num;
            total + = num;
        }
        Cout  <<  endl  <<  "The last value that you have entered is:\t"  <<  num;
    } // end of the block
    Cout  <<  endl  <<  "The average of the values read is:\t"  <<  total / count;
    return 0;
}
```

- A variable defined inside a program block is a local variable of that program block and is accessible only inside that program block.

Scope of Variables

- The **scope of a variable** refers to the extent of a program over which that variable name identifies the same memory location:
  - the scope of a local variable is therefore the body of the function or the program block in which it is defined, and
  - the scope of a global variable is the body of any function that is defined after that global variable.
- Local variables in different functions may have the same name, and a local variable may have the same name as a global variable.
- When a local variable has the same name as a global, any reference to that name in the function in which the local variable is defined refers to the local variable.

Example

The output of the following program is:

Output

output from function main:

num = 10

output from function proc1:

num = 5

value =6

output from function proc2:

num = 10

value =8

Scope of Variables

```
#include <iostream>
using namespace std;
int num = 10;
void proc1(void)
{
    int num = 5;
    int value = 6;
    cout << endl << "num =\t" << num;
    cout << endl << "value =\t" << value;
}
void proc2(void)
{
    int value = 8;
    cout << endl << "num =\t" << num;
    cout << endl << "value =\t" << value;
}
int main()
{
    cout << endl << "output from function main:";
    cout << endl << "num =\t" << num;
    cout << endl << "output from function proc1:"
    Proc1();
    cout << endl << "output from function proc2:"
    Proc2();
}
```

Exercise F15

Provide the output of the following program.

```
#include <iostream>
using namespace std;

int num = 7;
void proc1(void)
{
    int num = 10,  value = 8;
    cout    << endl    <<      "num =\t"      <<      num;
    cout    << endl    <<      "value =\t"    <<      value;
}

void proc2(void)
{
    int value = 4;
    cout    << endl    <<      "num =\t"      <<      num;
    cout    << endl    <<      "value =\t"    <<      value;
}

int main()
{
    cout    << endl    << "output from function main:";
    cout    << endl    <<      "num =\t"      <<      num;
    cout    << endl    << "output from function proc1:"
    Proc1();
    cout    << endl    << "output from function proc2:"
    Proc2();
}
```

Default Arguments

- In C++, you can specify in the function prototype of a function, the **default arguments** of some of that function's parameters in the following way:
  1. Follow the parameter's name or the parameter's type with the equal sign (=), which is then followed by the default argument.
  2. Default arguments may only be specified for consecutive parameters, starting with the right-most parameter.

- The following are valid function prototypes with default arguments:
    a) void foo(int, int = 5);
    b) int goo(float, double, int = 0);
    c) double hoo(char ch = 'A', double num = 100.00, int count = 20);

- The following are invalid function prototypes with default arguments:
    a. void joo(int = 0, int); *// default arguments are not assigned from left-to-right.*
    b. int koo(char = 'A', int , int = 0);
    / the first parameter has a default argument but the second does not have one */*

- After a default argument has been specified for a function's parameter, you may call that function without providing the argument that corresponds to that parameter: the compiler uses the default argument when one is not provided as illustrated in figure F5.
- Default arguments may be constants, global variables, or function calls.
- But they are in general constants that occur frequently when the function is called, and their use saves writing in these arguments at each invocation of the function.
- When a function with default arguments is called, the arguments supplied in the function call are associated to the parameters from left to right.
- Any remaining parameters are associated to their default arguments. Figure F5 illustrates calls to functions with default arguments.

Figure F5 Calling a Function with Default Arguments

Assuming that functions *foo*, *goo*, and *hoo* have the following function prototypes:

a) void foo(int, int = 5);
b) int goo(float, double, int = 0);
c) double hoo(char ch = 'A', double num = 100.00, int count = 20);

The following table shows how calls to these functions are processed by the compiler.

Function Calls	Processed by the Compiler as follows:
foo(num, 2);	foo(num, 2);
foo(num);	foo(num, 5);
result1 = goo(7.2, fnum, 9);	result1 = goo(7.2, fnum, 9);
result1 = goo(7.2, fnum);	result1 = goo(7.2, fnum, 0);
result2 = hoo( );	result2 = hoo('A', 100.00, 20);
result2 = hoo('F', 85.5);	result2 = hoo('F', 85.5, 20);
result2 = hoo('Z', 17.5, 30);	result2 = hoo('Z', 17.5, 30);

Exercise F16

Assuming that functions *joo(), koo(),* and *loo()* have the following function prototypes:

a) void joo(int, int = 9);
b) void koo(int, char = 'F', int = 0);
c) void loo(int MaxCount = 100, double rate = 8.25, double base = 250.0);

Specify how each of the following function calls is processed by the compiler:

1. joo(5, 15);
2. joo(20);
3. koo(25, 'B', 15);
4. koo(15, 'A');
5. koo(5);
6. loo( );
7. loo(50, 6.5);
8. loo(150);
9. loo(200, 5.5, 175.5);

Function Name Overloading

- Very often, two or more functions of a program conceptually perform the same task, but the number and/or the data types of some of their arguments are different.
- For example, you may have a function to compute the average of three integer values, and another one to compute the average of three double precision floating-point values.

- You may also have a function to compute the average of two integer values, and another one to compute the average of three integer values.
- Giving the same name to these functions can make the program easier to read and understand
- In C++ two or more functions may have the same name, as long as there is a way to distinguish them based on their parameters.
- This feature is called **function name overloading**.
- The compiler determines the right version of the function to call from a set of overloaded functions by inspecting the arguments specified in the function call.
- The example in Figure 14 illustrates the use of overloaded function names in a source module.

Figure F6 Function Name Overloading

```
Line Number
1     /*****************************************************************
2     Program to compute the average of two integer values, the average
3     of three integer values and the average of two double precision
4      floating-point values.
5     *****************************************************************/
6     #include   <iostream>
7     using namespace std;

8     int ComputeAverage( int, int);
9         //  to compute the average of two integer  values
10    int ComputeAverage( int, int, int);
11       // to compute the average of three integer  values
12    double ComputeAverage(double, double);
13      //to compute the average of two floating-point values
14
15    int main()
16    {
17       int result1,      // the average of two integer values
18          result2;       // the average of three integer values
19       double result3;   //the average of two floating-point values
20
21       result1 = ComputeAverage(4, 5);     // call to function defined in line 35
22
23       result2 = ComputeAverage(5, 4, 6);  // call to function defined in line 40
24
25       result3 = ComputeAverage(4.0, 5.0); // call to function defined in line 45
26
27       cout  <<  "\n\nresult1=\t"  <<  result1  <<  "\n\nresult2=\t"
31             <<   result2   <<   "\n\nresult3=\t"   <<   result3;
32       return 0;
33    }
34
```

```
35    int ComputeAverage(int value1, int value2)
36    {
37       return (value1 + value2) / 2;
38    }
39
40    int ComputeAverage(int value1, int value2, int value3)
41    {
42       return (value1 + value2 + value3) / 3;
43    }
44
45    double ComputeAverage(double value1, double value2)
46    {
47       return (value1 + value2) / 2;
48    }
```

OUTPUT

result1= 4

result2= 5

result3= 4.5

Static Local Variables

- **A static local variable** is a local variable of a function that keeps its value between function calls.
- A static local variable declaration statement is preceded by the keyword **static**.

Example

Assume given the following two functions:

void procedure1 ( )	void procedure2 ( )
{	{
static int svalue = 0;	**int nvalue = 0;**
cout << "\tsvalue =\t" << svalue;	cout << "\tnvalue =\t" << nvalue;
svalue ++;	nvalue ++;
}	}

The output of the following program follows:

```
int   main(   )
{
    for ( int ct = 0; ct < 5; ct ++)
    {
        cout  <<  endl  <<  "ct=\t"   <<  ct;
        procedure1(   );
        procedure2(   );
    }
}
```

OUTPUT

ct = 0 svalue = 0 nvalue = 0

ct = 1 svalue = 1 nvalue = 0

ct = 2 svalue = 2 nvalue = 0

ct = 3 svalue = 3 nvalue = 0

ct = 4 svalue = 4 nvalue = 0

Exercise F17

Given the following function definition:

```
void procedure (     )
{
   static  int  svalue = 20;
   cout  <<  "\tsvalue =\t"  <<  svalue;
   svalue  - = 2;
}
```

Show the output of the following program:

```
int   main(   )
{
    for ( int ct = 0; ct < 5; ct ++)
    {
        cout  <<  endl  <<  "ct=\t"   <<  ct;
        procedure(   );
    }
}
```

Structures and Functions

- **The return type of a function can be a structure.**

Example

Assume given the following structure:

```
struct  EmployeeInfo
{
    string  name;
    double payRate;
    int  hours;
    double grossPay;
};
```

Write a function to read the information about an employee, place it in a structure, and return it.

```
 EmployeeInfo  readInfo( void)
{
    EmployeeInfo  temp;
    cout  <<  endl  <<  "Enter the name, the pay rate and the number of hours:\t";
    cin  >>  temp.name  >>  temp.payRate  >>  temp.hours;
    return( temp );
}
```

- **A reference parameter of a function may have a structure as its data type.**

Example

Write a function *void computeGpay1(EmployeeInfo & emp)* that receives as argument a reference to an EmployeeInfo structure, computes, and sets its gross pay.

```
void  computeGpay1( EmployeeInfo  & emp )
{
    emp.grossPay  =  emp.payRate  *  emp.hours;
}
```

- **The data type of a value parameter of a function can be a structure.**

Example

Write a function *void printInfo(EmployeeInfo emp)* that receives as argument an EmployeeInfo structure and prints the values of its members.

```
void  printInfo( EmployeeInfo emp )
{
    cout    <<  "\nNAME:\t"    <<  emp.name
            <<  "\nPAY RATE:\t"  <<  emp.payRate
            <<  "\nHOURS:\t"    <<  emp.hours
            <<  "\nGROSS PAY:\t  <<  emp.grossPay;
}
```

Write a program segment to read the information about an employee, compute its gross pay, and print its name, pay rate, number of hours worked, and gross pay.

```
EmployeeInfo    emp;          // to hold the information about the employee

/*------------------------- read the information about the employee ----------------------------*/
emp   =   readInfo(  );

/*-------------------------compute the employee's gross pay ------------------------------------*/
computeGpay1( emp );

/*------------------------- print the information about the employee ----------------------------*/
printInfo( emp );
```

Exercise F18

Given the following structure:

```
struct  Demo
{
    int  first,
         second;
};
```

Do the following:

1.
   a. Write a function *readDemo( )* that reads the values for the member variables of a structure *Demo* and returns that structure.
   b. Write a code segment to define a *Demo* structure variable named *temp* and to read values into its members variables by calling function *readDemo( ).*
2.
   a. Write a function *addDemo( )* that receives as arguments two *Demo* structures as value parameters, and then builds and returns another *Demo* structure such that the values of its members variables are the sums of the values of the corresponding member variables of the structures received as arguments.

b. Write a code segment to create the *Demo* structure variable *demoR* such that the values of its member variables are the sums of the values of the corresponding member variables of the Demo structures *demo1* (with values 5 and 7 respectively) and *demo2* (with values 10 and 15 respectively). The sums of the values of the member variables of the structure variables are computed by calling the function *addDemo( )* defined above.

3.

a. Write a *void* function *updateDemo1( )* that has a *Demo* structure as a reference parameter. This function adds 10 to the value of the first member variable of the structure, and subtracts 5 from the value of its second member variable.
b. Write a code segment to define a *Demo* structure variable named *temp* with its member variables initialized with 6 and 7 respectively. It then adds 10 to the value of its first member variable and subtracts 5 from the value of its second member variable by calling function *updaDemo1( )*.

Programs with two or more Source Modules

- The functions that make up a C++ program may be stored in one or more files called source files.
- A global variable defined in one source file may be accessed in another source file only if it has been declared (but not defined) in that source.
- You declare (without defining) a global variable by preceding its declaration statement with the keyword **extern**.
- A new memory location is not created for a variable whose declaration statement is preceded with the keyword **extern**.
- The prototype of a function defined in another source module may also be preceded with the keyword **extern**, but this is not necessary.

```
/**************************  Source file progA.cpp *************************
   Program to process a salesman's total sales in one or more cities
   ***************************************************************************/
   #include   <iostream>
   using namespace std;

   double processSales(void);       //to read total sales and compute their sum
   double computeAvgeSale(double)   // to compute the average sale

   int cityCount;    // to hold the number of cities

   int main( )
   {
      double   totalSale;  // to hold the salesman's sum of all cities total sales

      /*--read the salesman total sales in all the cities and compute their sum--*/
      totalSale = processSales( );
      cout   <<   endl   <<   "You have made sales in: "   <<   cityCount   <<   "
                              cities";
      cout  <<  endl  <<  "The sum of all your total sales in all the cities is:\t"
            <<   totalSale;

      /*---------- compute the salesman's average total sale in one city ------- */
      cout  << endl  << "Your average total sale in one city is:\t"
            <<  computeAverageSale( totalSale );

      return 0;
   }
```

```
/****************************** Source file progB.cpp **********************/

   #include   <iostream>
   using namespace std;

   #define DUMMYSALE  -99

   extern int  cityCount; //declare the variable to hold the number of cities

   /*------------- definition of the function processSales( ) ------------------*/
   /*  read the salesman total sales in one or more cities and compute their sum */
   double processSales ( )
   {
      double citySale,  // to hold a city total sale
         sumSale  ;     // to hold the current total sales

      cityCount = 0;
      sumSale = 0;

      /*------- read the total sales in all the cities and compute their sum ---*/
      cin   >>   citySale;
      while( citySale != DUMMYSALE )
      {
         sumSale += citySale;
         cityCount++ ;
         cin   >>  citySale;
      }

      return( sumSale );
   }


   /*---------------definition of function ComputeAverageSale( ) ---------------*/
   /*  compute a salesman average total sale in a city
   */
   double computeAverageSale( double total )
   {
      return( total / cityCount );
   }
```

- When a program consists of two or more source modules, the function prototypes of the functions in a source module are in general stored in a header file that is included in every source module in which one or more of these functions are called.

```
   /************************ Header file progB.h ******************************/

    double processSales(void);          //to read total sales and compute their sum
   double computeAverageSale(double);  // to compute the average sale
```

```
/*************************  Source file progA.cpp ************************
   Program to process a salesman's total sales in one or more cities
   ************************************************************************/
   #include   <iostream>
   using namespace std;

   #include  "progB.h"

   int cityCount;    // to hold the number of cities

   int main( )
   {
      double   totalSale;  // to hold the salesman's sum of all cities total sales

      /*--read the salesman total sales in all the cities and compute their sum--*/
      totalSale = processSales( );
      cout   <<   endl   <<   "You have made sales in: "   <<   cityCount   <<   "
                              cities";
      cout  <<  endl  <<  "The sum of all your total sales in all the cities is:\t"
            <<   totalSale;

      /*---------- compute the salesman's average total sale in one city ------- */
      cout  << endl  << "Your average total sale in one city is:\t"
            <<  computeAverageSale( totalSale );

      return 0;
   }
```

Compiling, Linking, and Execution a Program with two or more source modules

- The following are the steps that you must follow to execute a program with two or more source modules:

  1. Compile each source module to create the corresponding object module: on our UNIX system, the command to compile the above source modules are:

     g++ -c progA.cpp the object file **progA.o** is created
     g++ -c progB.cpp the object file **progB.o** is created

  2. Use the **linker** to combine all the object modules and the library functions called in your program into an executable module: on our UNIX system, the command to combine the above object modules and the library functions that are called in them into the executable file **prog.bin** follows:

     g++ progA.o progB.o -o prog.bin

Function Templates

- A **function template** is an alternative way to overloading a function name when all the functions have the same number of parameters but with different data types.
- For example, **s**uppose that in a program we have to write three functions: The first interchanges the values of two character variables, the second one interchanges the values of two integer variables, and the last one interchanges the values of two double precision variables. Instead of writing the three separate functions, we can write a template function that can be used in any of the above three situations.
- C++ automatically generates separate **function template specializations** to handle each type of call appropriately.
- A **template function definition** begins with the keyword **template** followed by a **template parameter list** specified in angle brackets as follows:

template < ***class*** *T1,* ***class*** *T2, . . . ,* ***class*** *Tn* > or

template < ***typename*** *T1,* ***typename*** *T2, . . . ,* ***typename*** *Tn* >

Where: *T1, T2, . . . , Tn* are the **type parameters** that represent the different data types that will be used in the definition of the function template.

Example

Function template of functions to interchange the values of two variables

```
template <class T>
void   swapValues( T & var1 , T & var2)
{
    T   temp;
    temp  = var1;
    var1  = var2;
    var2 = temp;
}
```

Example

Function template that receives variables in two data types and output the maximum size of those variables/ data types.

Note:

C++ provides a *compile-time* unary operator named **sizeof** with the following syntax:

sizeof *<variable>* returns the number of bytes used to represent *<variable>*.

sizeof *(<variable>)* returns the number of bytes used to represent *<variable>*.

sizeof *<array-name>* returns the number of bytes in the array with name *<array-name>*.

sizeof *(<array-name>)* returns the number of bytes in the array with name *<array-name>*.

sizeof *(<data-type>)* returns the number of bytes used to represent *a variable* with the data type *<data-type>*.

Examples

```
int  inum, list[ 10 ];
double  dnum;
cout  <<  "\n size of inum is:\t"  <<  sizeof inum;
cout  <<  "\n size of int is:\t"  <<  sizeof (int);
cout  <<  "\n size of list is:\t"  <<  sizeof (list);
cout  <<  "\n size of dnum is:\t"  <<  sizeof dnum;
cout  <<  "\n size of double is:\t"  <<  sizeof (double);
```

Output

```
size of inum is:    4
size of int is:   4
size of list is:   40
size of dnum is:    8
size of double is:   8
```

Note that since the operator **sizeof** is a compiler-time operator, it cannot be used in a function to determine the size of an array passed to that function as an argument as follows:

```
void funct( int list[ ] )
{
    int size =  sizeof ( list )/ sizeof( int );

    .   .   .

}
```

```
template  <class T1, class T2>
int largest(  T1 &var1, T2 &var2 )
{
    int max;
    if ( sizeof( var1 ) > sizeof(var2) )
        max  = sizeof(var1);
    else
        max = sizeof(var2);
    return max;
}
```

Notes

- Functions templates do not have function prototypes: they must therefore be included in any source module in which they are called.
- One way to use functions templates in a program is to place them in a header file and to include that header file in any program where the function template is called.

Example

```
/*-- Program to read two values and to compute the difference of the largest value minus the smallest -*/
template <class T>
void   swapValues( T & var1 , T & var2)
{
    T   temp;
    temp  = var1;
    var1  = var2;
    var2 = temp;
}
int main(   )
{
    int inum1, inum2;
    double dnum1, dnum2;

    /*------ read two integer values and compute the difference of the largest minus the smallest -----*/
    cin   >>  inum1   >>   inum2;
    if(inum1 < inum2)
        swapValues( inum1, inum2);
    cout   <<   inum1 – inum2;

    /*- read two double precision values and compute the difference of the largest minus the smallest */
    cin   >>  dnum1   >>   dnum2;
    if(dnum1 < dnum2)
        swapValues( dnum1, dnum2);
    cout   <<   dnum1 – dnum2;
    return 0;
}
```

Exercise F19

Write the function template of functions to find the maximum of three values.