starting out with >>>

C++

From Control Structures through Objects

EIGHTH EDITION

TONY GADDIS

Chapter 6:

Functions

Addison-Wesley
is an imprint of

PEARSON

6.1

Modular Programming

Modular Programming

- <u>Modular programming</u>: breaking a program up into smaller, manageable functions or modules

- <u>Function</u>: a collection of statements to perform a task

- Motivation for modular programming:
  - Improves maintainability of programs
  - Simplifies the process of writing programs

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
    statement;
    statement;          main function
    statement;
}
```

```
void function2()
{
    statement;
    statement;          function 2
    statement;
}
```

```
void function3()
{
    statement;
    statement;          function 3
    statement;
}
```

```
void function4()
{
    statement;
    statement;          function 4
    statement;
}
```

6.2

Defining and Calling Functions
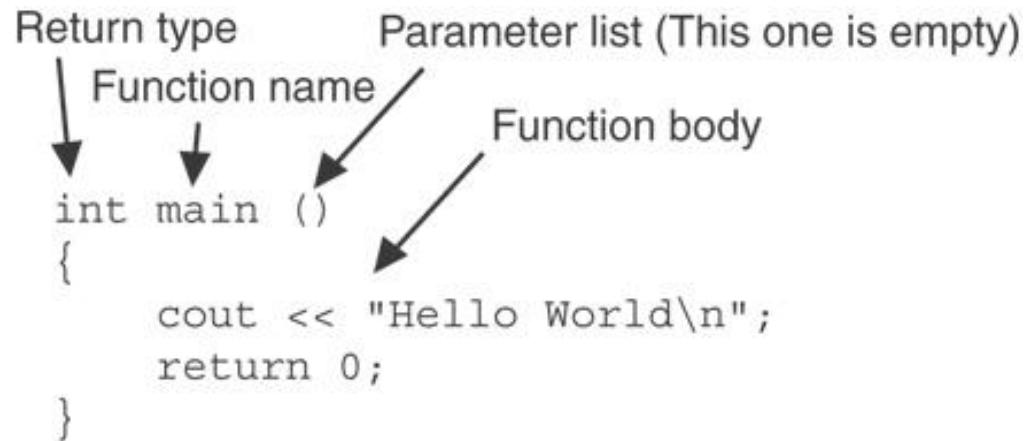
Defining and Calling Functions

- <u>Function call</u>: statement causes a function to execute

- <u>Function definition</u>: statements that make up a function

Function Definition

- Definition includes:
    - <u>return type:</u> data type of the value that function returns to the part of the program that called it
    - <u>name:</u> name of the function. Function names follow same rules as variables
    - <u>parameter list:</u> variables containing values passed to the function
    - <u>body:</u> statements that perform the function's task, enclosed in `{ }`

Function Definition


```
Return type          Parameter list (This one is empty)
    Function name                Function body

int main ()
{
    cout << "Hello World\n";
    return 0;
}
```

Note: The line that reads `int main()` is the *function header*.

Function Return Type

- If a function returns a value, the type of the value must be indicated:

  ```
  int main()
  ```

- If a function does not return a value, its return type is `void`:

  ```
  void printHeading()
  {
          cout << "Monthly Sales\n";
  }
  ```

Calling a Function

- To call a function, use the function name followed by `()` and `;`

  ```
  printHeading();
  ```

- When called, program executes the body of the called function

- After the function terminates, execution resumes in the calling function at point of call.
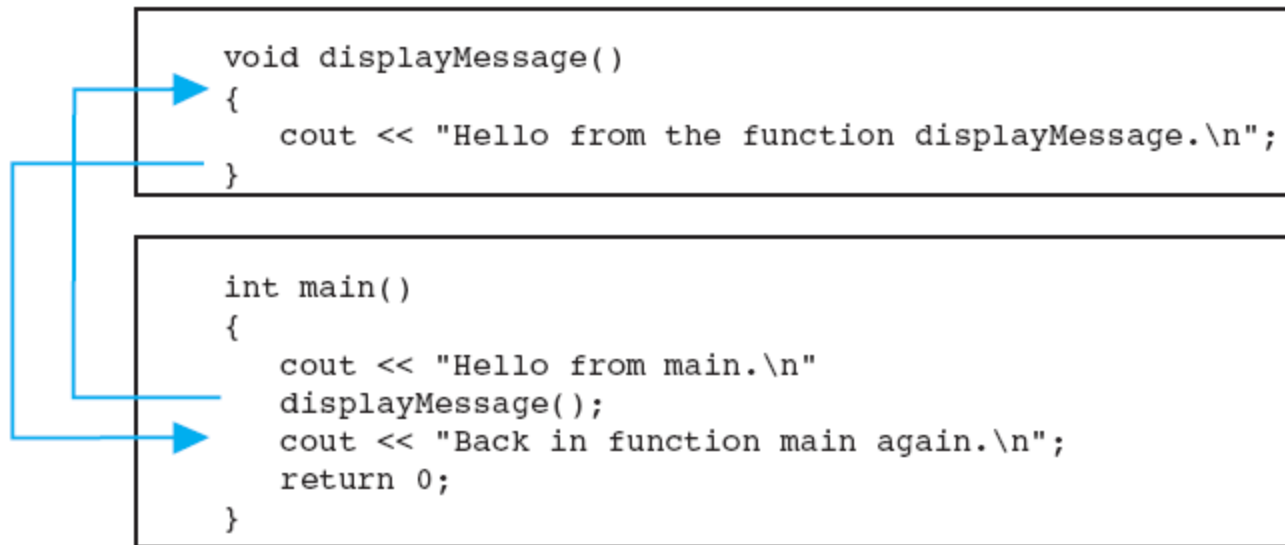
Functions in Program 6-1

Program 6-1

```cpp
 1  // This program has two functions: main and displayMessage
 2  #include <iostream>
 3  using namespace std;
 4
 5  //*****************************************
 6  // Definition of function displayMessage  *
 7  // This function displays a greeting.      *
 8  //*****************************************
 9
10  void displayMessage()
11  {
12     cout << "Hello from the function displayMessage.\n";
13  }
14
15  //*****************************************
16  // Function main                          *
17  //*****************************************
18
19  int main()
20  {
21     cout << "Hello from main.\n";
22     displayMessage();
23     cout << "Back in function main again.\n";
24     return 0;
25  }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```

Flow of Control in Program 6-1


```
void displayMessage()
{
    cout << "Hello from the function displayMessage.\n";
}
```

```
int main()
{
    cout << "Hello from main.\n"
    displayMessage();
    cout << "Back in function main again.\n";
    return 0;
}
```

Calling Functions

- `main` can call any number of functions
- Functions can call other functions
- Compiler must know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter

6.3

Function Prototypes

Function Prototypes

- Ways to notify the compiler about a function before a call to the function:

  - Place function definition before calling function's definition

  - Use a <u>function prototype</u> (<u>function declaration</u>) – like the function definition without the body
    - Header: `void printHeading()`
    - Prototype: `void printHeading();`

Function Prototypes in Program 6-5

Program 6-5

```
1   // This program has three functions: main, First, and Second.
2   #include <iostream>
3   using namespace std;
4
5   // Function Prototypes
6   void first();
7   void second();
8
9   int main()
10  {
11      cout << "I am starting in function main.\n";
12      first();    // Call function first
13      second();   // Call function second
14      cout << "Back in function main again.\n";
15      return 0;
16  }
17
```

(Program Continues)

Function Prototypes in Program 6-5

```
18    //**********************************
19    // Definition of function first.      *
20    // This function displays a message.  *
21    //**********************************
22
23    void first()
24    {
25       cout << "I am now inside the function first.\n";
26    }
27
28    //**********************************
29    // Definition of function second.     *
30    // This function displays a message.  *
31    //**********************************
32
33    void second()
34    {
35       cout << "I am now inside the function second.\n";
36    }
```

Prototype Notes

- Place prototypes near top of program

- Program must include either prototype or full function definition before any call to the function – compiler error otherwise

- When using prototypes, can place function definitions in any order in source file

6.4

Sending Data into a Function

Sending Data into a Function

- Can pass values into a function at time of call:

  ```
  c = pow(a, b);
  ```

- Values passed to function are <u>arguments</u>

- Variables in a function that hold the values passed as arguments are <u>parameters</u>

A Function with a Parameter Variable

```
void displayValue(int num)
{
    cout << "The value is " << num << endl;
}
```

The integer variable `num` is a parameter.
It accepts any integer value passed to the function.

Function with a Parameter in Program 6-6

Program 6-6

```cpp
1  // This program demonstrates a function with a parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototype
6  void displayValue(int);
7
8  int main()
9  {
10     cout << "I am passing 5 to displayValue.\n";
11     displayValue(5);  // Call displayValue with argument 5
12     cout << "Now I am back in main.\n";
13     return 0;
14  }
15
```

(Program Continues)

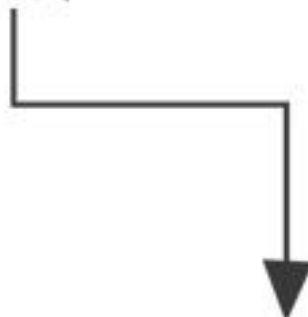# Function with a Parameter in Program 6-6

Program 6-6 *(continued)*

```
16    //*************************************************
17    // Definition of function displayValue.            *
18    // It uses an integer parameter whose value is displayed. *
19    //*************************************************
20
21    void displayValue(int num)
22    {
23        cout << "The value is " << num << endl;
24    }
```

Program Output

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```

Function with a Parameter in Program 6-6

```
displayValue(5);



void displayValue(int num)
{
  cout << "The value is " << num << endl;
}
```

The function call in line 11 passes the value 5

as an argument to the function.

Other Parameter Terminology

- A parameter can also be called a <u>formal parameter</u> or a <u>formal argument</u>
- An argument can also be called an <u>actual parameter</u> or an <u>actual argument</u>

Parameters, Prototypes, and Function Headers

- For each function argument,
  - the prototype must include the data type of each parameter inside its parentheses
  - the header must include a declaration for each parameter in its ()

```
void evenOrOdd(int);     //prototype
void evenOrOdd(int num)  //header
evenOrOdd(val);          //call
```

Function Call Notes

- Value of argument is copied into parameter when the function is called

- A parameter's scope is the function which uses it

- Function can have multiple parameters

- There must be a data type listed in the prototype `()` and an argument declaration in the function header `()` for each parameter

- Arguments will be promoted/demoted as necessary to match parameters

Passing Multiple Arguments

When calling a function and passing multiple arguments:

- the number of arguments in the call must match the prototype and definition

- the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

Passing Multiple Arguments in Program 6-8

Program 6-8

```
1    // This program demonstrates a function with three parameters.
2    #include <iostream>
3    using namespace std;
4
5    // Function Prototype
6    void showSum(int, int, int);
7
8    int main()
9    {
10       int value1, value2, value3;
11
12       // Get three integers.
13       cout << "Enter three integers and I will display ";
14       cout << "their sum: ";
15       cin >> value1 >> value2 >> value3;
16
17       // Call showSum passing three arguments.
18       showSum(value1, value2, value3);
19       return 0;
20   }
21
```
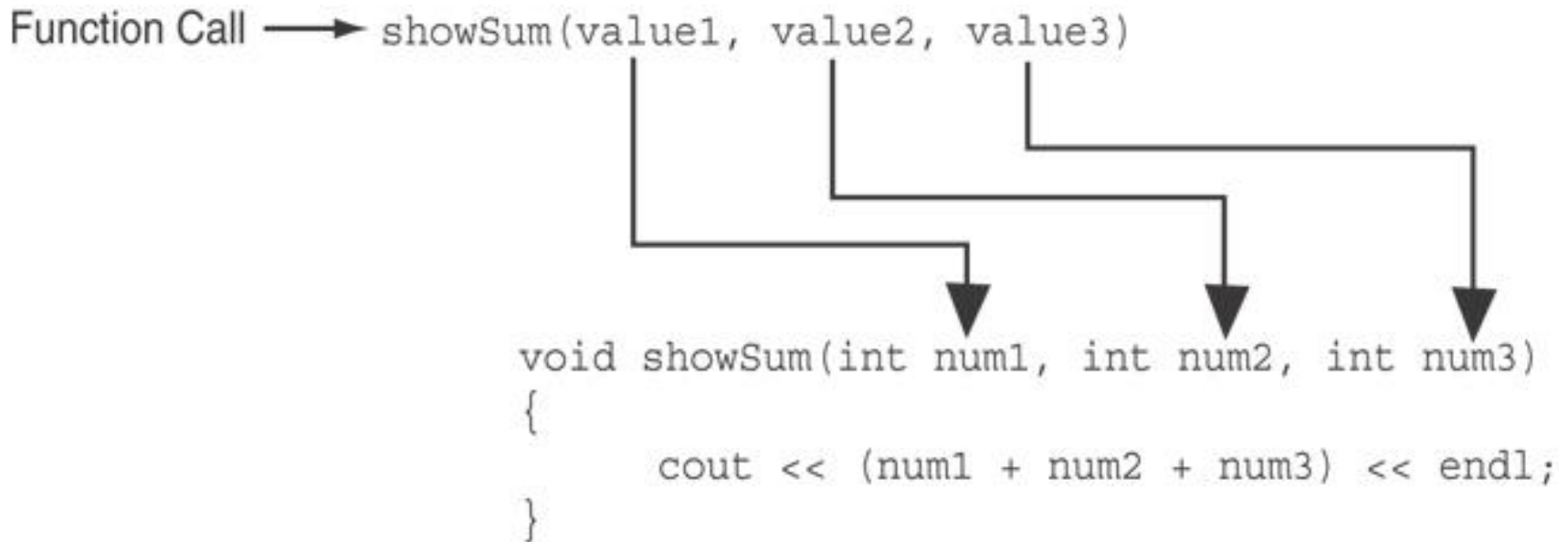
(Program Continues)

Passing Multiple Arguments in Program 6-8

```
22   //*********************************************************
23   // Definition of function showSum.                        *
24   // It uses three integer parameters. Their sum is displayed. *
25   //*********************************************************
26
27   void showSum(int num1, int num2, int num3)
28   {
29       cout << (num1 + num2 + num3) << endl;
30   }
```

Program Output with Example Input Shown in Bold

Enter three integers and I will display their sum: **4 8 7 [Enter]**
19

Passing Multiple Arguments in Program 6-8

Function Call ⟶ showSum(value1, value2, value3)

```
void showSum(int num1, int num2, int num3)
{
        cout << (num1 + num2 + num3) << endl;
}
```

The function call in line 18 passes value1, value2, and value3 as a arguments to the function.

6.5

Passing Data by Value

Passing Data by Value

- Pass by value: when an argument is passed to a function, its value is copied into the parameter.

- Changes to the parameter in the function do not affect the value of the argument

Passing Information to Parameters by Value

- Example: `int val=5;`

    `evenOrOdd(val);`

val

5

argument in
calling function

num

5

parameter in
`evenOrOdd` function

- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

6.6

Using Functions in Menu-Driven Programs

Using Functions in Menu-Driven Programs

- Functions can be used
  - to implement user choices from menu
  - to implement general-purpose tasks:
    - Higher-level functions can call general-purpose functions, minimizing the total number of functions and speeding program development time
- *See Program 6-10 in the book*

6.7

The `return` Statement

The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- In a `void` function without a `return` statement, the function ends at its last `}`

Performing Division in Program 6-11

Program 6-11

```cpp
1   // This program uses a function to perform division. If division
2   // by zero is detected, the function returns.
3   #include <iostream>
4   using namespace std;
5
6   // Function prototype.
7   void divide(double, double);
8
9   int main()
10  {
11      double num1, num2;
12
13      cout << "Enter two numbers and I will divide the first\n";
14      cout << "number by the second number: ";
15      cin >> num1 >> num2;
16      divide(num1, num2);
17      return 0;
18  }
```

(Program Continues)

Performing Division in Program 6-11

```
20   //**********************************************************
21   // Definition of function divide.                          *
22   // Uses two parameters: arg1 and arg2. The function divides arg1*
23   // by arg2 and shows the result. If arg2 is zero, however, the  *
24   // function returns.                                       *
25   //**********************************************************
26
27   void divide(double arg1, double arg2)
28   {
29      if (arg2 == 0.0)
30      {
31         cout << "Sorry, I cannot divide by zero.\n";
32         return;
33      }
34      cout << "The quotient is " << (arg1 / arg2) << endl;
35   }
```

Program Output with Example Input Shown in Bold
```
Enter two numbers and I will divide the first
number by the second number: 12 0 [Enter]
Sorry, I cannot divide by zero.
```

6.8

Returning a Value From a Function

Returning a Value From a Function

- A function can return a value back to the statement that called the function.
- You've already seen the `pow` function, which returns a value:

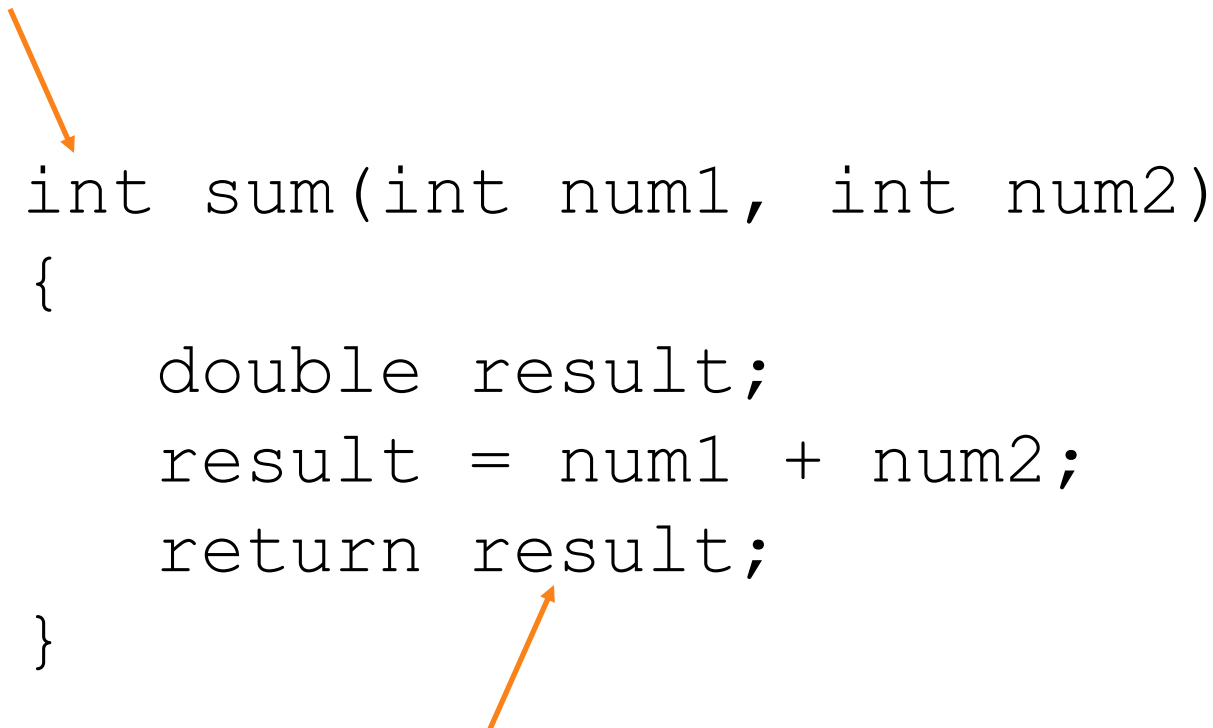```
double x;
x = pow(2.0, 10.0);
```

# Returning a Value From a Function

- In a value-returning function, the `return` statement can be used to return a value from function to the point of call. Example:

```cpp
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

A Value-Returning Function

Return Type

```
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

Value Being Returned

A Value-Returning Function

```
int sum(int num1, int num2)
{
    return num1 + num2;
}
```

Functions can return the values of expressions, such as `num1 + num2`

Function Returning a Value in Program 6-12

Program 6-12

```
1   // This program uses a function that returns a value.
2   #include <iostream>
3   using namespace std;
4
5   // Function prototype
6   int sum(int, int);
7
8   int main()
9   {
10      int value1 = 20,    // The first value
11          value2 = 40,    // The second value
12          total;          // To hold the total
13
14      // Call the sum function, passing the contents of
15      // value1 and value2 as arguments. Assign the return
16      // value to the total variable.
17      total = sum(value1, value2);
18
19      // Display the sum of the values.
20      cout << "The sum of " << value1 << " and "
21          << value2 << " is " << total << endl;
22      return 0;
23  }
```
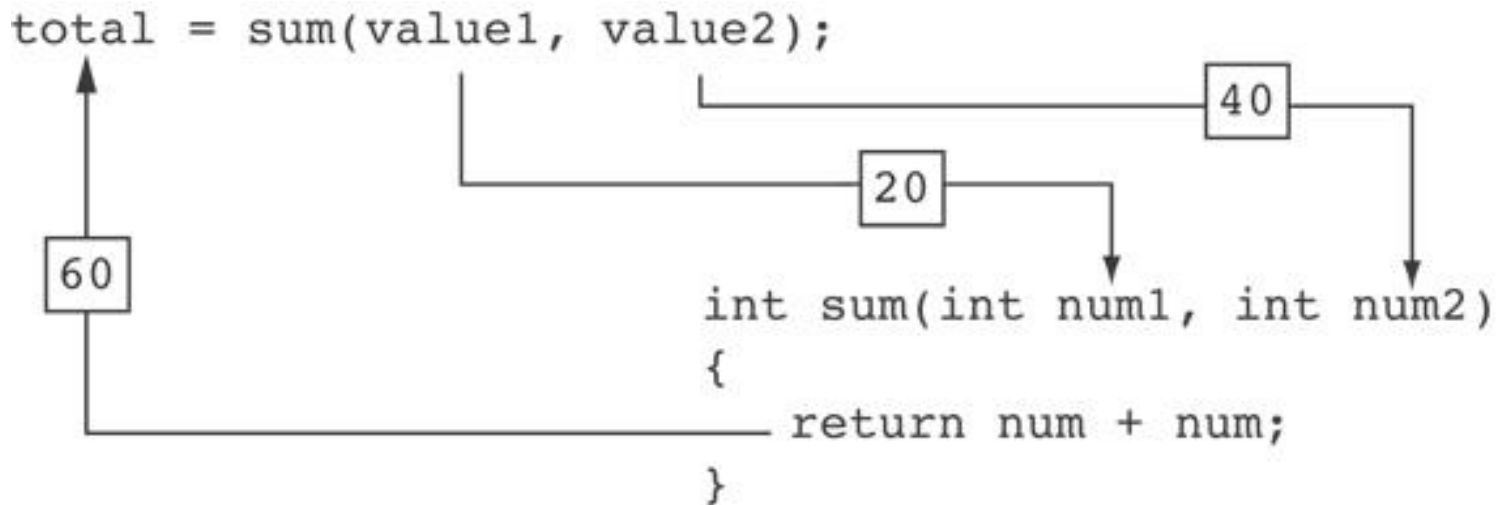
(Program Continues)

Function Returning a Value in Program 6-12

```
24
25   //******************************************************
26   // Definition of function sum. This function returns   *
27   // the sum of its two parameters.                      *
28   //******************************************************
29
30   int sum(int num1, int num2)
31   {
32      return num1 + num2;
33   }
```

Program Output
The sum of 20 and 40 is 60

Function Returning a Value in Program 6-12
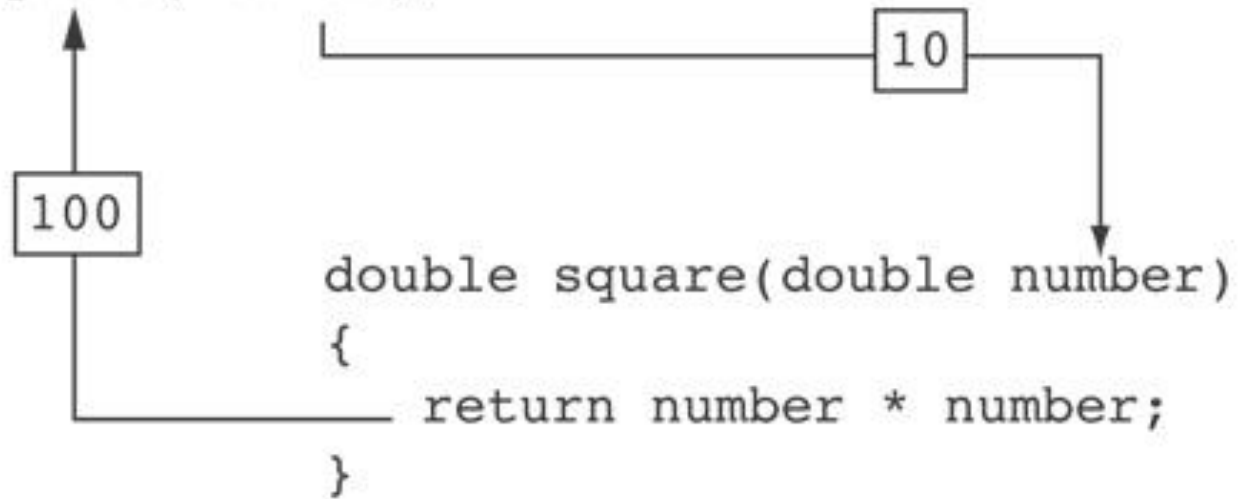


```
total = sum(value1, value2);
```

```
int sum(int num1, int num2)
{
    return num + num;
}
```

The statement in line 17 calls the sum function, passing `value1` and `value2` as arguments. The return value is assigned to the `total` variable.

Another Example from Program 6-13

```
area = PI * square(radius);
```

`100`

`10`

```
double square(double number)
{
    return number * number;
}
```

Returning a Value From a Function

- The prototype and the definition must indicate the data type of return value (not `void`)

- Calling function should use return value:
  - assign it to a variable
  - send it to `cout`
  - use it in an expression

6.9

Returning a Boolean Value

Addison-Wesley
is an imprint of

PEARSON

Returning a Boolean Value

- Function can return `true` or `false`

- Declare return type in function prototype and heading as `bool`

- Function body must contain `return` statement(s) that return `true` or `false`

- Calling function can use return value in a relational expression

Returning a Boolean Value in Program 6-15

Program 6-15

```
1   // This program uses a function that returns true or false.
2   #include <iostream>
3   using namespace std;
4
5   // Function prototype
6   bool isEven(int);
7
8   int main()
9   {
10      int val;
11
12      // Get a number from the user.
13      cout << "Enter an integer and I will tell you ";
14      cout << "if it is even or odd: ";
15      cin >> val;
16
17      // Indicate whether it is even or odd.
18      if (isEven(val))
19         cout << val << " is even.\n";
20      else
21         cout << val << " is odd.\n";
22      return 0;
23   }
24
```

(Program Continues)

Returning a Boolean Value in Program 6-15

```
25   //*******************************************************************
26   // Definition of function isEven. This function accepts an          *
27   // integer argument and tests it to be even or odd. The function    *
28   // returns true if the argument is even or false if the argument    *
29   // is odd. The return value is a bool.                              *
30   //*******************************************************************
31
32   bool isEven(int number)
33   {
34      bool status;
35
36      if (number % 2 == 0)
37         status = true;   // The number is even if there is no remainder.
38      else
39         status = false;  // Otherwise, the number is odd.
40      return status;
41   }
```

Program Output with Example Input Shown in Bold
```
Enter an integer and I will tell you if it is even or odd: 5 [Enter]
5 is odd.
```

6.10

Local and Global Variables

Local and Global Variables

- Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.

- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

Local Variables in Program 6-16
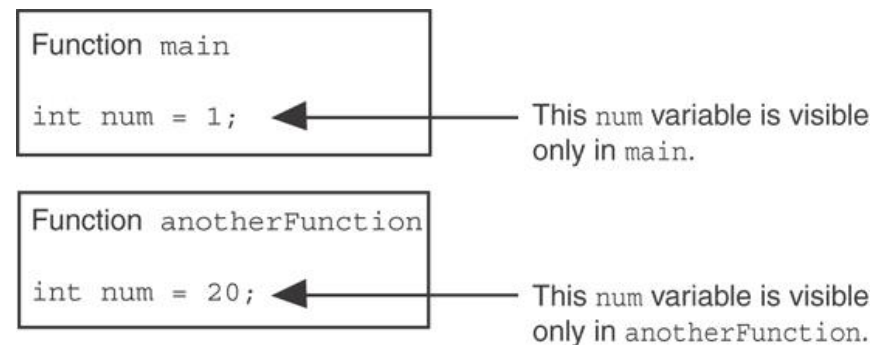
Program 6-16

```cpp
1   // This program shows that variables defined in a function
2   // are hidden from other functions.
3   #include <iostream>
4   using namespace std;
5
6   void anotherFunction(); // Function prototype
7
8   int main()
9   {
10      int num = 1;    // Local variable
11
12      cout << "In main, num is " << num << endl;
13      anotherFunction();
14      cout << "Back in main, num is " << num << endl;
15      return 0;
16   }
17
18   //*****************************************************
19   // Definition of anotherFunction                     *
20   // It has a local variable, num, whose initial value  *
21   // is displayed.                                      *
22   //*****************************************************
23
24   void anotherFunction()
25   {
26      int num = 20;   // Local variable
27
28      cout << "In anotherFunction, num is " << num << endl;
29   }
```

Local Variables in Program 6-16

Program Output
```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

When the program is executing in `main`, the `num` variable defined in `main` is visible. When `anotherFunction` is called, however, only variables defined inside it are visible, so the `num` variable in `main` is hidden.

Function `main`

`int num = 1;` ◄───── This `num` variable is visible only in `main`.

Function `anotherFunction`

`int num = 20;` ◄───── This `num` variable is visible only in `anotherFunction`.

Local Variable Lifetime

- A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable.

- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.

- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables and Global Constants

- A global variable is any variable defined outside all the functions in a program.

- The scope of a global variable is the portion of the program from the variable definition to the end.

- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.
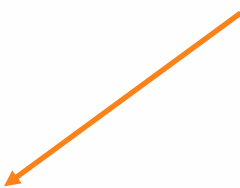
Global Variables and Global Constants

- You should avoid using global variables because they make programs difficult to debug.

- Any global that you create should be *global constants*.

Global Constants in Program 6-19

Program 6-19

```
1    // This program calculates gross pay.
2    #include <iostream>
3    #include <iomanip>
4    using namespace std;
5
6    // Global constants
7    const double PAY_RATE = 22.55;      // Hourly pay rate
8    const double BASE_HOURS = 40.0;     // Max non-overtime hours
9    const double OT_MULTIPLIER = 1.5;  // Overtime multiplier
10
11   // Function prototypes
12   double getBasePay(double);
13   double getOvertimePay(double);
14
15   int main()
16   {
17      double hours,              // Hours worked
18             basePay,            // Base pay
19             overtime = 0.0,    // Overtime pay
20             totalPay;           // Total pay
```

Global constants defined for values that do not change throughout the program's execution.

Addison-Wesley
is an imprint of

PEARSON

Global Constants in Program 6-19

The constants are then used for those values throughout the program.

```
29          // Get overtime pay, if any.
30          if (hours > BASE_HOURS)
31              overtime = getOvertimePay(hours);
```

```
56      // Determine base pay.
57      if (hoursWorked > BASE_HOURS)
58          basePay = BASE_HOURS * PAY_RATE;
59      else
60          basePay = hoursWorked * PAY_RATE;
```

```
75          // Determine overtime pay.
76          if (hoursWorked > BASE_HOURS)
77          {
78              overtimePay = (hoursWorked - BASE_HOURS) *
79                      PAY_RATE * OT_MULTIPLIER;
```

Initializing Local and Global Variables

- Local variables are not automatically initialized. They must be initialized by programmer.

- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

6.11

Static Local Variables

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.

- `static` local variables retain their contents between function calls.

- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Local Variables Do Not Retain Values Between Function calls in Program 6-21

Program 6-21

```cpp
1   // This program shows that local variables do not retain
2   // their values between function calls.
3   #include <iostream>
4   using namespace std;
5
6   // Function prototype
7   void showLocal();
8
9   int main()
10  {
11     showLocal();
12     showLocal();
13     return 0;
14  }
15
```

(Program Continues)

Local Variables Do Not Retain Values Between Function calls in Program 6-21

Program 6-21 (continued)

```
16   //**********************************************************
17   // Definition of function showLocal.                     *
18   // The initial value of localNum, which is 5, is displayed. *
19   // The value of localNum is then changed to 99 before the  *
20   // function returns.                                       *
21   //**********************************************************
22
23   void showLocal()
24   {
25       int localNum = 5; // Local variable
26
27       cout << "localNum is " << localNum << endl;
28       localNum = 99;
29   }
```

Program Output
```
localNum is 5
localNum is 5
```

In this program, each time `showLocal` is called, the `localNum` variable is re-created and initialized with the value 5.

A Different Approach, Using a Static Variable in Program 6-22

Program 6-22

```cpp
1   // This program uses a static local variable.
2   #include <iostream>
3   using namespace std;
4
5   void showStatic(); // Function prototype
6
7   int main()
8   {
9       // Call the showStatic function five times.
10      for (int count = 0; count < 5; count++)
11          showStatic();
12      return 0;
13  }
14
```

(Program Continues)

A Different Approach, Using a Static Variable in Program 6-22

Program 6-22 *(continued)*

```
15   //**********************************************************
16   // Definition of function showStatic.                      *
17   // statNum is a static local variable. Its value is displayed *
18   // and then incremented just before the function returns.   *
19   //**********************************************************
20
21   void showStatic()
22   {
23      static int statNum;
24
25      cout << "statNum is " << statNum << endl;
26      statNum++;
27   }
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

`statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

If you do initialize a local static variable, the initialization only happens once. See Program 6-23.

```
16    //*********************************************************
17    // Definition of function showStatic.                    *
18    // statNum is a static local variable. Its value is displayed *
19    // and then incremented just before the function returns.  *
20    //*********************************************************
21
22    void showStatic()
23    {
24        static int statNum = 5;
25
26        cout << "statNum is " << statNum << endl;
27        statNum++;
28    }
```

Program Output
```
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

6.12

Default Arguments

Default Arguments

A <u>Default argument</u> is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:
  ```
  void evenOrOdd(int = 0);
  ```
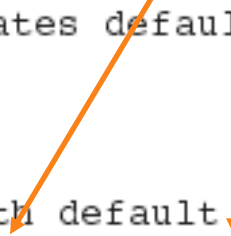- Can be declared in header if no prototype

- Multi-parameter functions may have default arguments for some or all of them:
  ```
  int getSum(int, int=0, int=0);
  ```

Default Arguments in Program 6-24

Default arguments specified in the prototype

Program 6-24

```
1    // This program demonstrates default function arguments.
2    #include <iostream>
3    using namespace std;
4
5    // Function prototype with default arguments
6    void displayStars(int = 10, int = 1);
7
8    int main()
9    {
10      displayStars();        // Use default values for cols and rows.
11      cout << endl;
12      displayStars(5);       // Use default value for rows.
13      cout << endl;
14      displayStars(7, 3);    // Use 7 for cols and 3 for rows.
15      return 0;
16   }
```

(Program Continues)

Default Arguments in Program 6-24

```
18   //******************************************************
19   // Definition of function displayStars.                *
20   // The default argument for cols is 10 and for rows is 1.*
21   // This function displays a square made of asterisks.   *
22   //******************************************************
23
24   void displayStars(int cols, int rows)
25   {
26      // Nested loop. The outer loop controls the rows
27      // and the inner loop controls the columns.
28      for (int down = 0; down < rows; down++)
29      {
30         for (int across = 0; across < cols; across++)
31            cout << "*";
32         cout << endl;
33      }
34   }
```

Program Output

```
**********

*****

*******
*******
*******
```

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```

6.13

Using Reference Variables as Parameters

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to 'return' more than one value

Passing by Reference

- A <u>reference variable</u> is an alias for another variable
- Defined with an ampersand (`&`)

  ```
  void getDimensions(int&, int&);
  ```
- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Passing a Variable By Reference in Program 6-25

Program 6-25

The & here in the prototype indicates that the parameter is a reference variable.
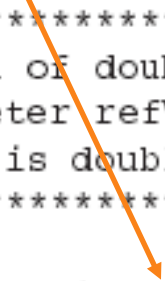
```
1   // This program uses a reference variable as a function
2   // parameter.
3   #include <iostream>
4   using namespace std;
5
6   // Function prototype. The parameter is a reference variable.
7   void doubleNum(int &);
8
9   int main()
10  {
11      int value = 4;
12
13      cout << "In main, value is " << value << endl;
14      cout << "Now calling doubleNum..." << endl;
15      doubleNum(value);
16      cout << "Now back in main. value is " << value << endl;
17      return 0;
18  }
19
```

Here we are passing value by reference.

(Program Continues)

Passing a Variable By Reference in Program 6-25

The & also appears here in the function header.

```
20    //*********************************************************
21    // Definition of doubleNum.                              *
22    // The parameter refVar is a reference variable. The value *
23    // in refVar is doubled.                                  *
24    //*********************************************************
25
26    void doubleNum (int &refVar)
27    {
28        refVar *= 2;
29    }
```

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main. value is 8
```

Reference Variable Notes

- Each reference parameter must contain &
- Space between type and & is unimportant
- Must use & in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

6.14

Overloading Functions

Overloading Functions

- <u>Overloaded functions</u> have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);                // 1
void getDimensions(int, int);           // 2
void getDimensions(int, double);        // 3
void getDimensions(double, double);     // 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);                  // 1
getDimensions(length, width);           // 2
getDimensions(length, height);          // 3
getDimensions(height, base);            // 4
```

Function Overloading in Program 6-27

Program 6-27

```
1   // This program uses overloaded functions.
2   #include <iostream>
3   #include <iomanip>
4   using namespace std;
5
6   // Function prototypes
7   int square(int);
8   double square(double);
9
10  int main()
11  {
12     int userInt;
13     double userFloat;
14
15     // Get an int and a double.
16     cout << fixed << showpoint << setprecision(2);
17     cout << "Enter an integer and a floating-point value: ";
18     cin >> userInt >> userFloat;
19
20     // Display their squares.
21     cout << "Here are their squares: ";
22     cout << square(userInt) << " and " << square(userFloat);
23     return 0;
24  }
```

The overloaded functions have different parameter lists

Passing a **double**

Passing an **int**

(Program Continues)

Function Overloading in Program 6-27

```
26    //**********************************************************
27    // Definition of overloaded function square.              *
28    // This function uses an int parameter, number. It returns the *
29    // square of number as an int.                            *
30    //**********************************************************
31
32    int square(int number)
33    {
34       return number * number;
35    }
36
37    //**********************************************************
38    // Definition of overloaded function square.              *
39    // This function uses a double parameter, number. It returns   *
40    // the square of number as a double.                      *
41    //**********************************************************
42
43    double square(double number)
44    {
45       return number * number;
46    }
```

Program Output with Example Input Shown in Bold
```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```

6.15

The `exit()` Function

The `exit()` Function

- Terminates the execution of a program
- Can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for abnormal termination of program
- Requires `cstdlib` header file

The `exit()` Function

- Example:

  ```
  exit(0);
  ```

- The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure:

  ```
  exit(EXIT_SUCCESS);
  exit(EXIT_FAILURE);
  ```

6.16

Stubs and Drivers

Stubs and Drivers

- Useful for testing and debugging program and function logic and design
- <u>Stub</u>: A dummy function used in place of an actual function
  - Usually displays a message indicating it was called. May also display parameters
- <u>Driver</u>: A function that tests another function by calling it
  - Various arguments are passed and return values are tested