Chapter 6: Modularity Using Functions
Objectives

In this chapter, you will learn about:

- Function and parameter declarations
- Returning a single value
- Returning multiple values
- Rectangular to polar coordinate conversion
- Variable scope
- Variable storage categories
- Common programming errors
Function and Parameter Declarations

• Interaction with a function includes:
  – Passing data to a function correctly when its called
  – Returning values from a function when it ceases operation
• A function is called by giving the function’s name and passing arguments in the parentheses following the function name

\[
\text{function-name} (\text{data passed to function});
\]

This identifies the called function
This passes data to the function

Figure 6.1 Calling and passing data to a function
Function and Parameter Declarations (continued)

• Before a function is called, it must be declared to function that will do calling

• Declaration statement for a function is referred to as function prototype

• Function prototype tells calling function:
  – Type of value that will be formally returned, if any
  – Data type and order of the values the calling function should transmit to the called function

• Function prototypes can be placed with the variable declaration statements above the calling function name or in a separate header file
Calling a Function

- Requirements when calling a function include:
  - Using the name of the function
  - Enclosing any data passed to the function in the parentheses following the function name, using the same order and type declared in the function prototype
The items enclosed in the parentheses are called **arguments** of the called function.

```
findMax(firstnum, secnum);
```

This identifies the `findMax()` function to be passed two values.

**Figure 6.2** Calling and passing two values to `findMax()`
Calling a Function (continued)

Figure 6.3 The \texttt{findMax()} function receives actual values.
Defining a Function

- Every C++ function consists of two parts:
  - Function header
  - Function body
- Function header’s purpose:
  - Identify data type of value function returns, provide function with name, and specify number, order, and type of arguments function expects
- Function body’s purpose:
  - To operate on passed data and return, at most, one value directly back to the calling function
Figure 6.4 The general format of a function

function header line

{ constant and variable declarations; any other C++ statements; }

Function header

Function body

findMax(firstnum, secnum);

This statement calls findMax()

The value in firstnum is passed

The value in secnum is passed

findMax(int x, int y)

The parameter named x

The parameter named y

Figure 6.5 Storing values in parameters
Placement of Statements

• General rule for placing statements in a C++ program:
  – All preprocessor directives, named constants, variables, and functions must be declared or defined before they can be used
  – Although this rule permits placing both preprocessor directives and declaration statements throughout the program, doing so results in poor program structure
Although useful functions having an empty parameter list are extremely limited, they can occur.

Function prototype for such a function requires writing the keyword `void` or nothing at all between the parentheses following the function’s name.

Examples:

```cpp
int display();
int display(void);
```
**Default Arguments**

- C++ provides **default arguments** in a function call for added flexibility
  - Primary use: to extend parameter list of existing functions without requiring any change in calling parameter lists already used in a program
  - Listed in the function prototype and transmitted automatically to the called function when the corresponding arguments are omitted from the function call
  - Example: Function prototype with default arguments
    ```
    void example(int, int = 5, double = 6.78)
    ```
Reusing Function Names (Overloading)

• C++ provides the capability of using the same function name for more than one function
  – Referred to as function overloading

• Only requirement for creating more than one function with same name:
  – Compiler must be able to determine which function to use based on the parameters’ data types (not the data type of the return value, if any)
  – Which of the functions is called depends on the argument type supplied at the time of the call
Function Templates

• Function template: Single complete function that serves as a model for a family of functions
  – Function from the family that is actually created depends on the specific function call
• Generalize the writing of functions that perform essentially the same operation, but on different parameter data types
• Make it possible to write a general function that handles all cases but where the compiler can set parameters, variables, and even return type based on the actual function call
Returning a Single Value

- Function receiving an argument passed by value cannot inadvertently alter value stored in the variable used for the argument.
- Function receiving passed by value arguments can process the values sent to it in any fashion and return one, and only one, “legitimate” value directly to the calling function.

Figure 6.10 A function directly returns at most one value.
Inline Functions

• Calling a function places a certain amount of overhead on a computer
  – Placing argument values in a reserved memory region (called the \textbf{stack}) that the function has access to
  – Passing control to the function
  – Providing a reserved memory location for any returned value (again, using the stack for this purpose)
  – Returning to the correct point in the calling program
Paying overhead associated with calling a function is justified when a function is called many times
  - Can reduce a program’s size substantially
For small functions that are not called many times, overhead of passing and returning values might not be warranted

**Inline functions:**
  - Group repeating lines of code together under a common function name
  - Have the compiler place this code in the program wherever the function is called
• Advantage: Increase in execution speed
  – Because the inline function is expanded and included in every expression or statement calling it, no execution time is lost because of the call and return overhead a non-inline function requires
• Each time an inline function is referenced the complete code is reproduced and stored as an integral part of the program
• A non-inline function is stored in memory only once
• Inline functions should be used only for small functions that aren’t called extensively in a program
Returning Multiple Values

- In a typical function invocation, called function receives values from its calling function, stores and manipulates the passed values, and directly returns at most one value
  - **Pass by value**: When data is passed in this manner
  - **Pass by reference**: Giving a called function direct access to its calling function’s variables is referred to as
    - The called function can reference, or access, the variable whose address has been passed as a pass by reference argument
Passing and Using Reference Parameters

• From the sending side, calling a function and passing an address as an argument that’s accepted as a reference parameter is the same as calling a function and passing a value.

• Whether a value or an address is actually passed depends on the parameter types declared.
Passing and Using Reference Parameters (continued)

**Figure 6.11** The equivalence of arguments and parameters in Program 6.8
Passing and Using Reference Parameters (continued)

Figure 6.12  The relationship between argument and parameter names
A Case Study: Rectangular to Polar Coordinate Conversion

Figure 6.14 The problem-solver algorithm
Figure 6.15 The correspondence between polar (distance and angle) and Cartesian (x and y) coordinates
A Case Study: Rectangular to Polar Coordinate Conversion (cont’d)

Figure 6.16 A top level structure diagram
Figure 6.17  Parameter values when `polar()` is called
A function can be thought of as a closed box, with slots at the top to receive values and a single slot at the bottom to return a value.

**Figure 6.19** A function can be considered a closed box.
Variable Scope (continued)

- **Local variables**: Variables created in a function that are conventionally available only to the function.
- **Scope**: Section of the program where the identifier is valid or “known”.
- A variable with **local scope** is simply one with storage locations set aside for it by a declaration statement inside the function that declared them.
- A variable with **global scope** has storage created for it by a declaration statement located outside any function.
Variable Scope (continued)

Figure 6.20 The three storage areas reserved by Program 6.15
Variable Scope (continued)

Figure 6.21 Relating the scope and type of a variable
Scope Resolution Operator

- When a local variable has the same name as a global variable, all references to the variable name made within the local variable’s scope refer to the local variable
- See Program 6.16
Scope Resolution Operator (continued)

• To reference a global variable when a local variable of the same name is in scope, use C++’s scope resolution operator, which is ::

• See Program 6.16.a
Misuse of Globals

- Global variables allow programmers to “jump around” the normal safeguards provided by functions.
- Instead of passing variables to a function, it is possible to make all variables global: *do not do this*
  - Indiscriminate use of global variables destroys the safeguards C++ provides to make functions independent and insulated from each other.
  - Using only global variables can be especially disastrous in large programs with many user-created functions.
Variable Storage Categories

• A variable’s scope can be thought of as the space in the program where the variable is valid

• In addition to space dimension represented by scope, variables have a time dimension that refers to the length of time storage locations are reserved for a variable

• This time, dimension is referred to as the variable’s lifetime

• When and how long a variable’s storage locations are kept before they are released can be determined by the variable’s storage category
Variable Storage Categories (continued)

- The four available storage categories are:
  - auto
  - static
  - extern
  - register

```c++
auto int num;    // auto storage category and int data type
static int miles; // static storage category and int data type
register int dist; // register storage category and int data type
extern int volts; // extern storage category and int data type
auto float coupon; // auto storage category and float data type
static double yrs; // static storage category and double data type
extern float yld; // extern storage category and float data type
auto char inKey;  // auto storage category and char variable type
```
Local Variable Storage Categories

- Local variables can be members only of the `auto`, `static`, or `register` storage categories.

- Storage for automatic local variables is reserved or created automatically each time a function:
  - As long as the function hasn’t returned control to its calling function, all automatic variables local to the function are “alive”.

- A local `static` variable isn’t created and destroyed each time the function declaring it is called:
  - Local `static` variables remain in existence for the program’s lifetime.
Most computers have a few high-speed storage areas, called **registers**, located in the CPU that can also be used for variable storage.

- Because registers are located in the CPU, they can be accessed faster than normal memory storage areas located in the computer’s memory unit.
Global Variable Storage Categories

• Global variables are created by definition statements external to a function
• By their nature, global variables do not come and go with the calling of a function
• After a global variable is created, it exists until the program in which it’s declared has finished executing
• Global variables can be declared with the static or extern storage category, but not both
Common Programming Errors

• Passing incorrect data types
• Errors that occur when the same variable is declared locally in both the calling and the called functions
• Omitting the called function’s prototype before or within the calling function
• Terminating a function header with a semicolon
• Forgetting to include the data type of a function’s parameters in the function header
Summary

• A function is called by giving its name and passing any data to it in the parentheses following the name
• A function’s return type is the data type of the value the function returns
• Arguments passed to a function when it is called must conform to the parameters specified by the function header
• Functions can be declared to all calling functions by means of a function prototype
• Every variable has a storage category, which determines how long the variable’s value is retained