

THIRD EDITION

A First Book of C++

From Here to There

CHAPTER 6

Modularity Using Functions

Objectives

You should be able to describe:

- Function and Parameter Declarations
- Returning a Single Value
- Pass by Reference
- Variable Scope
- Variable Storage Class
- Common Programming Errors

Function and Parameter Declarations

- All C++ programs must contain a `main()` function
 - May also contain unlimited additional functions
- Major programming concerns when creating functions:
 - How does a function interact with other functions (including `main`)?
 - Correctly passing data to function
 - Correctly returning values from a function

Function and Parameter Declarations (continued)

- Function call process:
 - Give function name
 - Pass data to function as arguments in parentheses following function name
- Only after called function successfully receives data passed to it can the data be manipulated within the function

Function and Parameter Declarations (continued)

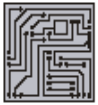
FIGURE 6.1 *Calling and Passing Data to a Function*

```
functionName (data passed to function);
```

This identifies the called function

This passes data to the function

Function and Parameter Declarations (continued)



Program 6.1

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

void findMax(int, int); // the function declaration (prototype)

int main()
{
    int firstnum, secnum;

    cout << "\nEnter a number: ";
    cin >> firstnum;
    cout << "Great! Please enter a second number: ";
    cin >> secnum;

    findMax(firstnum, secnum); // the function is called here

    return 0;
}
```

Function and Parameter Declarations (continued)

- Program 6.1 not complete
 - `findMax()` must be written and added
 - Done in slide 15
- Complete program components:
 - `main()`: referred to as **calling** program
 - `findMax()`: referred to as **called** program
- Complete program can be compiled and executed

Function Prototypes

- **Function Prototype:** declaration statement for a function
 - Before a function can be called, it must be declared to the calling function
 - Tells the calling function:
 - The type of value to be returned, if any
 - The data type and order of values transmitted to the called function by the calling function

Function Prototypes (continued)

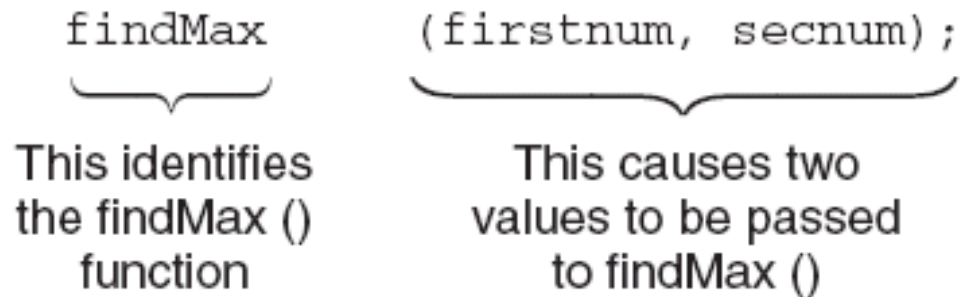
- **Example:** the function prototype in Program 6.1

```
void findMax(int, int);
```

 - Declares that `findMax()` expects two integer values sent to it
 - `findMax()` returns no value (`void`)
- **Prototype statement placement options:**
 - Together with variable declaration statements just above calling function name (as in Program 6.1)
 - In a separate header file to be included using a `#include` preprocessor statement

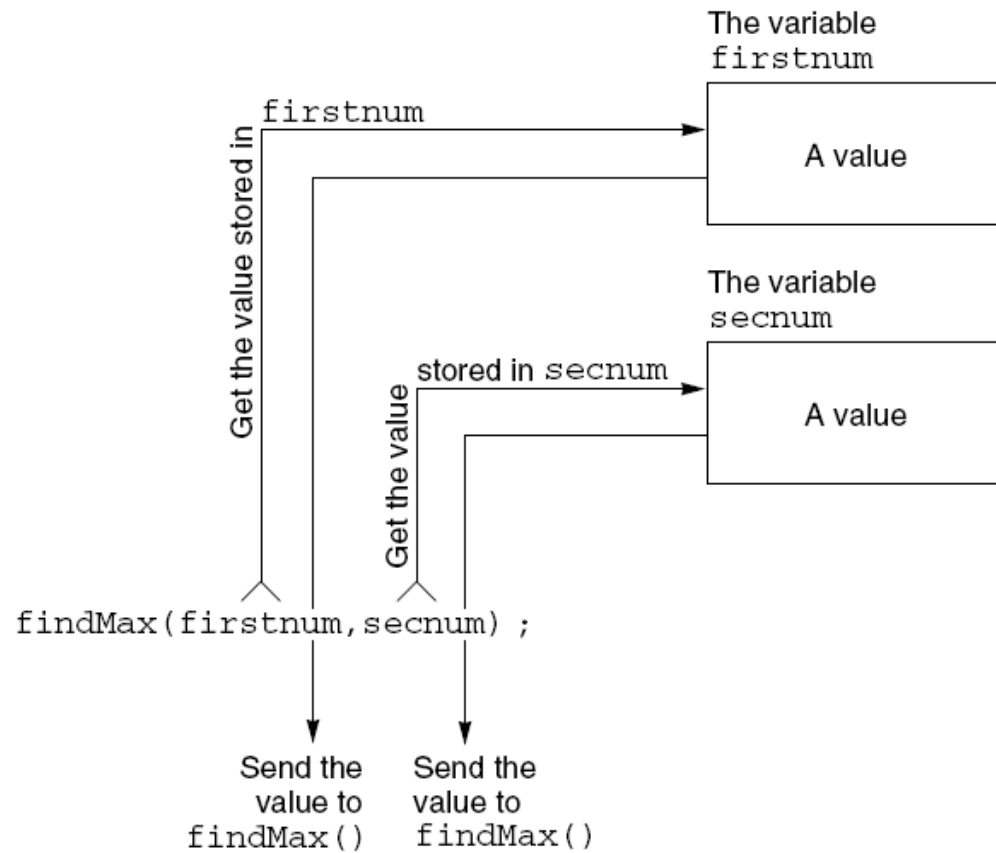
Calling a Function

FIGURE 6.2 *Calling and Passing Two Values to findMax ()*



Calling a Function (continued)

FIGURE 6.3 `findMax()` Receives Actual Values



Defining a Function

- A function is defined when it is written
 - Can then be used by any other function that suitably declares it
- **Format:** two parts
 - **Function header** identifies:
 - Data type returned by the function
 - Function name
 - Number, order and type of arguments expected by the function
 - **Function body:** statements that operate on data
 - Returns one value back to the calling function

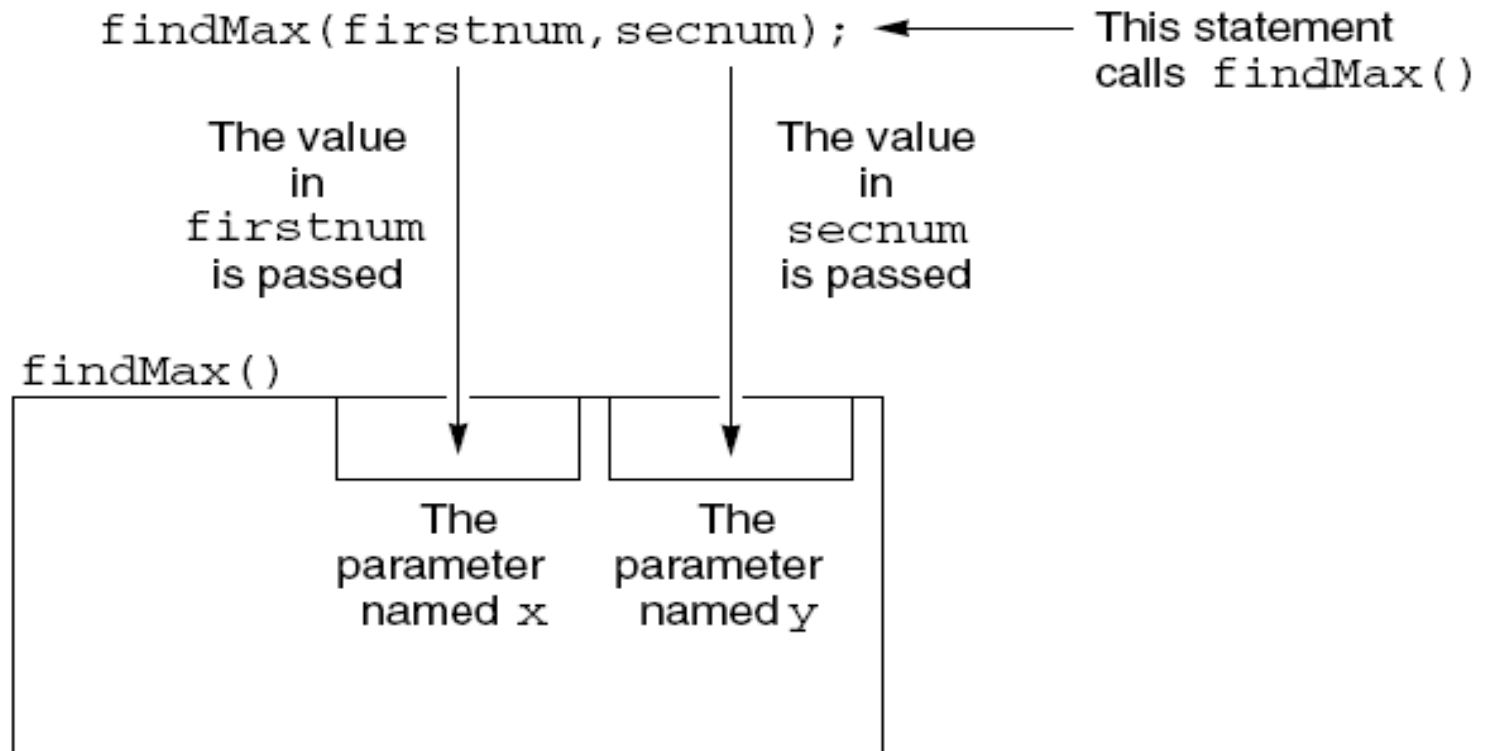
Defining a Function (continued)

FIGURE 6.4 *General Format of a Function*

```
function header line ← Function header
{
    C++ statements;      } Function body
}
```

Defining a Function (continued)

FIGURE 6.5 *Storing Values into Parameters*



Defining a Function (continued)

`findMax()` function definition (from program 6.1)

```
void findMax (int x, int y)
{
    // start of function body
    int maxnum;           // variable declaration
    if (x >= y)           // find the maximum number
        maxnum = x;
    else
        maxnum = y;
    cout << "\nThe maximum of the two numbers is "
<<maxnum<< endl;
} // end of function body and end of function
```

Defining a Function (continued)

- Order of functions in a program:
 - Any order is allowed
 - `main()` usually first
 - `main()` is the driver function
 - Gives reader overall program concept before details of each function encountered
- Each function defined outside any other function
 - Each function separate and independent
 - No nesting of function definitions allowed

Placement of Statements

- **Requirement:** items that must be either declared or defined before they are used:
 - Preprocessor directives
 - Named constants
 - Variables
 - Functions
- Otherwise, C++ is flexible in requirements for ordering of statements

Placement of Statements (continued)

- Recommended ordering of statements
 - Good programming practice

```
preprocessor directives
function prototypes
int main()
{
    symbolic constants
    variable declarations
    other executable statements
    return value
}
function definitions
```

Function Stubs

- Possible programming approach:
 - Write `main()` first and add functions as developed
 - Program cannot be run until all functions are included
- Stub: beginning of a final function
 - Can be used as a placeholder for a function until the function is completed
 - A “fake” function that accepts parameters and returns values in proper form
 - Allows `main` to be compiled and tested before all functions are completed

Functions with Empty Parameter Lists

- Extremely limited use
- Prototype format:

```
int display ();  
Int display (void);
```
- Information provided in above prototypes:
 - display takes no parameters
 - display returns an integer

Default Arguments

- Values listed in function prototype
 - Automatically transmitted to the called function when the arguments omitted from function call

- **Example:**

```
void example (int, int = 5, double = 6.78);
```

- Provides default values for last two arguments
- Following function calls are valid:

```
example(7, 2, 9.3) // no defaults used
```

```
example(7, 2) // same as example(7, 2, 6.78)
```

```
example(7) // same as example(7, 5, 6.78)
```

Reusing Function Names - Overloading

- **Function overloading:** using same function name for more than one function
 - Compiler must be able to determine which function to use based on data types of parameters (not data type of return value)
- Each function must be written separately
 - Each acts as a separate entity
- Use of same name does not require code to be similar
 - **Good programming practice:** functions with the same name perform similar operations

Reusing Function Names - Overloading (continued)

Example: two functions named `cdabs()`

```
void cdabs(int x) // compute and display the absolute
                //value of an integer
{
    if ( x < 0 )
        x = -x;
    cout << "The absolute value of the integer is " << x <<
    endl;
}
void cdabs(float x) // compute and display the
                   //absolute value of a float
{
    if ( x < 0 )
        x = -x;
    cout << "The absolute value of the float is " << x <<
    endl;
}
```

Reusing Function Names - Overloading (continued)

- **Function call:** `cdabs(10);`
 - Causes compiler to use the function named `cdabs()` that expects an integer argument
- **Function call:** `cdabs(6.28f);`
 - Causes compiler to use the function named `cdabs()` that expects a double-precision argument
- Major use of overloaded functions
 - Constructor functions

Function Templates

- Most high-level languages require each function to have its own name
 - Can lead to a profusion of names
- **Example:** functions to find the absolute value
 - Three separate functions and prototypes required

```
void abs (int);  
void fabs (float);  
void dabs (double);
```
- Each function performs the same operation
 - Only difference is data type handled

Function Templates (continued)

- Example of function template:

```
template <class T>
void showabs(T number)
{
    if (number < 0)
        number = -number;
    cout << "The absolute value of the number "
         << " is " << number << endl;
    return;
}
```

- Template allows for one function instead of three
 - T represents a general data type
 - T replaced by an actual data type when compiler encounters a function call

Function Templates (continued)

- Example (continued):

```
int main()
{
    int num1 = -4;
    float num2 = -4.23F;
    double num3 = -4.23456;
    showabs(num1);
    showabs(num2);
    showabs(num3);
    return 0;
}
```

- Output from above program:

```
The absolute value of the number is 4
The absolute value of the number is 4.23
The absolute value of the number is 4.23456
```

Returning a Single Value

- Passing data to a function:
 - Called function receives only a copy of data sent to it
 - Protects against unintended change
 - Passed arguments called **pass by value** arguments
 - A function can receive many values (arguments) from the calling function

Returning a Single Value (continued)

- **Returning data from a function**
 - Only one value directly returned from function
 - Called function header indicates type of data returned

- **Examples:**

```
void findMax(int x, int y)
```

- `findMax` accepts two integer parameters and returns no value

```
int findMax (float x, float y)
```

- `findMax` accepts two float values and returns an integer value

Inline Functions

- **Calling functions:** associated overhead
 - Placing arguments in reserved memory (stack)
 - Passing control to the function
 - Providing stack space for any returned value
 - Returning to proper point in calling program
- Overhead justified when function is called many times
 - Better than repeating code

Inline Functions (continued)

- Overhead not justified for small functions that are not called frequently
 - Still convenient to group repeating lines of code into a common function name
- **In-line function:** avoids overhead problems
 - C++ compiler instructed to place a copy of in-line function code into the program wherever the function is called

Inline Functions (continued)



Program 6.7

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

inline double tempvert(double inTemp) // an inline function
{
    return( (5.0/9.0) * (inTemp - 32.0) );
}

int main()
{
    const CONVERTS = 4; // number of conversions to be made
    int count; // start of variable declarations
    double fahren;

    for(count = 1; count <= CONVERTS; count++)
    {
        cout << "\nEnter a Fahrenheit temperature: ";
        cin >> fahren;
        cout << "The Celsius equivalent is "
             << tempvert(fahren) << endl;
    }

    return 0;
}
```

Pass by Reference

- Called function usually receives values as pass by value
 - Only copies of values in arguments are provided
- Sometimes desirable to allow function to have direct access to variables
 - Address of variable must be passed to function
 - Function can directly access and change the value stored there
- **Pass by reference:** passing addresses of variables received from calling function

Passing and Using Reference Parameters

- **Reference parameter:** receives the address of an argument passed to called function
- **Example:** accept two addresses in function `newval()`
- **Function header:**

```
void newval (double& num1, double& num2)
```

 - Ampersand, `&`, means “the address of”
- **Function Prototype:**

```
void newval (double&, double&);
```

Variable Scope

- **Scope**: section of program where identifier is valid (known or visible)
- **Local variables** (local scope): variables created inside a function or program component
 - Meaningful only when used in expressions inside the function in which it was declared
- **Global variables** (global scope): variables created outside any function
 - Can be used by all functions physically placed after global variable declaration

Scope Resolution Operator

- Local variable with the same name as a global variable
 - All references to variable name within scope of local variable refer to the local variable
 - Local variable name takes precedence over global variable name
- Scope resolution operator (`::`)
 - When used before a variable name the compiler is instructed to use the global variable
 - `::number` // scope resolution operator
 - // causes global variable to be used

Misuse of Globals

- Avoid overuse of globals
 - Too many globals eliminates safeguards provided by C++ to make functions independent
 - Misuse does not apply to function prototypes
 - Prototypes are typically global
- Difficult to track down errors in a large program using globals
 - Global variable can be accessed and changed by any function following the global declaration

Variable Storage Class

- Scope has a space and a time dimension
- **Time dimension** (lifetime): length of time that storage locations are reserved for a variable
 - All variable storage locations released back to operating system when program finishes its run
 - During program execution interim storage locations are reserved
 - **Storage class**: determines length of time that interim locations are reserved
 - **Four classes**: `auto`, `static`, `extern`, `register`

Local Variable Storage Classes

- Local variable can only be members of `auto`, `static` or `register` class
- **auto class**: default, if no class description included in variable's declaration statement
- Storage for `auto` local variables automatically reserved (created)
 - Each time a function declaring `auto` variables is called
 - Local `auto` variables are “alive” until function returns control to calling function

Local Variable Storage Classes (continued)

- **static storage class:** allows a function to remember local variable values between calls
 - `static` local variable lifetime = lifetime of program
 - Value stored in variable when function is finished is available to function next time it is called
- Initialization of `static` variables (local and global)
 - Done one time only, when program first compiled
 - Only constants or constant expressions allowed

Local Variable Storage Classes (continued)



Program 6.14

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

void teststat();    // function prototype

int main()
{
    int count;          // count is a local auto variable
    for(count = 1; count <= 3; count++)
        teststat();

    return 0;
}

void teststat()
{
    static int num = 0; // num is a local static variable
    cout << "The value of the static variable num is now "
         << num << endl;
    num++;

    return;
}
```

Local Variable Storage Classes (continued)

- **register Storage class:** same as `auto` class except for location of storage for class variables
 - Uses high-speed registers
 - Can be accessed faster than normal memory areas
 - Improves program execution time
- Some computers do not support `register` class
 - Variables automatically switched to `auto` class

Global Variable Storage Classes

- **Global variables:** created by definition statements external to a function
 - Do not come and go with the calling of a function
 - Once created, a global variable is alive until the program in which it is declared finishes executing
 - May be declared as members of `static` or `extern` classes
- **Purpose:** to extend the scope of a global variable beyond its normal boundaries

Common Programming Errors

- Passing incorrect data types between functions
 - Values passed must correspond to data types declared for function parameters
- Declaring same variable name in calling and called functions
 - A change to one local variable does not change value in the other
- Assigning same name to a local and a global variable
 - Use of a variable's name only affects local variable's contents unless the `::` operator is used

Common Programming Errors (continued)

- Omitting a called function's prototype
 - The calling function must be alerted to the type of value that will be returned
- Terminating a function's header line with a semicolon
- Forgetting to include the data type of a function's parameters within the function header line

Summary

- A function is called by giving its name and passing data to it
 - If a variable is an argument in a call, the called function receives a copy of the variable's value
- **Common form of a user-written function:**

```
returnDataType functionName(parameter list)
{
    declarations and other C++ statements;
    return expression;
}
```

Summary (continued)

- A function's return type is the data type of the value returned by the function
 - If no type is declared, the function is assumed to return an integer value
 - If the function does not return a value, it should be declared as a `void` type
- Functions can directly return at most a single data type value to their calling functions
 - This value is the value of the expression in the return statement

Summary (continued)

- **Reference parameter:** passes the address of a variable to a function
- **Function prototype:** function declaration
- **Scope:** determines where in a program the variable can be used
- **Variable class:** determines how long the value in a variable will be retained