Chapter 6:
User-Defined Functions

Introduction

- Functions are often called modules
- They are like miniature programs that can be combined to form larger programs
- They allow complicated programs to be divided into manageable pieces

Predefined Functions

- In C++, a function is similar to that of a function in algebra
  - It has a name
  - It does some computation
- Some of the predefined mathematical functions are:

```
sqrt(x)
pow(x, y)
floor(x)
```

Predefined Functions (cont'd.)

- Predefined functions are organized into separate libraries
  - I/O functions are in `iostream` header
  - Math functions are in `cmath` header
- To use predefined functions, you must include the header file using an `include` statement
- See Table 6-1 in the text for some common predefined functions

User-Defined Functions

- Value-returning functions: have a return type
  - Return a value of a specific data type using the `return` statement
- Void functions: do not have a return type
  - *Do not* use a `return` statement to return a value

Value-Returning Functions

- To use these functions, you must:
  - Include the appropriate header file in your program using the include statement
  - Know the following items:
    - Name of the function
    - Number of parameters, if any
    - Data type of each parameter
    - Data type of the value returned: called the type of the function

Value-Returning Functions (cont'd.)

- Can use the value returned by a value-returning function by:
  - Saving it for further calculation
  - Using it in some calculation
  - Printing it
- A value-returning function is used in an assignment or in an output statement

Value-Returning Functions (cont'd.)

- Heading (or function header): first line of the function
  - Example: `int abs(int number)`
- Formal parameter: variable declared in the heading
  - Example: `number`
- Actual parameter: variable or expression listed in a call to a function
  - Example: `x = pow(u, v)`

Syntax: Value-Returning Function

- Syntax:

```
functionType functionName(formal parameter list)
{
    statements
}
```

- `functionType` is also called the data type or return type

Syntax: Formal Parameter List

```
dataType identifier, dataType identifier, ...
```

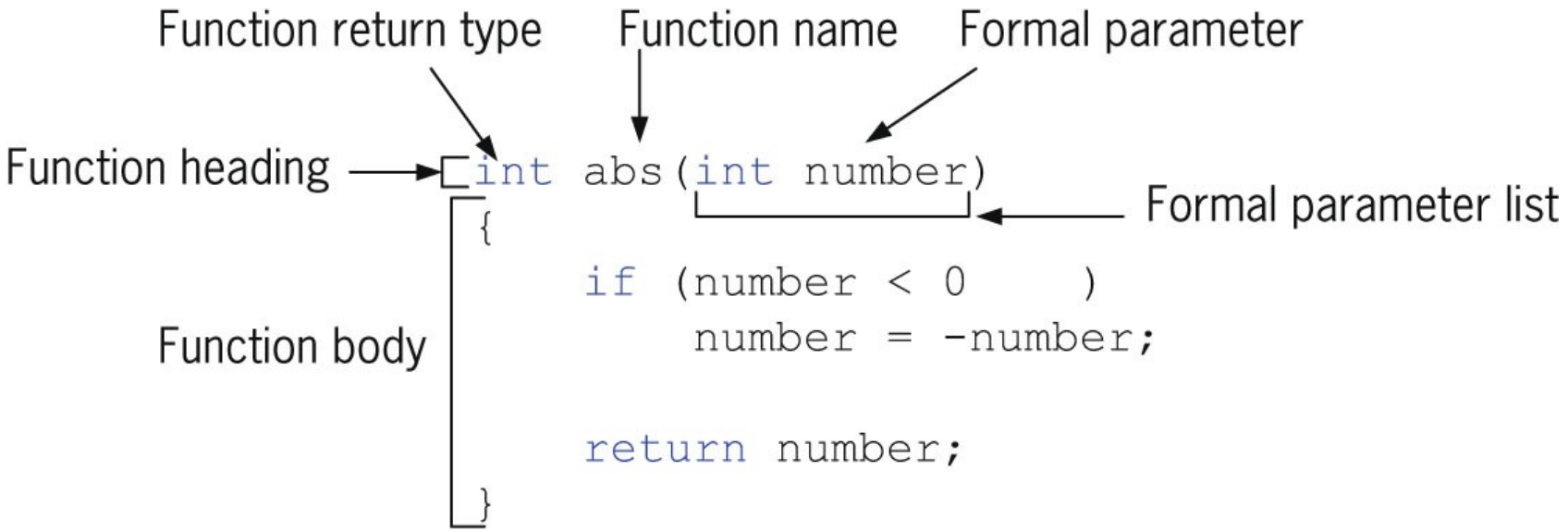

FIGURE 6-1 Various parts of the function abs

Function Call

- Syntax to call a value-returning function:

```
functionName(actual parameter list)
```

Syntax: Actual Parameter List

- Syntax of the actual parameter list:

```
expression or variable, expression or variable, ...
```

- Formal parameter list can be empty:

```
functionType functionName()
```

- A call to a value-returning function with an empty f

```
functionName()
```

list is:

`return` Statement

- Function returns its value via the `return` statement
  - It passes this value outside the function

Syntax: `return` Statement

- Syntax:

```
return expr;
```

- In C++, `return` is a reserved word
- When a return statement executes
  - Function immediately terminates
  - Control goes back to the caller
- When a `return` statement executes in the function `main`, the program terminates

Syntax: `return` Statement (cont'd.)

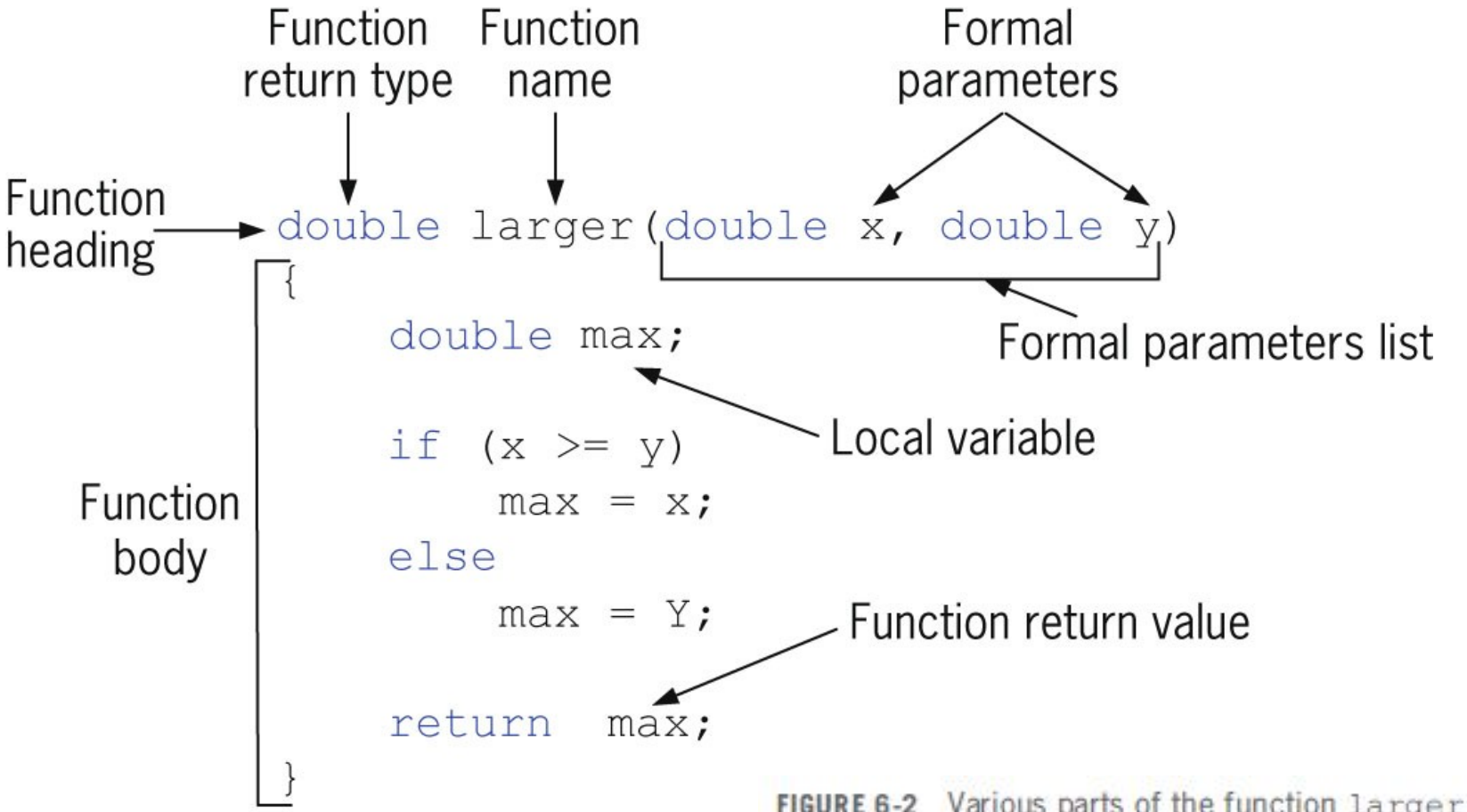

FIGURE 6-2 Various parts of the function `larger`

Function Prototype

- Function prototype: function heading without the body of the function
- Syntax:

```
functionType functionName(parameter list);
```

- Not necessary to specify the variable name in the parameter list
- Data type of each parameter must be specified

Value-Returning Functions: Some Peculiarities

```
int secret(int x)
{
    if (x > 5)              //Line 1
        return 2 * x;       //Line 2
}
```

A correct definition of the function secret is:

```
int secret(int x)
{
    if (x > 5)              //Line 1
        return 2 * x;       //Line 2

    return x;               //Line 3
}
```

Value-Returning Functions: Some Peculiarities (cont'd.)

```
return x, y;   //only the value of y will be returned


int funcRet1()
{
     int x = 45;

     return 23, x;   //only the value of x is returned
}

int funcRet2(int z)
{
     int a = 2;
     int b = 3;

     return 2 * a + b, z + b; //only the value of z + b is returned
}
```

Flow of Execution

- Execution always begins at the first statement in the function `main`
- Other functions are executed only when called
- Function prototypes appear before any function definition
  - Compiler translates these first
- Compiler can then correctly translate a function call

Flow of Execution (cont'd.)

- Function call transfers control to the first statement in the body of the called function
- When the end of a called function is executed, control is passed back to the point immediately following the function call
  - Function's returned value replaces the function call statement

Void Functions

- User-defined void functions can be placed either before or after the function `main`
- If user-defined void functions are placed after the function `main`
  - The function prototype must be placed before the function `main`
- Void function does not have a return type
  - `return` statement without any value is typically used to exit the function early

Void Functions (cont'd.)

- Formal parameters are optional
- A call to a void function is a stand-alone statement
- Void function definition syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

Void Functions (cont'd.)

- Formal parameter list syntax:

```
dataType& variable, dataType& variable, ...
```

- Function call syntax:

```
functionName(actual parameter list);
```

- Actual parameter list syntax:

```
expression or variable, expression or variable, ...
```

Void Functions (cont'd.)

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
- Reference parameter: a formal parameter that receives the location (memory address) of the corresponding actual parameter

Value Parameters

- If a formal parameter is a value parameter:
  - The value of the corresponding actual parameter is copied into it
  - Formal parameter has its own copy of the data
- During program execution
  - Formal parameter manipulates the data stored in its own memory space

Reference Variables as Parameters

- If a formal parameter is a reference parameter
  - It receives the memory address of the corresponding actual parameter
- During program execution to manipulate data
  - Changes to formal parameter will change the corresponding actual parameter

Reference Variables as Parameters (cont'd.)

- Reference parameters are useful in three situations:
  - Returning more than one value
  - Changing the actual parameter
  - When passing the address would save memory space and time

Value and Reference Parameters and Memory Allocation

- When a function is called
  - Memory for its formal parameters and its local variables is allocated in the function data area
- For a value parameter, the actual parameter's value is copied into the formal parameter's memory cell
  - Changes to the formal parameter do not affect the actual parameter's value

Value and Reference Parameters and Memory Allocation (cont'd.)

- For a reference parameter, the actual parameter's address passes to the formal parameter
  - Both formal and actual parameters refer to the same memory location
  - During execution, changes made to the formal parameter's value permanently change the actual parameter's value

Reference Parameters and Value-Returning Functions

- Can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value via `return` statement
- If a function needs to return more than one value, change it to a void function and use reference parameters to return the values

Scope of an Identifier

- Scope of an identifier: where in the program the identifier is accessible
- Local identifier: identifiers declared within a function (or block)
- Global identifier: identifiers declared outside of every function definition
- C++ does not allow nested functions
  - Definition of one function cannot be included in the body of another function

Scope of an Identifier (cont'd.)

- Rules when an identifier is accessed:
  - Global identifiers are accessible by a function or block if:
    - Declared before function definition
    - Function name different from identifier
    - Parameters to the function have different names
    - All local identifiers have different names

Scope of an Identifier (cont'd.)

- Rules when an identifier is accessed (cont'd.):
  - Nested block
    - Identifier accessible from declaration to end of block in which it is declared
    - Within nested blocks if no identifier with same name exists
  - Scope of function name similar to scope of identifier declared outside any block
    - i.e., function name scope = global variable scope

Scope of an Identifier (cont'd.)

- Some compilers initialize global variables to default values
- Scope resolution operator in C++ is `::`
- By using the scope resolution operator
  - A global variable declared before the definition of a function (or block) can be accessed by the function (or block)
  - Even if the function (or block) has an identifier with the same name as the global variable

Scope of an Identifier (cont'd.)

- To access a global variable declared after the definition of a function, the function must not contain any identifier with the same name
  - Reserved word `extern` indicates that a global variable has been declared elsewhere

Global Variables, Named Constants, and Side Effects

- Using global variables causes side effects
- A function that uses global variables is not independent
- If more than one function uses the same global variable:
  - Can be difficult to debug problems with it
  - Problems caused in one area of the program may appear to be from another area
- Global named constants have no side effects

Static and Automatic Variables

- Automatic variable: memory is allocated at block entry and deallocated at block exit
  - By default, variables declared within a block are automatic variables
- Static variable: memory remains allocated as long as the program executes
  - Global variables declared outside of any block are static variables

Static and Automatic Variables (cont'd.)

- Can declare a static variable within a block by using the reserved word static
- Syntax:

```
static dataType identifier;
```

- Static variables declared within a block are local to the block
  - Have same scope as any other local identifier in that block

Debugging: Using Drivers and Stubs

- Driver program: separate program used to test a function
- When results calculated by one function are needed in another function, use a function stub
- Function stub: a function that is not fully coded

Function Overloading: An Introduction

- In a C++ program, several functions can have the same name
- Function overloading: creating several functions with the same name
- Function signature: the name and formal parameter list of the function
  - Does *not* include the return type of the function

Function Overloading (cont'd.)

- Two functions are said to have different formal parameter lists if both functions have either:
  - A different number of formal parameters
  - If the number of formal parameters is the same, but the data type of the formal parameters differs in at least one position
- Overloaded functions must have different function signatures

Function Overloading (cont'd.)

- The parameter list supplied in a call to an overloaded function determines which function is executed

Functions with Default Parameters

- In a function call, the number of actual and formal parameters must be the same
  - C++ relaxes this condition for functions with default parameters
- Can specify the value of a default parameter in the function prototype
- If you do not specify the value for a default parameter when calling the function, the default value is used

Functions with Default Parameters (cont'd.)

- All default parameters must be the rightmost parameters of the function
- If a default parameter value is not specified:
  - You must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
- Cannot assign a constant value as a default value to a reference parameter

Summary

- Functions (modules) divide a program into manageable tasks
- C++ provides standard, predefined functions
- Two types of user-defined functions: value-returning functions and void functions
- Variables defined in a function heading are called formal parameters
- Expressions, variables, or constant values in a function call are called actual parameters

Summary (cont'd.)

- Function heading and the body of the function are called the definition of the function
- A value-returning function returns its value via the `return` statement
- A prototype is the function heading without the body of the function
- User-defined functions execute only when they are called
- Void functions do not have a data type

Summary (cont'd.)

- Two types of formal parameters:
  - A value parameter receives a copy of its corresponding actual parameter
  - A reference parameter receives the memory address of its corresponding actual parameter
- Variables declared within a function (or block) are called local variables
- Variables declared outside of every function definition (and block) are global variables

Summary (cont'd.)

- Automatic variable: variable for which memory is allocated on function/block entry and deallocated on function/block exit
- Static variable: memory remains allocated throughout the execution of the program
- C++ functions can have default parameters