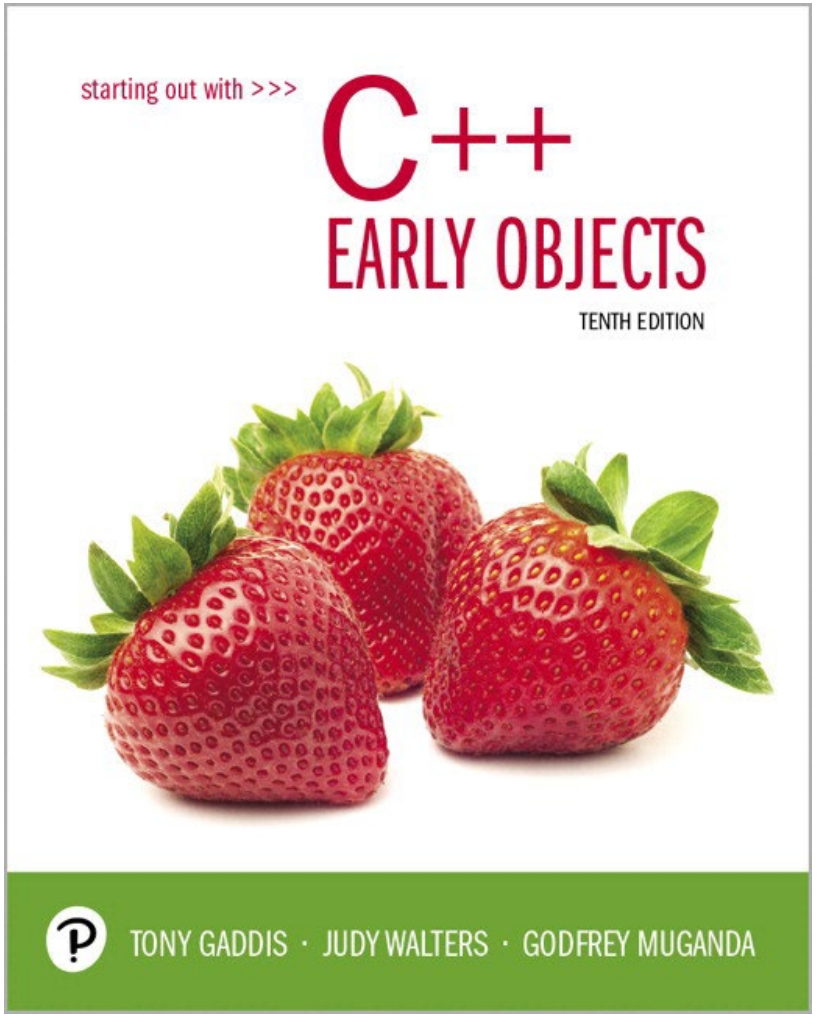
Starting Out with C++ Early Objects

Tenth Edition



Chapter 6

Functions

Topics 1 of 2

6.1 Modular Programming

6.2 Defining and Calling Functions

6.3 Function Prototypes

6.4 Sending Data into a Function

6.5 Passing Data by Value

6.6 The `return` Statement

6.7 Returning a Value from a Function

6.8 Returning a Boolean Value

Topics 2 of 2

6.9 Using Functions in a Menu-Driven Program

6.10 Local and Global Variables

6.11 Static Local Variables

6.12 Default Arguments

6.13 Using Reference Variables as Parameters

6.14 Overloading Functions

6.15 The `exit()` Function

6.16 Stubs and Drivers

6.1 Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules. It supports the divide-and-conquer approach to solving a problem.
- Function: a collection of statements to perform a specific task
- Motivation for modular programming
  - Simplifies the process of writing programs
  - Improves the maintainability of programs

6.2 Defining and Calling Functions

- Function call: a statement that causes a function to execute
- Function definition: the statements that make up a function

Function Definition 1 of 2

- A definition includes

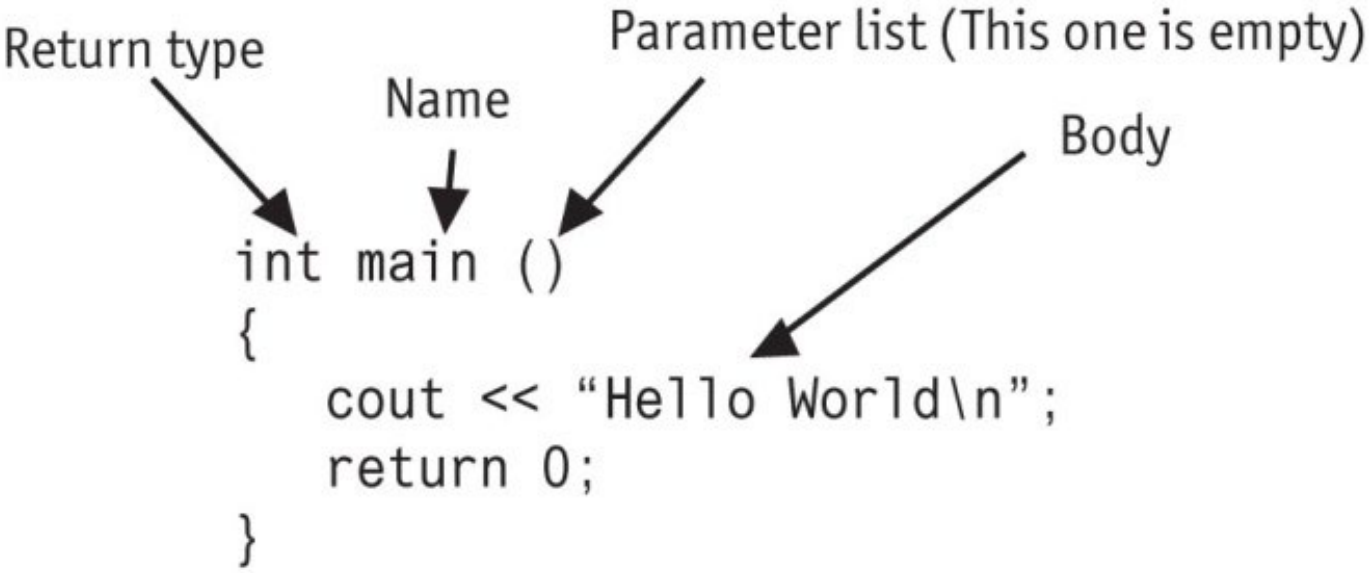
  name: the name of the function. Function names follow the same rules as variable names

  parameter list: the variables that hold the values that are passed to the function when it is called

  body: the statements that perform the function's task

  return type: data type of the value the function returns to the part of the program that called it

Function Definition 2 of 2



Function Header

- The function header consists of
  - the function *return type*
  - the function *name*
  - the function *parameter list*
- Example:

```
int main()
```

- Note: There is no `;` at the end of the 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 << "\tMonthly Sales\n";
}
```

Calling a Function 1 of 2

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

```
printHeading();
```

- When a function is called, the program executes the body of the function
- After the function terminates, execution resumes in the calling module at the point of call

Calling a Function 2 of 2

- **`main`** is automatically called when the program starts
- **`main`** can call any number of functions
- Functions can call other functions

6.3 Function Prototypes 1 of 2

The compiler must know the following about a function before it is called

- its name
- the return type
- the number of parameters
- the data type of each parameter

Function Prototypes 2 of 2

There are multiple ways to notify the compiler about a function before making a call to the function:

- Place the function definition before the calling function's definition
- Use a function prototype (similar to the header of the function)
  - Header: `void printHeading()`
  - Prototype: `void printHeading();`

Prototype Notes

- Place prototypes near the top of the program
- A program must include either a prototype or full function definition before any call to the function, otherwise a compiler error occurs
- When using prototypes, the function definitions can be placed in any order in the source file. Traditionally, **`main`** is placed first.

6.4 Sending Data into a Function

- You can pass values into a function at time of a call

```
c = sqrt(a*a + b*b);
```

- Values passed to a function are arguments
- Variables in a function that hold values passed as arguments are parameters
- Alternate names:
  - argument: actual argument, actual parameter
  - parameter: formal argument, formal parameter

Parameters, Prototypes, and Function Headings

- For each function argument,
  - the prototype must include the data type of each parameter in its `()`. It may also include a parameter name:

```
void evenOrOdd(int);   or
void evenOrOdd(int num);  // prototype
```

  - the header must include a declaration, with variable type and name, for each parameter in its `()`

```
void evenOrOdd(int num)    //header
```

- The call for the above function could look like this:

```
evenOrOdd(val);            //call
```

Function Call Notes

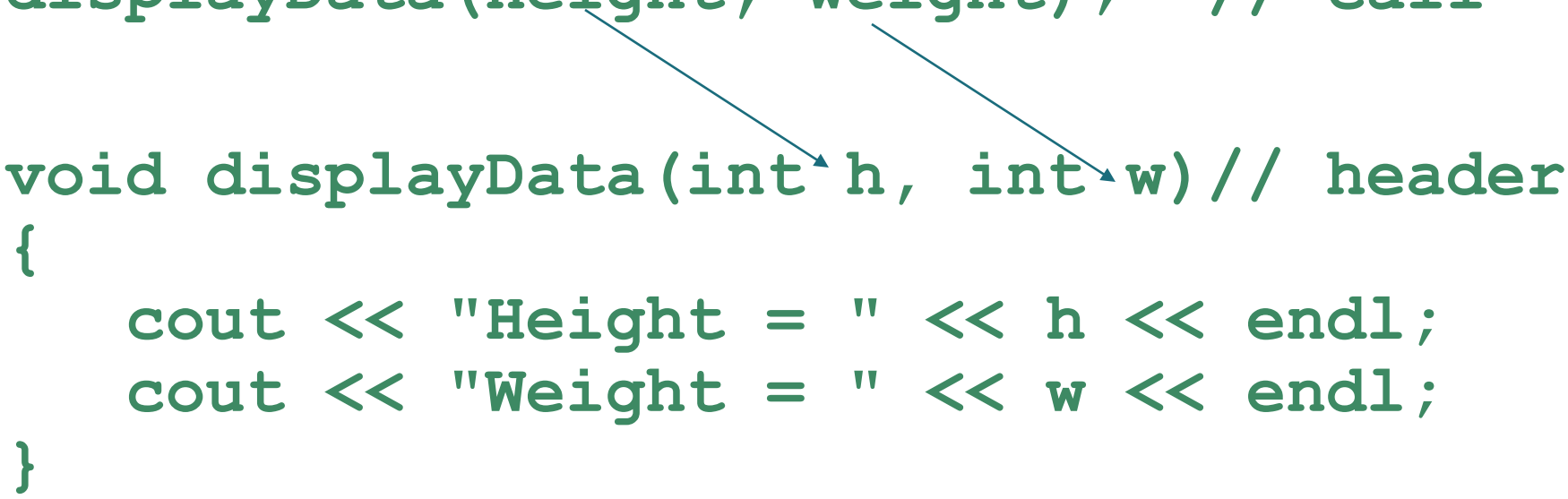
- The value of the argument is copied into the parameter when the function is called
- A function can have > 1 parameter
- There must be a data type listed in the prototype `()` and an argument declaration in the function heading `()` for each parameter
- Arguments will be promoted/demoted as necessary to match parameters. Be careful!

Calling Functions with Multiple Arguments

When calling a function with multiple arguments

- the number of arguments in the call must match the function prototype and definition
- the value of the first argument will be copied into the first parameter, the value of the second argument into the second parameter, etc.

Calling Functions with Multiple Arguments Illustration



6.5 Passing Data by Value

- Pass by value: when an argument is passed to a function, a copy of its value is placed in the parameter
- The function cannot access the original argument
- Changes made to the parameter in the function do not affect the value of the argument in the calling function

Passing Data to Parameters by Value

- Example:
```
int val = 5;
evenOrOdd(val);
```

- **`evenOrOdd`** may change the variable **`num`**, but it will have no effect on variable **`val`**

6.6 The `return` Statement

- Is used to end execution of a function
- It can be placed anywhere in a function
  - Statements that follow the **`return`** statement will not be executed
- It can be used to prevent abnormal termination of program
- Without a **`return`** statement, the function ends at its last `}`

6.7 Returning a Value from a Function

- The **`return`** statement can be used to return a value from a function to the module that made the function call
- The prototype and function header must indicate the data type of return value (not **`void`**)
- The calling function should use the returned value, *e.g.*,
  - assign it to a variable
  - send it to **`cout`**
  - use it in an arithmetic computation
  - use it in a relational expression

Returning a Value – the `return` Statement

- Format: `return expression;`
- ***expression*** may be a variable, a literal value, or an expression.
- ***expression*** should be of the same data type as the declared return type of the function (it will be converted if not)

6.8 Returning a Boolean Value

- A function can return **`true`** or **`false`**
- You can declare the return type in the function prototype and header as **`bool`**
- The function body must contain **`return`** statement(s) that return **`true`**, **`false`**, or **`bool`** variables or expressions.
- The calling function can use the return value in a relational expression

Boolean `return` Example

```
bool isValid(int);              // prototype

bool isValid(int val)           // header
{
   int min = 0, max = 100;
   if (val >= min && val <= max)
      return true;
   else
      return false;
}

if (isValid(score))             // call
   …
```

Programming Style and `return` statements

A function may calculate a return value and use a single return statement. The previous example could be written as:

```
bool isValid(int val)      // header
{
   bool result;
   int min = 0, max = 100;
   if (val >= min && val <= max)
      result = true;
   else
      result = false;
   return result;          // single return
}
```

6.9 Using Functions in a Menu-Driven Program

Functions can be used

- to implement user choices from menus
- to implement general-purpose tasks
  - Higher-level functions can call general-purpose functions
  - This minimizes the total number of functions and speeds program development time

Screen Management in a Menu-Driven Program

You can clear the screen to remove prior output while a program is running:

- Windows: `system("cls");`
- Linux and Mac OS: `system("clear");`

To allow the user enough time to read output before the screen clears, use code like:

```
cout << "Press the Enter key to continue.";
cin.get();  // clear the input buffer
cin.get();  // get the Enter key
```

6.10 Local and Global Variables

- local variable: is defined within a function or a block; accessible only within the function or the block. Parameters are also local variables.
- Other functions and blocks can define variables with the same name
- When a function is called, local variables in the calling function are not accessible from within the called function

Local Variable Lifetime

- A local variable only exists while its defining function is executing
- Local variables created when a function defines them and are destroyed when the function terminates
- Data cannot be retained in local variables between calls to the function in which they are defined

Local and Global Variables

- global variable: a variable defined outside all functions; it is accessible to all functions within its scope
- Can be sean as an easy way to share data between functions
- Scope of a global variable is from its point of definition to the program end
- Use sparingly

Initializing Local and Global Variables

- Local variables must be initialized by the programmer
- Global variables are initialized to `0` (numeric) or `NULL` (character) when the variable is defined. These can be overridden with explicit initial values.

Global Variables – Why 'Use Sparingly'?

Global variables make:

- Programs that are difficult to debug
- Functions that cannot easily be re-used in other programs
- Programs that are hard to understand

Global Constants

- A global constant is a named constant that can be used by every function in a program
- It is useful if there are unchanging values that are used throughout the program
- They are safer to use than global variables, since the value of a constant cannot be modified during program execution

Local and Global Variable Names

- Local variables can have same names as global variables
- When a function contains a local variable that has the same name as a global variable, the global variable is unavailable from within the function. The local definition "hides" or "shadows" the global definition.

6.11 Static Local Variables

- Local variables
  - Only exist while the function is executing
  - Are redefined each time function is called
  - Lose their contents when function terminates
- **`static`** local variables
  - Are defined with key word **`static`**

```
static int counter;
```

  - Are defined and initialized only the first time the function is executed
  - Retain their values between function calls

6.12 Default Arguments 1 of 2

- Values that are passed automatically if arguments are missing from a function call
- Must be a constant or literal declared in the prototype or header (whichever occurs first)

```
void evenOrOdd(int x = 0);
```

- Multi-parameter functions may have default arguments for some or all parameters

```
int getSum(int, int=0, int=0);
```

Default Arguments 2 of 2

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int, int=0, int=0);// OK

int getSum(int, int=0, int);  // wrong!
```

- 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);   // wrong!
```

6.13 Using Reference Variables as Parameters

- This is a mechanism that allows a function to work with the *original* argument from the function call, not a *copy* of the argument
- It allows the function to modify values that are stored in the calling environment
- It provides a way for the function to 'return' more than 1 value

Reference Variables

- A reference variable is an alias for another variable
- When used as a function parameter, it is defined with an ampersand (**&**) in the prototype and in the header

```
void getDimensions(int&, int&);
```

- Changes made to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters by reference

Pass by Reference Example

```
void squareIt(int &); //prototype

void squareIt(int &num)
{
    num *= num;
}

int localVar = 5;

squareIt(localVar);   // localVar now
                      // contains 25
```

Reference Variable Notes

- Each reference parameter must contain **&**
- An argument passed to a reference parameter must be a variable. It cannot be an expression or a constant.
- Use only when it is appropriate, such as when the function must input or change the value of the argument passed to it.
- Files (*i.e.*, file stream objects) should be passed by reference.

6.14 Overloading Functions

- The signature of a function is the function name and the data types of the parameters, in order. The return type is not part of the signature.
- Overloaded functions are two or more functions that have the same name but different signatures
- This can be used to create functions that perform the same task but take different parameter types or a different number of parameters
- The compiler will determine which version of the function to call by the argument and parameter lists

Overloaded Functions Example

If a program has these overloaded functions,

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

then 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
```

6.15 The `exit()` Function

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

`exit()` – Passing Values to Operating System

- Use an integer value to indicate program status
- Often, 0 means successful completion, non-zero indicates a failure condition
- Can use named constants defined in **`cstdlib`**:
  - **`EXIT_SUCCESS`** and
  - **`EXIT_FAILURE`**

6.16 Stubs and Drivers

- Stub: A dummy function used in place of an actual function
- It usually displays a message indicating it was called. May also display the values of the arguments and return a test value.
- Driver: A function that tests a function by calling it
- Stubs and drivers are useful for testing and debugging program logic and design

Copyright