Chapter 6: Functions

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley
is an imprint of
PEARSON

Topics

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 (continued)

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
- Function: a collection of statements to perform a specific task
- Motivation for modular programming
  - Simplifies the process of writing programs
  - Improves maintainability of programs

6.2 Defining and Calling Functions

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

Function Definition

- Definition includes

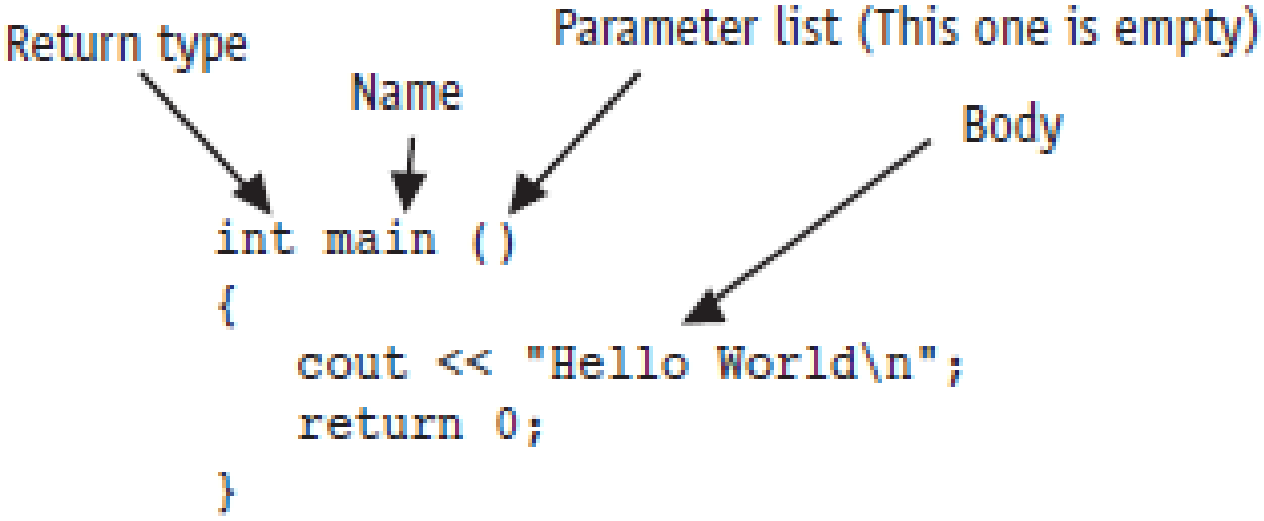
  name: name of the function. Function names follow same rules as variable names

  parameter list: variables that hold the values passed to the function

  body: 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



Function Header

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

```
int main()
```

- Note: 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

- 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

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

6.3 Function Prototypes

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

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

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 **function prototype** (similar to the heading of the function
  - Heading: `void printHeading()`
  - Prototype: `void printHeading();`

Prototype Notes

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

6.4 Sending Data into a Function

- Can pass values into a function at time of call

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

- Values passed to function are arguments
- Variables in 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 `()`

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

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

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

- The function call for the above function would look like this: `evenOrOdd(val);  //call`

Function Call Notes

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

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 first argument will be copied into the first parameter, the second argument into the second parameter, etc.

Calling Functions with Multiple Arguments Illustration

```
displayData(height, weight);   // call


void displayData(int h, int w)// heading
{
   cout << "Height = " << h << endl;
   cout << "Weight = " << w << endl;
}
```

6.5 Passing Data by Value

- Pass by value: when argument is passed to a function, a copy of its value is placed in the parameter
- Function cannot access the original argument
- Changes 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** can change variable **num**, but it will have no effect on variable **val**

6.6 The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - Any statements that follow the **`return`** statement will not be executed
- 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

- **`return`** statement can be used to return a value from the function to the module that made the function call
- Prototype and definition must indicate data type of return value (not **`void`**)
- Calling function should use return 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 (will be converted if not)

6.8 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

Boolean `return` Example

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

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

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

6.9 Using Functions in a Menu-Driven Program

Functions can be used

- to implement user choices from menu
- 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

6.10 Local and Global Variables

- local variable: defined within a function or block; accessible only within the function or block
- 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 and Global Variables

- global variable: a variable defined outside all functions; it is accessible to all functions within its scope
- Easy way to share large amounts of data between functions
- Scope of a global variable is from its point of definition to the program end
- Use sparingly

Local Variable Lifetime

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

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

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

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 contents between function calls

6.12 Default Arguments

- Values passed automatically if arguments are missing from the function call
- Must be a constant declared in prototype

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

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

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


Default Arguments

- 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

- 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 1 value

Reference Variables

- A reference variable 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

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 **&**
- Argument passed to reference parameter must be a variable (cannot be an expression or constant)
- Use only when 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

- Overloaded functions are two or more functions that 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 list

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

`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: dummy function in place of actual function
- Usually displays a message indicating it was called. May also display parameters
- Driver: function that tests a function by calling it
- Stubs and drivers are useful for testing and debugging program logic and design

Chapter 6: Functions

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley
is an imprint of
PEARSON