Functions

Modified from Chapters 4 & 5

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Top Down Design

- Top Down Design (also called stepwise refinement)
  - Break the algorithm into subtasks
  - Break each subtask into smaller subtasks
  - Eventually the smaller subtasks are trivial to implement in the programming language

- We can use functions to implement these smaller subtasks.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Procedural Abstraction

- The Black Box Analogy
  - A black box refers to something that we know how to use, but the method of operation is unknown

- Functions and the Black Box Analogy
  - A programmer who uses a function needs to know what the function does, not how it does it
  - A programmer needs to know what will be produced if the proper arguments are put into the box

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overview

- Predefined Functions
- Programmer-defined Functions
- Scope and Local Variables
- Call-By-Value Parameters and Call-By-Reference Parameters
- Overloading Function Names
- Debugging Techniques

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Predefined Functions

Modified from Section 4.2

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Function Libraries

- C++ comes with libraries of predefined functions
- A library must be “included” in a program before using functions in this library
- An include directive tells the compiler which library header file to include.
- Example: to include the math library:

```
#include <cmath>
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Function Calls

- Syntax: Function_name(Argument_List)
  - Argument_List is a comma separated list: (Argument_1, Argument_2, … , Argument_Last)
- Examples:
  - pow is a function in the “cmath” library. It raises a number to the given power.
  - One of its declarations is
    - double pow(double base, double exp);
  - cout << “2.5 to the power 3.0 is “ << pow(2.5, 3.0);
  - Check **cppreference.com** for predefined function declarations

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Programmer-Defined Functions

Modified from Sections 4.3, 5.1

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Programmer-Defined Functions

- Two components of a function definition
  - Function declaration (or function prototype)
    - Shows how the function is called
    - Must appear in the code before the function can be called
    - Syntax: Type_returned Function_Name(Parameter_List);

;

  - Function definition
    - Describes how the function does its task
    - Can appear before or after the function is called
    - Syntax: Type_returned Function_Name(Parameter_List)

```
{
        // code to make the function work
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Alternate Declarations

- Two forms for function declarations
  - List formal parameters' types and names
  - List types of formal parameters, but no names
  - Examples:

```
double total_cost(int number_par, double price_par);
double total_cost(int, double);
```

- Function headers (the first line of a function definition) must always list formal parameter names.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Placing Definitions

- A function call must be preceded by either
  - The function’s declaration

  OR

  - The function’s definition. If the function’s definition precedes the call, a declaration is not needed

- Placing the function declaration prior to the main function and the function definition after the main function leads naturally to building your own libraries in the future.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

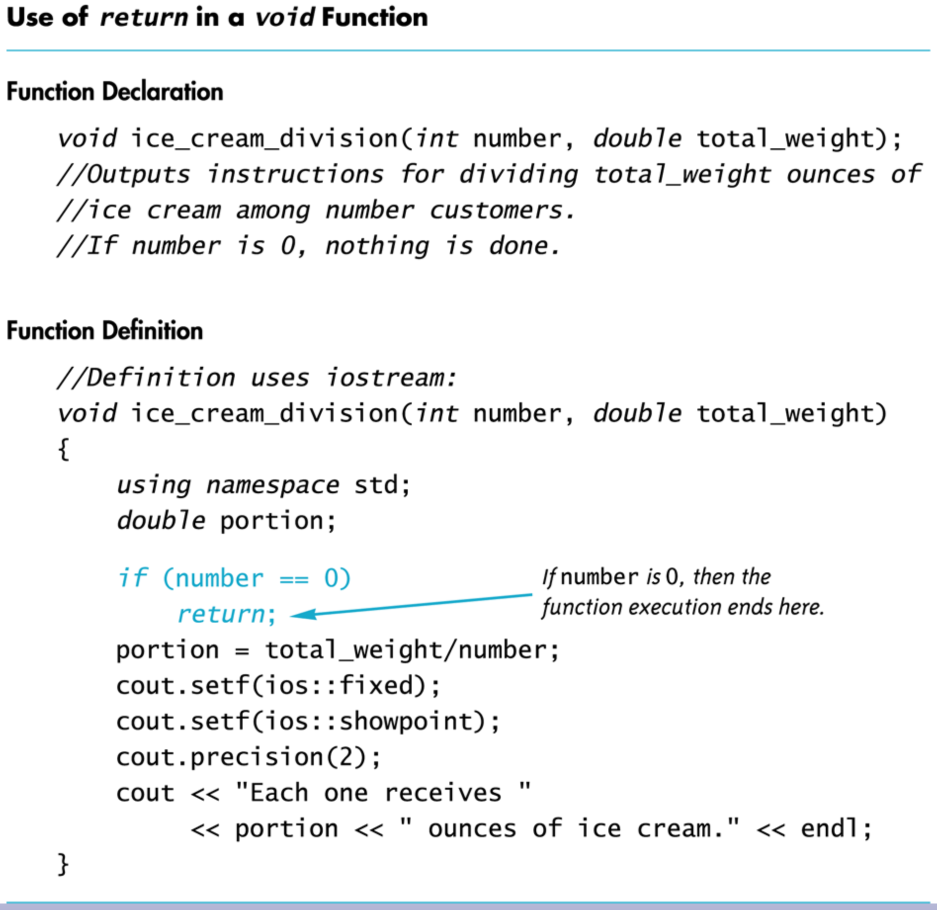
Return Type and The Return Statement

- One return statement ends the function call immediately
- It returns the value calculated by the function
- Syntax: return expression;
  - A maximum of one expression can be included in a return statement, i.e., a function can return a maximum of one value
  - The data type of expression should match the declared return data type of the function
- Define a void-function to implement a function that returns no value
  - Keyword void replaces the type of the value returned
    - void means that no value is returned by the function
  - The return statement does not include an expression
    - i.e. “return;”

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

void-Functions: Why Use a Return?

- A return statement can end the function execution before reaching the last line



Use of *return* in a *void* Function

Function Declaration

```
void ice_cream_division(int number, double total_weight);
//Outputs instructions for dividing total_weight ounces of
//ice cream among number customers.
//If number is 0, nothing is done.
```

Function Definition

```
//Definition uses iostream:
void ice_cream_division(int number, double total_weight)
{
    using namespace std;
    double portion;

    if (number == 0)
        return;
    portion = total_weight/number;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Each one receives "
         << portion << " ounces of ice cream." << endl;
}
```

If number *is* 0*, then the function execution ends here.*

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Scope and Local Variables

Modified from Sections 3.2, 4.5

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Scope of Variables

- Scope of a variable is the extent of the program code within which the variable can we accessed or declared or worked with.
- Local variables: variables defined within a function or a block are said to be local to this function or block.
  - Local variables do not exist outside the function/block in which they are declared.
- Global variables: variables defined outside of all of the functions and blocks, which can be accessed from any part of the program.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Local Variables in Blocks

- A block is a section of code enclosed by braces
- A statement block is a block that is not a function body or the body of the main part of a program
- The scope of a variable local to a block begins from the point of its declaration and ends at the end of the block
- Statement blocks can be nested in other statement blocks
  - If a single identifier is declared as a variable in each of two blocks, one within the other (nested), then these are two different variables with the same name
  - The inner block variable's scope begins from the point of its declaration and ends at the end of the inner block
  - The inner block variable's scope is excluded from the outer block variable's scope

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Local Variables in Functions

- Variables declared in a function:
  - Are local to that function, they cannot be used from outside the function
  - Their scopes begin from the point of declarations and end at the end of the function
- Function arguments are also local to the function
  - They are used just as if they were declared in the function body
  - Do NOT re-declare the formal parameters in the function body, they are declared in the function declaration

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Global Variables and Constants

- Global Variables
  - Have the program as their scope
  - Declared outside any function body
  - Declared before any function that uses it
- Global non-constant variables are rarely used.
  - Generally make programs more difficult to understand and maintain
- Global constant variables:
  - Example:

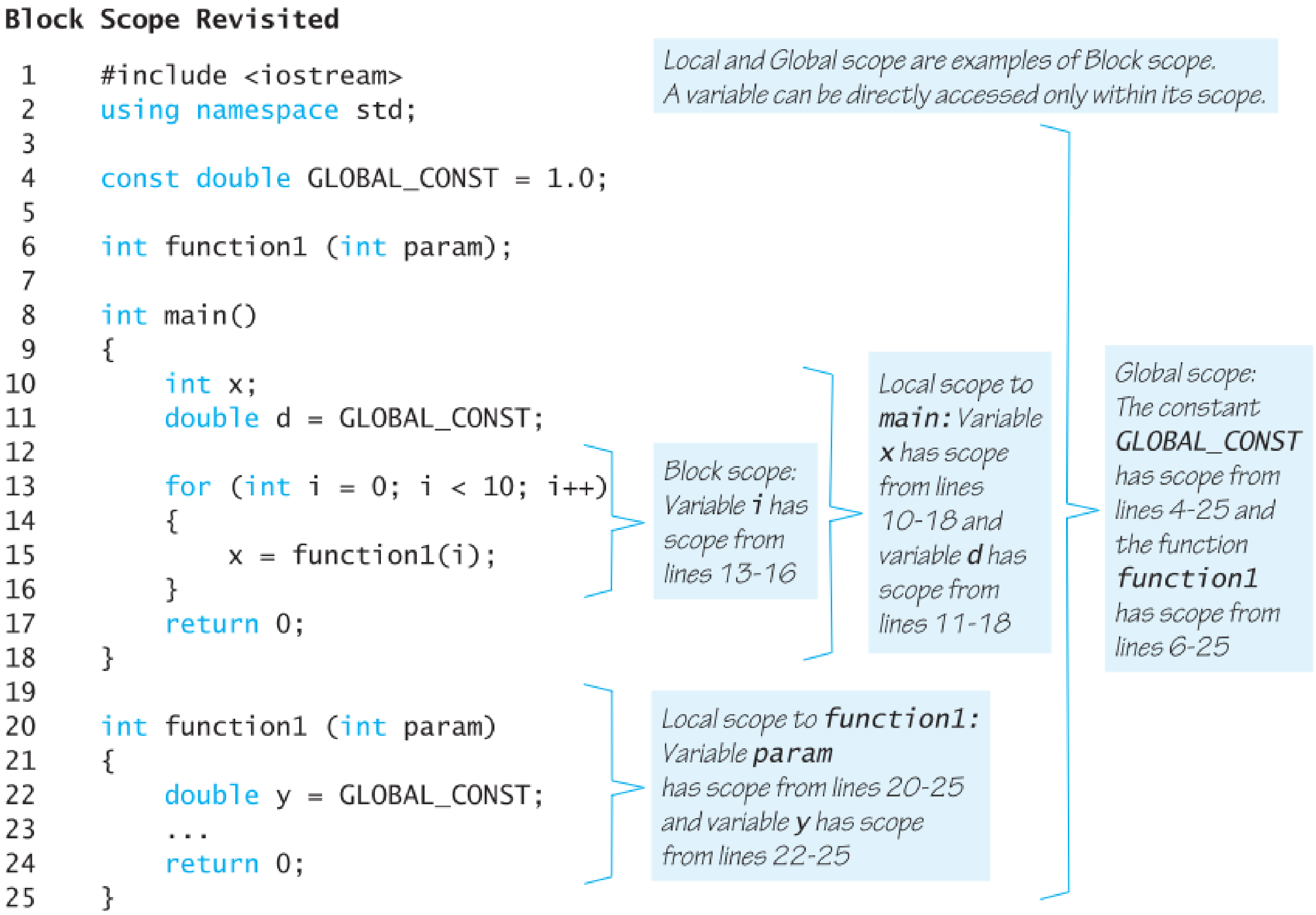
    ```
    const double PI = 3.14159;
    double volume(double);
    int main()
    {…}
    ```

  - PI is available to the main function and to function volume

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 4.14

Back Next



Block Scope Revisited

```
 1    #include <iostream>
 2    using namespace std;
 3
 4    const double GLOBAL_CONST = 1.0;
 5
 6    int function1 (int param);
 7
 8    int main()
 9    {
10        int x;
11        double d = GLOBAL_CONST;
12
13        for (int i = 0; i < 10; i++)
14        {
15            x = function1(i);
16        }
17        return 0;
18    }
19
20    int function1 (int param)
21    {
22        double y = GLOBAL_CONST;
23        ...
24        return 0;
25    }
```

Local and Global scope are examples of Block scope. A variable can be directly accessed only within its scope.

*Block scope: Variable **i** has scope from lines 13-16*

*Local scope to **main:** Variable **x** has scope from lines 10-18 and variable **d** has scope from lines 11-18*

*Global scope: The constant **GLOBAL_CONST** has scope from lines 4-25 and the function **function1** has scope from lines 6-25*

*Local scope to **function1:** Variable **param** has scope from lines 20-25 and variable **y** has scope from lines 22-25*

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Call-By-Value Parameters and Call-By-Reference Parameters

Modified from Section 5.2

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Call-by-Value Parameters

- Call-by-value means that the formal parameters receive the values of the arguments
- When a function is called, the formal parameters are initialized to the values of the arguments in the function call
- Call-by-value parameters do not modify the variables used in the function call

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Call-by-Reference Parameters

- Call-by-value is not adequate when we need a sub-task to obtain input values
  - To obtain input values, we need to change the variables that are arguments to the function

- Call-by-reference parameters allow us to change the variable used in the function call
  - Arguments for call-by-reference parameters must be non-constant variables, not numbers

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Call-by-Reference Example

-
  ```
  void get_input(double& f_temp)
   {
          cout << “ Convert a Fahrenheit temperature to
  Celsius.\n” << “ Enter a temperature in Fahrenheit: “;
          cin >> f_temp;
  }
  ```

- ‘&’ symbol (ampersand) identifies f_temp as a call-by-reference parameter
  - Used in both declaration and definition!

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Call-By-Reference Details

- Call-by-reference works almost as if the argument variable is substituted for the formal parameter, not the argument's value
- The memory location of the argument variable is given to the formal parameter
  - Whatever is done to a formal parameter in the function body, is actually done to the value at the memory location of the argument variable
- Call-by-value and call-by-reference parameters can be mixed in the same function

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Call Comparisons
Call-By-Reference vs Value

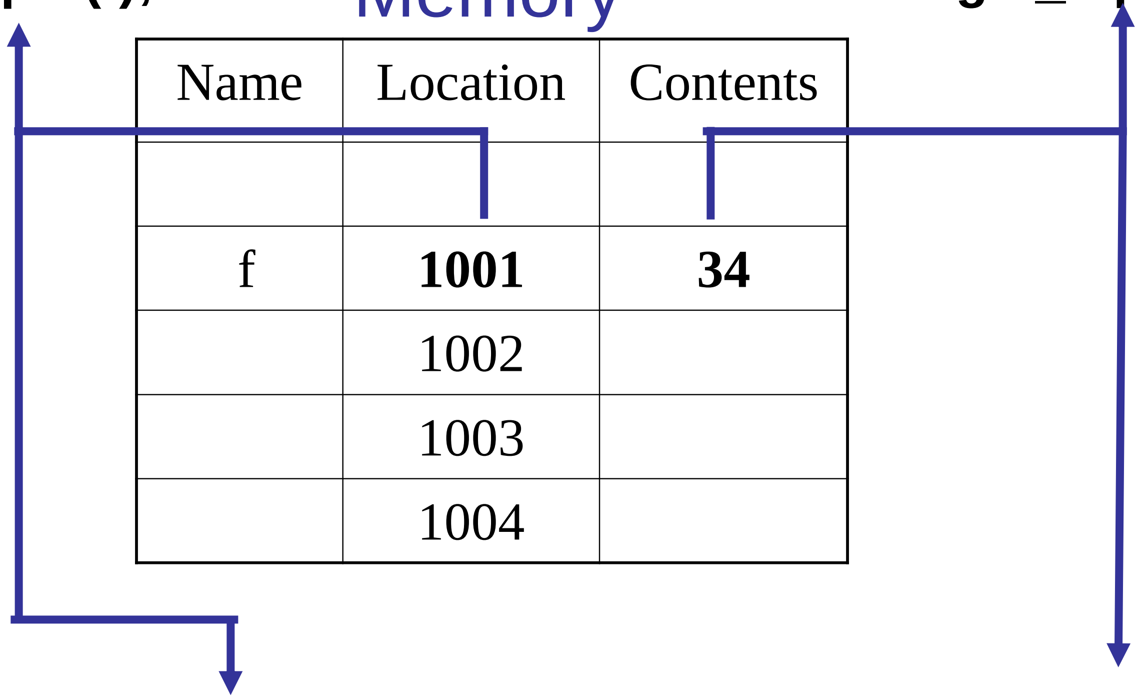
- Call-by-reference
  - The function call:

get_input(f);

- Call-by-value
  - The function call:

get_input(f);

Memory



Name	Location	Contents
f	**1001**	**34**
	1002	
	1003	
	1004	

void get_input(double& f_temp);

void get_input(double f_temp);

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Example: swap_values

-
  ```
  void swap(int& variable1, int& variable2)
  {
      int temp = variable1;
      variable1 = variable2;
      variable2 = temp;
  }
  ```
- If called with swap(first_num, second_num);
  - first_num is substituted for variable1 in the parameter list
  - second_num is substituted for variable2 in the parameter list
  - temp is assigned the value of variable1 (first_num) since the next line will loose the value in first_num
  - variable1 (first_num) is assigned the value in variable2 (second_num)
  - variable2 (second_num) is assigned the original value of variable1 (first_num) which was stored in temp

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Choosing Parameter Types

- How do you decide whether a call-by-reference or call-by-value formal parameter is needed?
  - Does the function need to change the value of the variable used as an argument?
    - Yes -> Use a call-by-reference formal parameter
    - No -> Use a call-by-value formal parameter
  - Does the function need to return multiple values as results?
    - Yes -> Use multiple call-by-reference formal parameters to hold the results

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading Function Names

Modified from Section 4.6

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading Function Names

- C++ allows more than one definition for the same function name
  - Very convenient for situations in which the “same” function is needed for different numbers or types of arguments
- Overloading a function name means providing more than one declaration and definition using the same function name

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading Details

- Overloaded functions
  - Must have different numbers of formal parameters
    AND / OR
  - Must have at least one different type of parameter

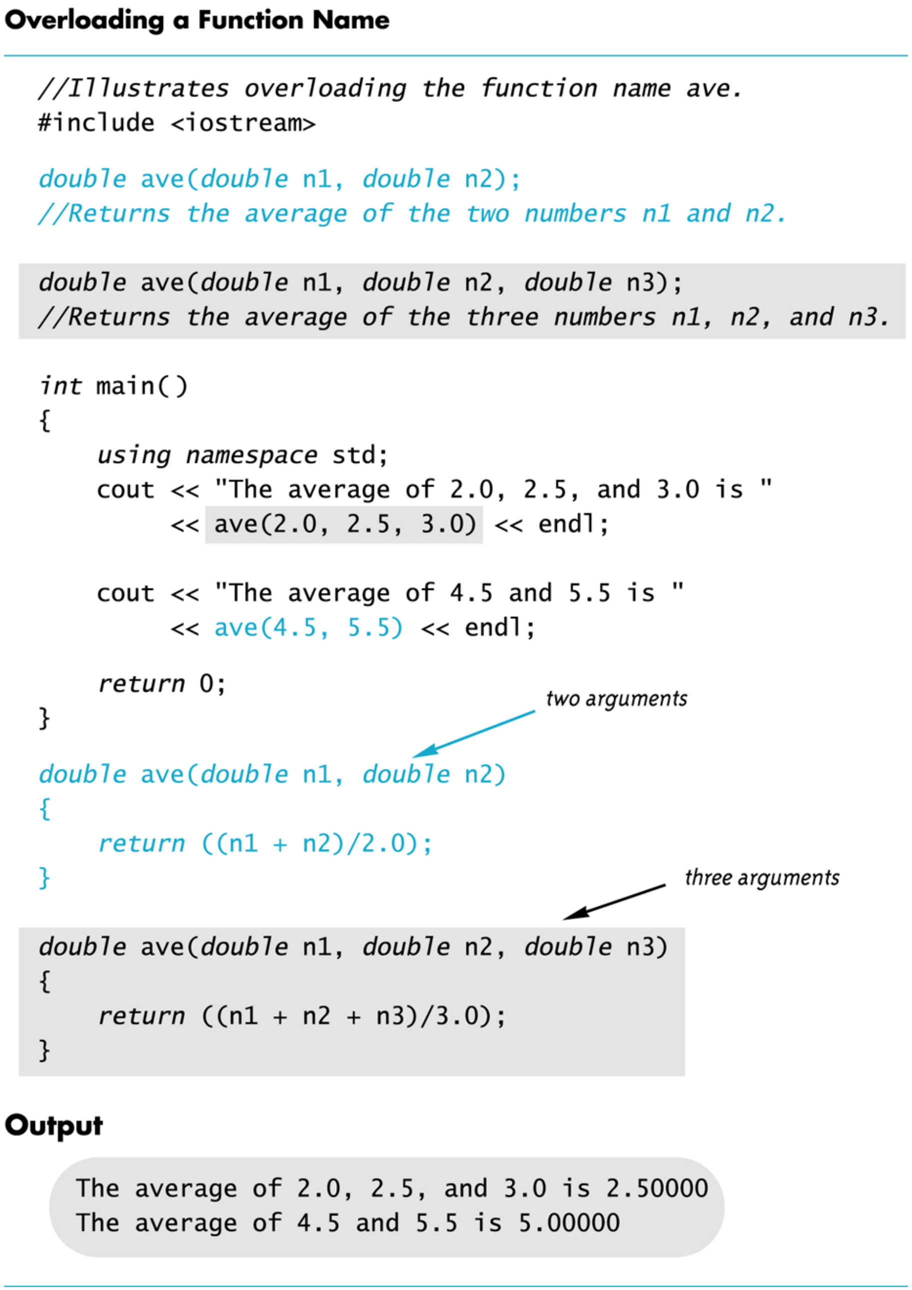
  - Can have different return types

Display 4.17

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 4.17

Back Next



Overloading a Function Name

```
//Illustrates overloading the function name ave.
#include <iostream>

double ave(double n1, double n2);
//Returns the average of the two numbers n1 and n2.

double ave(double n1, double n2, double n3);
//Returns the average of the three numbers n1, n2, and n3.

int main()
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;

    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;

    return 0;
}
```

two arguments

```
double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

three arguments

```
double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

Output

```
The average of 2.0, 2.5, and 3.0 is 2.50000
The average of 4.5 and 5.5 is 5.00000
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Debugging Techniques

Modified from Sections 5.4, 5.5

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

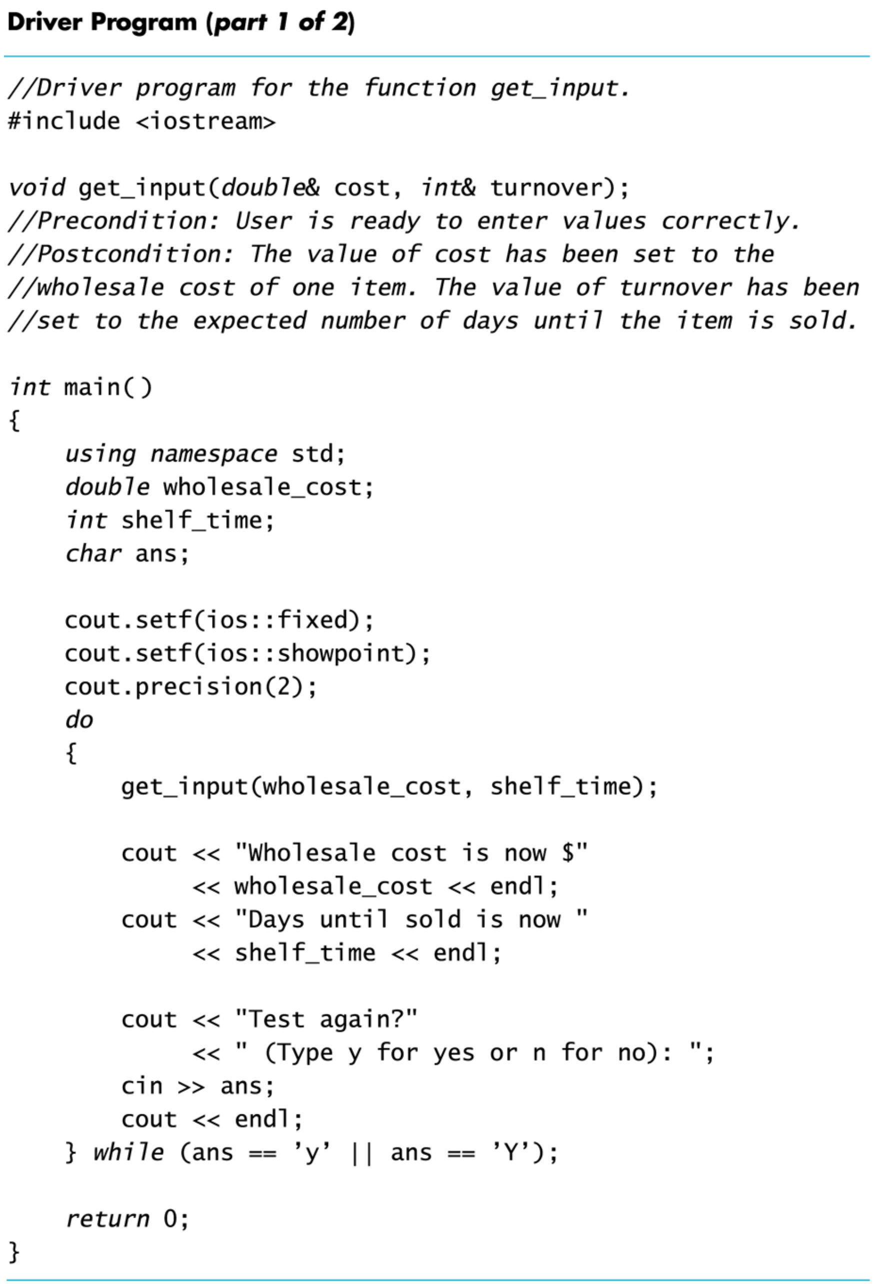
Testing and Debugging Functions

- Each function should be tested as a separate unit
- Testing individual functions facilitates finding mistakes
- Driver programs allow testing of individual functions
- Once a function is tested, it can be used in the driver program to test other functions
- Function get_input is tested in the driver program of Display 5.10 (1) and Display 5.10 (2)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 5.10 (1/2)

Back Next



Driver Program (*part 1 of 2*)

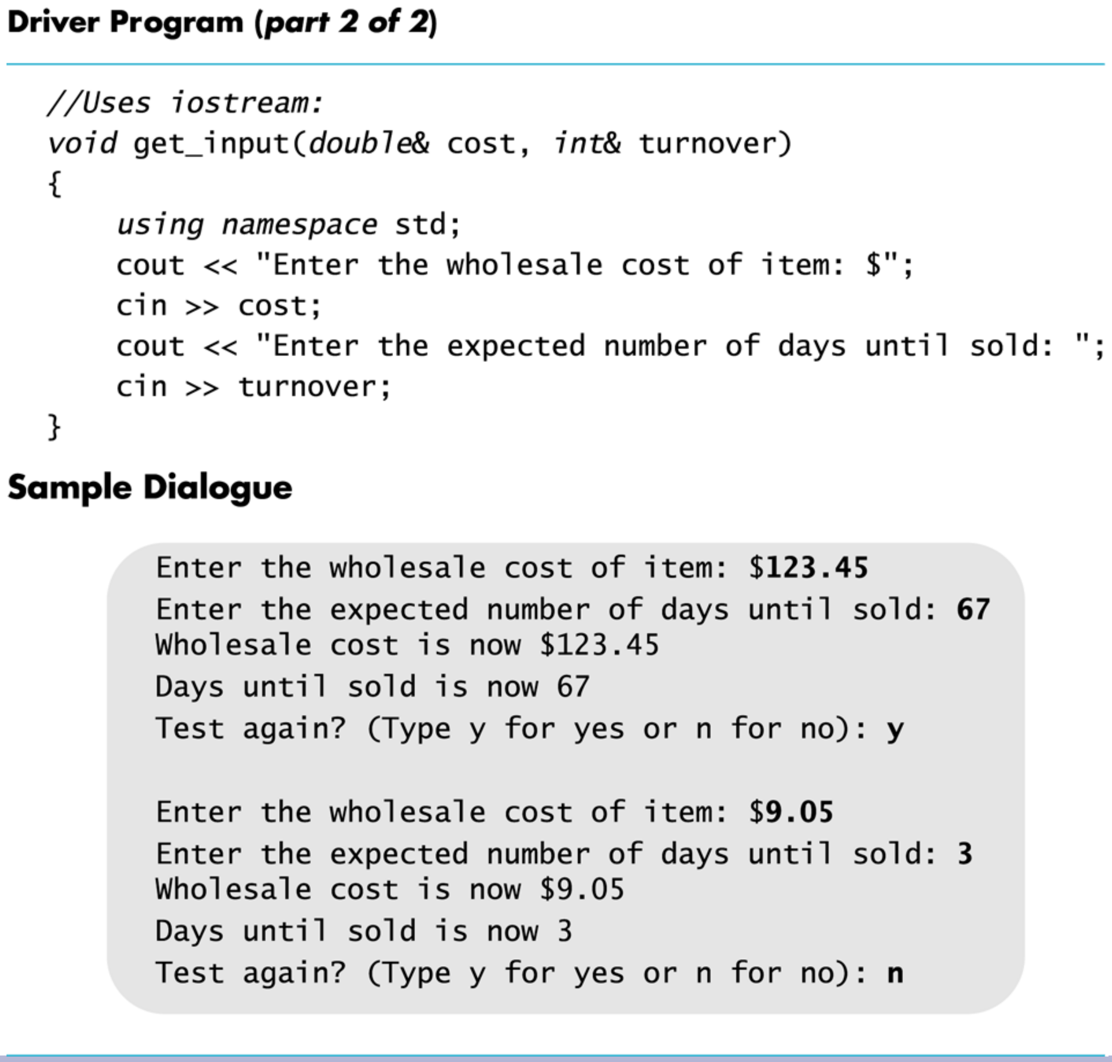
```
//Driver program for the function get_input.
#include <iostream>

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

int main( )
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 5.10 (2/2)

Back

Next



Driver Program (*part 2 of 2*)

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}
```

Sample Dialogue

```
Enter the wholesale cost of item: $123.45
Enter the expected number of days until sold: 67
Wholesale cost is now $123.45
Days until sold is now 67
Test again? (Type y for yes or n for no): y

Enter the wholesale cost of item: $9.05
Enter the expected number of days until sold: 3
Wholesale cost is now $9.05
Days until sold is now 3
Test again? (Type y for yes or n for no): n
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Stubs

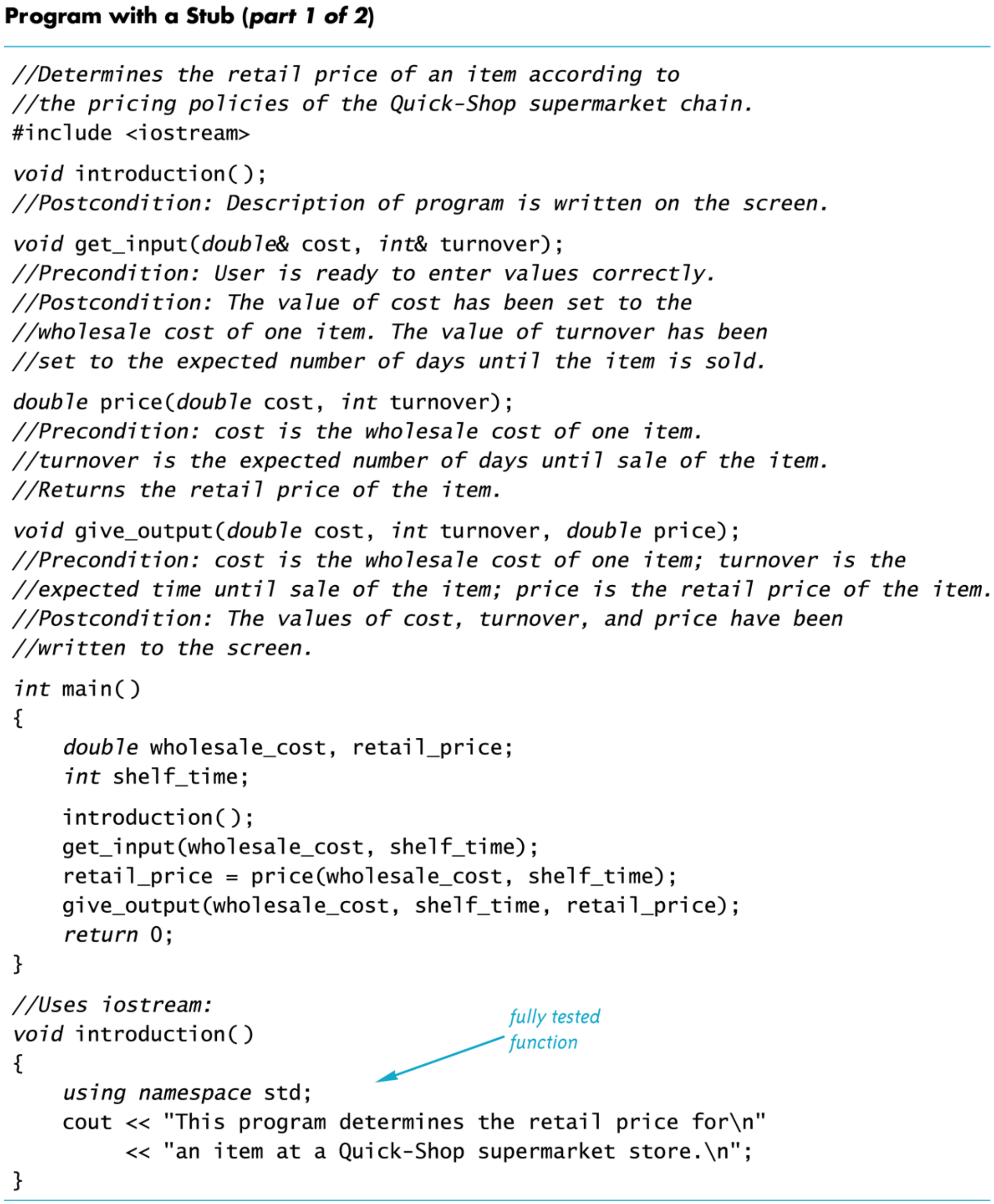
- When a function being tested calls other functions that are not yet tested, use a stub
- A stub is a simplified version of a function
  - Stubs usually provide values for testing rather than perform the intended calculation
  - Stubs should be so simple that you have confidence they will perform correctly
  - Function price is used as a stub to test the rest of the supermarket pricing program in

Display 5.11 (1) and **Display 5.11 (2)**

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 5.11 (1/2)

Back Next



Program with a Stub (*part 1 of 2*)

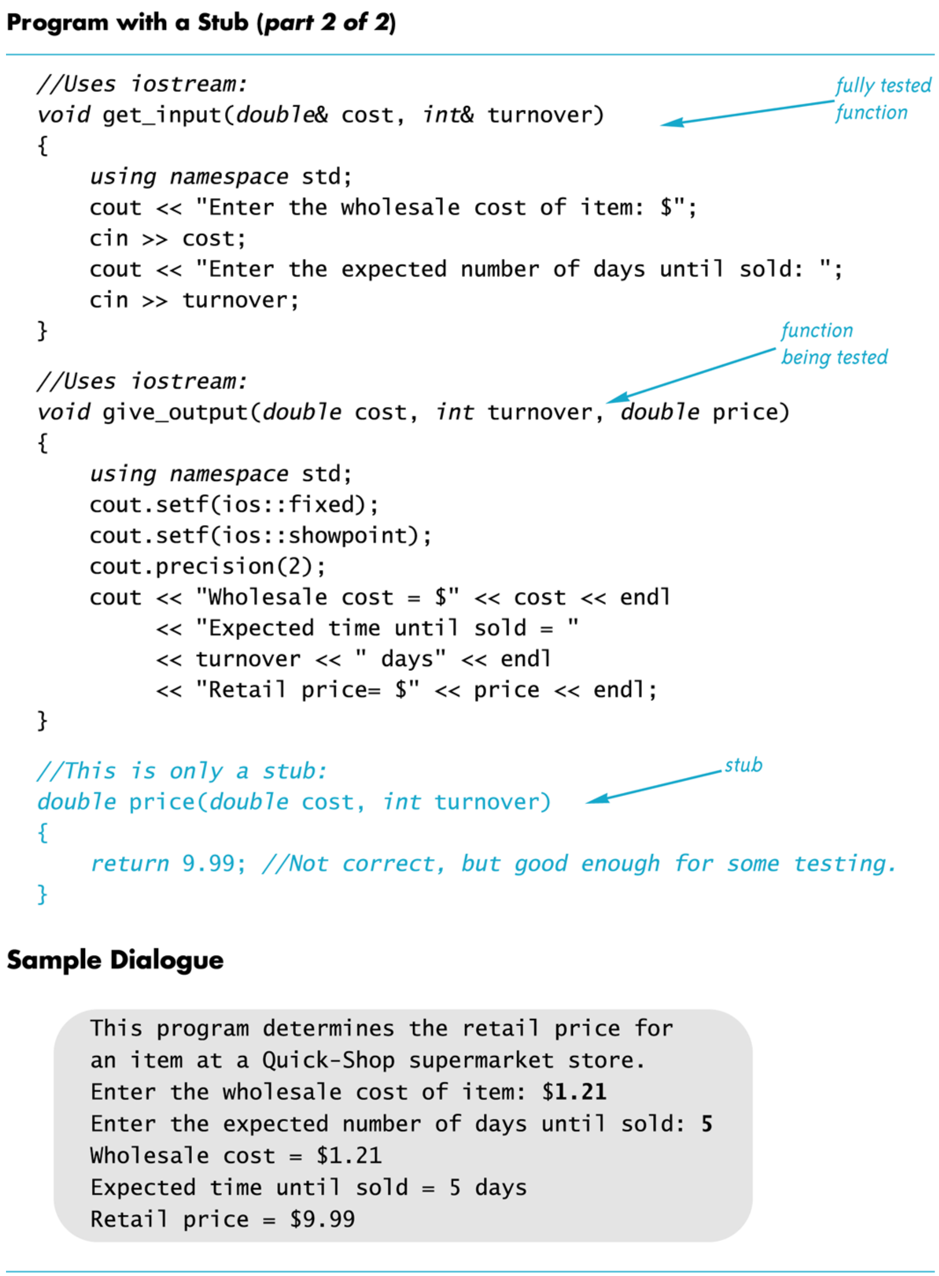
```
//Determines the retail price of an item according to
//the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

void introduction();
//Postcondition: Description of program is written on the screen.

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//turnover is the expected number of days until sale of the item.
//Returns the retail price of the item.

void give_output(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item; turnover is the
//expected time until sale of the item; price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price have been
//written to the screen.

int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}

//Uses iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}
```

fully tested function

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 5.11 (2/2)

Back Next



Program with a Stub (*part 2 of 2*)

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}

//Uses iostream:
void give_output(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price= $" << price << endl;
}

//This is only a stub:
double price(double cost, int turnover)
{
    return 9.99; //Not correct, but good enough for some testing.
}
```

fully tested function

function being tested

stub

Sample Dialogue

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $9.99
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Rule for Testing Functions

- Fundamental Rule for Testing Functions
  - Test every function in a program in which every other function in that program has already been fully tested and debugged.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

General Debugging Techniques

- Use a debugger
  - Tool typically integrated with a development environment that allows you to stop and step through a program line-by-line while inspecting variables

- The assert macro
  - Can be used to test pre or post conditions

    #include <cassert>

    assert(boolean expression)
  - If the boolean is false then the program will abort

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.