C++ Lecture 6

* void Functions, Reference Parameters, and Function Design
* CIS 251 • Shelby-Hoover Campus

void Functions

* All of the programmer-defined functions in chapter 4 return a value
  + The return type in the function prototype and the function header matches the type of value to be returned
  + The body of the function definition must contain a return statement with a value that matches the return type
* If a function displays its result within the body rather than returning it to the calling function, it may not need to return a value
* A function that does not return a value must have a return type (in both the prototype and the function header) of void

void Function Example

* Prototype (choose one of the following; either form will work):  
    
  void showCelsius(double fTemp);  
  void showCelsius(double);
* Definition that displays equivalent temperature values with proper formatting:  
    
  void showCelsius(double fTemp)  
  {  
   double cTemp = 5.0 \* (fTemp – 32) / 9.0;  
   cout.setf(ios::fixed);  
   cout.setf(ios::showpoint);  
   cout.precision(1);  
   cout << fTemp << " degrees Fahrenheit is equivalent to\n" << cTemp  
   << " degrees Celsius.\n";  
  }

Calling a void Function

* When calling a function that returns a value, the function call must be embedded within another statement that indicates what should be done with the returned value
* A void function call is a standalone statement: it should not be placed within another statement
* Example:  
    
  showCelsius(212.0);

void Return Statements

* A void function does not require a return statement
* If a void function contains a return statement, it should be the key word return followed by a semicolon (no value to return):   
    
  return;
* Some programmers prefer to include a return statement at the end of each void function body
* Other programmers use a return statement in a void function only when a particular condition is met that should cause the function to terminate early (not structured programming)

void Return Example

* A return statement to prevent division by zero:  
    
  void showPortionSize(int people, double totalOunces)  
  {  
   double portion;  
    
   if (people == 0)  
   {  
   cout << "Error! Zero people specified." << endl;  
   return; // Early exit from function  
   }  
    
   portion = totalOunces / people;  
    
   cout.setf(ios::fixed);  
   cout.setf(ios::showpoint);  
   cout.precision(2);  
   cout << "Each person receives " << portion << " ounces of ice cream."  
   << endl;  
  }

Return Statement Limits

* A value-returning function has a single return type
* The return statement in a value-returning function can only return a single value
* If a function needs to communicate several values back to the calling function, it cannot do so using the return statement

Call / Pass by Reference

* Parameters are pass by value by default
  + The parameter receives a copy of the argument
  + Changes to the parameter do not affect the argument
* A special type of parameter, a **reference parameter**, receives the argument’s memory address instead of a copy of its value
  + Changes made to a reference parameter are also applied to the argument in the calling function
  + If a function has several reference parameters, it can communicate several values back to the calling function

Reference Parameter Syntax

* Both the prototype and the header of a function definition must indicate that a parameter is a reference parameter by adding a single ampersand (&) after the parameter type:  
    
  returnType functionName(pType& pName);
* The call to such a function does not include the ampersand
* The argument to each reference parameter must be a variable, not a named constant or a literal (a literal value has no address, and a named constant’s value cannot be changed)

Call by Reference Example

* Prototype (choose one of the following; either form will work):  
    
  void addOneScore(double& sum, int& count);  
  void addOneScore(double&, int&);
* Definition:  
    
  void addOneScore(double& sum, int& count)  
  {  
   double newScore;  
    
   count++;  
   cout << "Enter score #" << count << ": ";  
   cin >> newScore;  
   sum += newScore;  
  }
* Call:  
    
  double gradeSum = 0.0;  
  int gradeCount = 0;  
    
  for (int i = 1; i <= 10; i++)  
   addOneScore(gradeSum, gradeCount);

Using Reference Parameters

* A function may change the value of a reference parameter by obtaining input for it or by assigning it a new value
* The prototype and header of a function may include a mixture of value and reference parameters (the value parameters omit the ampersand)
* A function with reference parameters may still return a value via the return type and return statement
* A parameter that should not be allowed to change the value of the argument in the calling function should remain pass-by-value, not pass-by-reference

Mixed Parameter Example

* Suppose a program contains a function with the following prototype:  
    
  void changeThem(int p1, int& p2);
* The function definition is as follows:  
    
  void changeThem(int p1, int& p2)  
  {  
   p1 = 5;  
   cout << "Parameter 1: " << p1 << endl;  
   p2 = 7;  
   cout << "Parameter 2: " << p2 << endl;  
  }

Mixed Parameter Call

* Suppose the main function calls changeThem as follows:  
    
  #include <iostream>  
  using namespace std;  
    
  void changeThem(int p1, int& p2);   
    
  int main()  
  {  
   int a1 = 9, a2 = 4;  
    
   cout << "Before the Call:" << endl;  
   cout << "Argument 1: " << a1 << endl;  
   cout << "Argument 2: " << a2 << endl;  
    
   changeThem(a1, a2);  
    
   cout << "After the Call:" << endl;  
   cout << "Argument 1: " << a1 << endl;  
   cout << "Argument 2: " << a2 << endl;  
    
   return 0;  
  }  
    
  // function definition goes here

Mixed Parameter Output

* The main function displays the values before the function call:  
    
  Before the Call:  
  Argument 1: 9  
  Argument 2: 4
* The call to the function changeThem displays the newly assigned parameter values:  
    
  Parameter 1: 5  
  Parameter 2: 7
* In main, the first argument (pass-by-value) retains its original value, but the second argument (pass-by-reference) receives the new value from the parameter:  
    
  After the Call:  
  Argument 1: 9  
  Argument 2: 7

Procedural Abstraction

* Any function can call another function
* The function prototype must appear before the function call so that the compiler can evaluate whether or not the call matches the definition
* **A function never contains another function’s definition** (the function definitions are independent, not nested)

Pre- and Postconditions

* In writing functions, it’s helpful to describe the states of the program before and after the function executes
* A **precondition** is a condition that is assumed to be true when the function begins (usually that the parameters have received specific argument values in the call)
* A **postcondition** is a condition that describes the effect of the function call (the value returned, the output generated, or the new values in the reference parameters once the function’s execution is complete)
* Preconditions and postconditions are typically listed in comments below the function prototype or above the function header

Condition Examples

* Example:  
  void addOneScore(double& sum, int& count);  
  /\* Preconditions: sum and count have existing values representing the total and count of grades entered thus far. The user is ready to enter an additional score. \*/  
  /\* Postconditions: The value entered by the user has been added to sum, and count has increased by 1. \*/

Program Testing

* It is impossible to test every possible value for every input requested by a program
* It is helpful to test **boundary values** (values for which the behavior changes) as well as values in the middle of expected ranges
* Example: if the output expected from a program should differ for input values between 0 and 7 versus input values between 8 and 12
  + Test the boundary values 0, 7, 8, and 12
  + Test at least one value between 0 and 7
  + Test at least one value between 8 and 12

Testing Functions

* A function may need to be tested in isolation before it is integrated into the program where it is used
  + A **driver** program calls the function with various arguments to verify that the results are consistent with what is expected
  + Separate from the final program
* To test a program before an individual function has been written, you may replace it with a **stub**
  + Same prototype and header as the real function
  + Behavior is simplified (display the parameter values to ensure they are received, return the same literal value every time)

Debugging a Program

* Visual Studio and many other programming environment provide a **debugger** that allows a programmer to see the changes in variables and on the screen statement by statement
* A simple debugging technique is to add extra output statements at crucial points in the program
  + Verifying that the program executes particular blocks (decision statements, loops, function calls)
  + Seeing the values in important variables
  + Don’t forget to remove these statements from the final program!
* The assert macro (provided by the library <cassert>) prevents a program from continuing execution if a Boolean expression evaluates to false
  + Prevents execution when preconditions are not met
  + Add the line #define NDEBUG to disable these macros