Functions

Prof. Seokin Hong

Agenda

- Predefined Functions
- Programmer-Defined Functions
- Procedural Abstraction
- Local and Global Variables
- Overloading Function Names
- Call-By-Reference Parameters
- Testing and Debugging

Predefined Functions

Function Libraries

- Predefined functions are found in libraries
- The library must be “included” in a program to make the functions available
- An include directive tells the compiler which library header file to include

- To include the math library containing **sqrt():**

```
#include <cmath>
```

- Newer standard libraries, such as cmath, also require the directive
  using namespace std;

A Predefined Function

- **Example: sqrt function**
  - theRoot = **sqrt(9.0)**;
  - Computes and returns the square root of a number
  - The number, 9, is called the argument
    - theRoot will contain 3.0

Function Calls

- **sqrt(9.0) is a function call**
  - It invokes, or sets in action, the sqrt function
  - The argument (9) can also be a variable or an expression

- **A function call can be used like any expression**
  - bonus = sqrt(sales) / 10;
  - Cout << “The side of a square with area “ << area
    << “ is “
    << sqrt(area);

Function Call Syntax

- **Function_name (Argument_List)**
  - Argument_List is a comma separated list:

    (Argument_1, Argument_2, … , Argument_Last)

- **Example:**
  - side = sqrt(area);
  - cout << “2.5 to the power 3.0 is “
    << pow(2.5, 3.0);

Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	square root	*double*	*double*	sqrt(4.0)	2.0	cmath
pow	powers	*double*	*double*	pow(2.0,3.0)	8.0	cmath
abs	absolute value for *int*	*int*	*int*	abs(-7) abs(7)	7 7	cstdlib
labs	absolute value for *long*	*long*	*long*	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	absolute value for *double*	*double*	*double*	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	ceiling (round up)	*double*	*double*	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	floor (round down)	*double*	*double*	floor(3.2) floor(3.9)	3.0 3.0	cmath
srand	Seed random number generator	*none*	*none*	srand()	none	cstdlib
rand	Random number	*none*	*int*	rand()	0-RAND_MAX	cstdlib

Random Number Generation

- **Step1:** Seed the random number generator only once

```
#include <cstdlib>
#include <ctime>

srand(time(0));
```

- **Step2:** The rand() function returns a random integer that is greater than or equal to 0 and less than RAND_MAX

```
rand();
```

Random Number Generation

- Use % and + to scale to the number range you want
- **Example:**
  - Generate a random number from 1-6 to simulate rolling a six-sided die:

```
int die = (rand() % 6) + 1;
```

- **Can you simulate rolling two dice?**
- **Generating a random number x where 10 < x < 21?**

Type Casting

- **Recall the problem with integer division:**

```
int totalCandy = 9, numberOfPeople = 4;
double candyPerPerson;
candyPerPerson = totalCandy / numberOfPeople;
```

candyPerPerson = 2, not 2.25!

- **A Type Cast produces a value of one type from another type**
  - **static_cast<double>**(totalCandy): produces a double representing the integer value of totalCandy

Type Casting (Cont.)

- **Example**

```
int totalCandy = 9, numberOfPeople = 4;
double candyPerPerson;
candyPerPerson = static_cast<double>(totalCandy) /  numberOfPeople;
```

candyPerPerson now is 2.25!

- This would also work:

```
candyPerPerson = totalCandy / static_cast<double>( numberOfPeople);
```

- This would not!

```
candyPerPerson = static_cast<double>( totalCandy / numberOfPeople);
```

Programmer-Defined Functions

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
}
```

The Return Statement

- **Ends the function call**
- **Returns the value calculated by the function**
- **Syntax:**

  return expression;

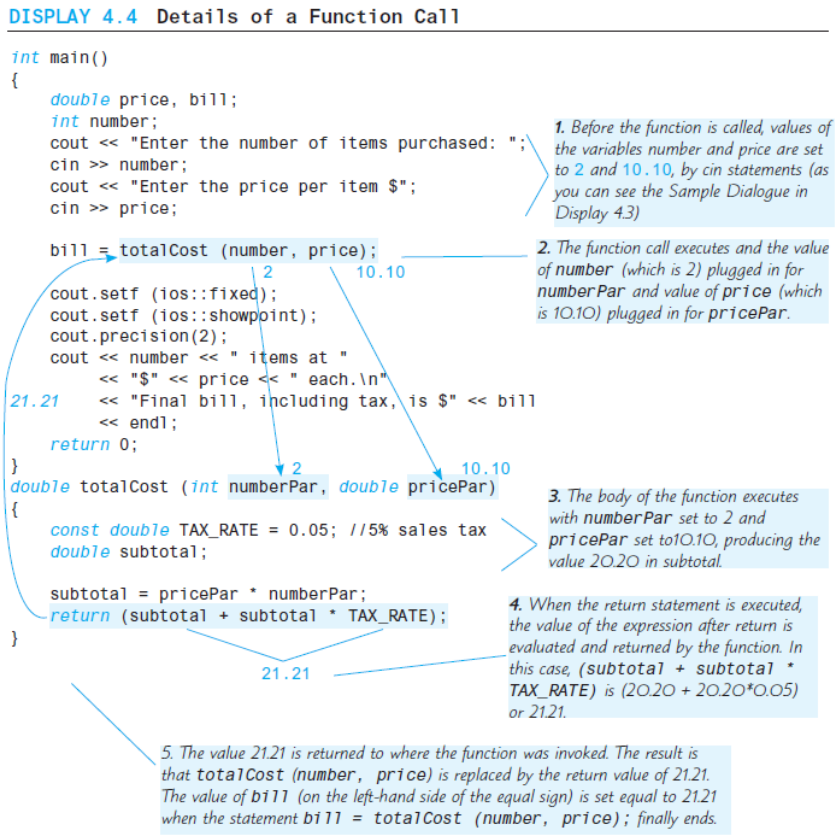
  - expression performs the calculation
    or
  - expression is a variable containing the calculated value

- **Example:**

  return subtotal + subtotal * TAX_RATE;

Function Call Details

- **The values of the arguments are plugged into the formal parameters (Call-by-value mechanism)**
  - The first argument is used for the first formal parameter
  - the second argument for the second formal parameter, and so forth
  - The value plugged into the formal parameter is used in all instances of the formal parameter in the function body



Alternate Declarations

- **Two forms for function declarations**
  - List **type** and **name** of formal parameters
  - List **type** of formal parameters, but not name

- **Examples:**

```
double totalCost(int numberPar, double pricePar);

double totalCost(int, double);
```

bool Return Values

- **A function can return a bool value**
  - Such a function can be used where a boolean expression is expected
    - Makes programs easier to read

- **if (((rate >=10) && (rate < 20)) || (rate == 0))**

  is easier to read as

  if (appropriate (rate))

appropriate could be defined as

```
bool appropriate(int rate)
{
  return (((rate >=10) && ( rate < 20)) || (rate == 0));
}
```

Procedural Abstraction

Procedural Abstraction

- A programmer who uses a function needs to know what the function does, *not how it does it*

- To know how to use a function simply by reading the function declaration and its comment

- **Information Hiding**
  - The function can be used without knowing how it is coded
  - The function body can be “hidden from view”
  - To change or improve a function definition without forcing programmers using the function to change what they have done

Procedural Abstraction and Functions

- **Write functions so the declaration and comment are all a programmer needs to use the function**
  - Function comment should tell all conditions required of arguments to the function
  - Function comment should describe the returned value
  - Variables used in the function, other than the formal parameters, should be declared in the function body

Case Study: Buying Pizza

- **What size pizza is the best buy?**
  - Which size gives the lowest cost per square inch?
  - Pizza sizes given in diameter
  - Quantity of pizza is based on the area which is proportional to the square of the radius

Case Study: Buying Pizza (Cont.)

- **Buying Pizza Problem Definition**
  - **Input**:
    - Diameter of two sizes of pizza
    - Cost of the same two sizes of pizza
  - **Output**:
    - Cost per square inch for each size of pizza
    - Which size is the best buy
      - Based on lowest price per square inch
      - If cost per square inch is the same, the smaller size will be the better buy

Case Study: Buying Pizza (Cont.)

- **Subtask 1**
  - Get the input data for each size of pizza
- **Subtask 2**
  - Compute price per inch for smaller pizza
- **Subtask 3**
  - Compute price per inch for larger pizza
- **Subtask 4**
  - Determine which size is the better buy
- **Subtask 5**
  - Output the results

Case Study: Buying Pizza (Cont.)

- **Subtask 2 and subtask 3 should be implemented as a single function**
  - Subtask 2 and subtask 3 are identical tasks
  - Subtask 2 and subtask 3 each return a single value

- **Choose an appropriate name for the function**
  - We'll use unitprice

Case Study: Buying Pizza (Cont.)

- **double unitprice(int diameter, int double price);**
  // Returns the price per square inch of a pizza
  // The formal parameter named ***diameter*** is the diameter of the pizza in inches.
  // The formal parameter named ***price*** is the price of the pizza.

Case Study: Buying Pizza (Cont.)

- **Subtask 1**
  - Ask for the input values and store them in variables
    - diameterSmall diameterLarge
      priceSmall priceLarge

- **Subtask 4**
  - Compare cost per square inch of the two pizzas

- **Subtask 5**
  - Standard output of the results

Case Study: Buying Pizza (Cont.)

- **Buying Pizza unitprice Algorithm**
  - Subtasks 2 and 3 are implemented as calls to function unitprice
  - **unitprice algorithm**
    - Compute the radius of the pizza
    - Computer the area of the pizza using πr^2
    - Return the value of (price / area)
  - **unitprice Pseudocode**

```
radius = one half of diameter;
area = π * radius * radius
return (price / area)
```

Case Study: Buying Pizza (Cont.)

- **Buying Pizza First try at unitprice**
  - double unitprice (int diameter, double price)

```
{
    const double PI = 3.14159;
    double radius, area;

     radius = diameter / 2;
     area = PI * radius * radius;
     return (price / area);
}
```

  - Oops! Radius should include the fractional part

Case Study: Buying Pizza (Cont.)

- **Buying Pizza First try at unitprice**
  - double unitprice (int diameter, double price)

```
{
    const double PI = 3.14159;
    double radius, area;

     radius = diameter / static_cast<double>(2) ;       // radius = diameter / 2.0 ;
     area = PI * radius * radius;
     return (price / area);
}
```

void-Functions

- **A subtask might produce**
  - No value (just input or output for example)
  - One value
  - More than one value
- **A void-function implements a subtask that returns no value or more than one value**

```
void showResults(double f_degrees, double c_degrees)
 {
      using namespace std;
      cout << f_degrees
              << “ degrees Fahrenheit is equivalent to “ << endl
              << c_degrees << “ degrees Celsius.” << endl;
       return;
}
```

Why Use a Return in void-functions?

- **Optional return statement ends the function**
  - Return statement can include no value to return
- **Return statement is implicit if it is not included**
- What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?

DISPLAY 5.3 Use of *return* in a *void* Function

Function Declaration

```
1   void iceCreamDivision(int number, double totalWeight);
2   //Outputs instructions for dividing totalWeight ounces of
3   //ice cream among number customers.
4   //If number is 0, nothing is done.
```

Function Definition

```
1   //Definition uses iostream:
2   void iceCreamDivision(int number, double totalWeight)
3   {
4       using namespace std;
5       double portion;
6
7       if (number == 0)
8           return;
9       portion = totalWeight/Number;
10      cout.setf(ios::fixed);
11      cout.setf(ios::showpoint);
12      cout.precision(2);
13      cout << "Each one receives "
14           << portion << " ounces of ice cream." << endl;
15  }
```

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

Local and Global Variables

Local Variables

- **Variables declared in a function:**
  - Are local to that function, they cannot be used from outside the function
  - Have the function as their scope

- **Variables declared in the main part of a program:**
  - Are local to the main part of the program, they cannot be used from outside the main part
  - Have the main part as their scope

```
1   //Computes the average yield on an experimental pea growing patch.
2   #include <iostream>
3   using namespace std;
4
5   double estTotal(int minPeas, int maxPeas, int podCount);
6   //Returns an estimate of the total number of peas harvested.
7   //The formal parameter podCount is the number of pods.
8   //The formal parameters minPeas and maxPeas are the minimum
9   //and maximum number of peas in a pod.
10
11  int main( )                                   This variable named averagePea is
12  {                                             local to the main part of the program.
13      int maxCount, minCount, podCount;
14      double averagePea, yield;
15
16      cout << "Enter minimum and maximum number of peas in a pod: ";
17      cin >> minCount >> maxCount;
18      cout << "Enter the number of pods: ";
19      cin >> podCount;
20      cout << "Enter the weight of an average pea (in ounces): ";
21      cin >> averagePea;
22
23      yield =
24          estTotal(minCount, maxCount, podCount) * averagePea;
25
26      cout.setf(ios::fixed);
27      cout.setf(ios::showpoint);
28      cout.precision(3);
29      cout << "Min number of peas per pod = " << minCount << endl
30           << "Max number of peas per pod = " << maxCount << endl
31           << "Pod count = " << podCount << endl
32           << "Average pea weight = "
33           << averagePea << " ounces" << endl
34           << "Estimated average yield = " << yield << " ounces"
35           << endl;
36
37      return 0;
38  }
39
40  double estTotal(int minPeas, int maxPeas, int podCount)
41  {
42      double averagePea;                        This variable named averagePea
43                                                is local to the function estTotal.
44      averagePea = (maxPeas + minPeas)/2.0;
45      return (podCount * averagePea);
46  }
```

Global Constants

- **Global Named Constant**
  - Available to more than one function as well as the main part of the program
  - Declared outside any function body
  - Declared outside the main function body
  - Declared before any function that uses it

- **Example:** **const double PI = 3.14159;**
  int main()
  {…}
  - PI is available to the main function and to function volume

A Global Named Constant ***(part 1 of 2)***

```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main()
{
    double radius_
    cout << "Enter
         << "and a
    cin >> radius_
    area_of_circle
    volume_of_spher
    cout << "Radius
         << "Area
         << " squar
         << "Volume
         << " cubi
    return 0;
}
```

A Global Named Constant ***(part 2 of 2)***

```
double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Sample Dialogue

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

Global Variables

- **Global Variable**
  - **rarely used** when more than one function must use a common variable
  - Declared just like a global constant except const is not used
  - Generally make programs more difficult to understand and maintain

Formal Parameters are Local Variables

- **Formal Parameters are actually variables that are local to the function definition**
  - 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

- **The call-by-value mechanism**
  - When a function is called, the formal parameters are initialized to the values of the arguments in the function call

Formal Parameter Used as a Local Variable (*part 1 of 2*)

```
//Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00; //Dollars per quarter hour.

double fee(int hours_worked, int minutes_worked);
//Returns the charges for hours_worked hours and
//minutes_worked minutes of legal services.

int main()
{
    int hours, minutes;
    double bill;

    cout << "Welcome to the offices of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}

double fee(int hours_worked, int minutes_worked)
{
    int quarter_hours;

    minutes_worked = hours_worked*60 + minutes_worked;
    quarter_hours = minutes_worked/15;
    return (quarter_hours*RATE);
}
```

The value of minutes *is not changed by the call to* fee.

minutes_worked *is a local variable initialized to the value of* minutes.

Block Scope

- **Local and global variables conform to the rules of Block Scope**
  - The code block (generally defined by the { }) where an identifier like a variable is declared determines the scope of the identifier

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*

Overloading Function Name

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**

Examples

```
double ave(double n1, double n2)
{
        return ((n1 + n2) / 2);
}
double ave(double n1, double n2, double n3)
{
    return (( n1 + n2 + n3) / 3);
}
```

- Compiler checks the number and types of arguments in the function call to decide which function to use

```
cout << ave( 10, 20, 30);
```

uses the second definition

Overloading Details

- **Overloaded functions**
  - Must have different numbers of formal parameters

    AND / OR

  - Must have at least one different type of parameter
  - Different **return** type is not sufficient for overloading

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;
}

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

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

two arguments

three arguments

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
```

Overloading Example

- **Revising the Pizza Buying program**
  - Rectangular pizzas are now offered!
  - Change the input and add a function to compute the unit price of a rectangular pizza
  - The new function could be named unitprice_rectangular
  - Or, the new function could be a new (overloaded) version of the unitprice function that is already used

- Example:

```
double unitprice(int length, int width, double price)
{
        double area = length * width;
        return (price / area);
}
```

Automatic Type Conversion

- **Given the definition**

```
double mpg(double miles, double gallons)
{
    return (miles / gallons);
}
```

- **what will happen if mpg is called in this way?**

```
cout << mpg(45, 2) << " miles per gallon";
```

- **The values of the arguments will automatically be converted to type double (45.0 and 2.0)**

Type Conversion Problem

- **Given the previous mpg definition and the following definition in the same program**

```
int mpg(int goals, int misses) // returns the Measure of Perfect Goals
{
    return (goals – misses);
}
```

- **what happens if mpg is called this way now?**

  cout << mpg(45, 2) << “ miles per gallon”;

  - The compiler chooses the function that matches parameter types so the Measure of Perfect Goals will be calculated

Call-by-Reference Parameters

Call-by-Reference Parameters

- **Call-by-value is not adequate when we need a sub-task to obtain input values**
  - Call-by-value means that the formal parameters receive the values of the arguments
  - Recall that we have changed the values of formal parameters in a function body, but we have not changed the arguments found in the function call

- **Call-by-reference parameters allow us to change the variable used in the function call**

Call-by-Reference Example

```
void getInput(double& fVariable)
 {
      using namespace std;
      cout << " Convert a Fahrenheit temperature"
                << " to Celsius.\n"
           << " Enter a temperature in Fahrenheit: ";
       cin >> fVariable;
}
```

- **'&' symbol (ampersand) identifies fVariable as a call-by-reference parameter**
  - Used in both declaration and definition!

DISPLAY 5.4 Call-by-Reference Parameters

```
1  //Program to demonstrate call-by-reference parameters.
2  #include <iostream>
3  void getNumbers(int& input1, int& input2);
4  //Reads two integers from the keyboard.
5  void swapValues(int& variable1, int& variable2);
6  //Interchanges the values of variable1 and variable2.
7  void showResults(int output1, int output2);
8  //Shows the values of variable1 and variable2, in that order.
9    int main( )
10 {
11     int firstNum = 0, secondNum = 0;
12
13     getNumbers(firstNum, secondNum);
14     swapValues(firstNum, secondNum);
15     showResults(firstNum, secondNum);
16       return 0;
17 }
18   //Uses iostream:
19   void getNumbers (int& input1, int& input2)
20 {
21       using namespace std;
22     cout << "Enter two integers: ";
23     cin >> input1
24         >> input2;
25 }
26   void swapValues(int& variable1, int& variable2)
27 {
28       int temp;
29     temp = variable1;
30     variable1 = variable2;
31     variable2 = temp;
32 }
33   //Uses iostream:
34   void showResults(int output1, int output2)
35 {
36       using namespace std;
37       cout << "In reverse order the numbers are: "
38           << output1 << " " << output2 << endl;
39 }
```

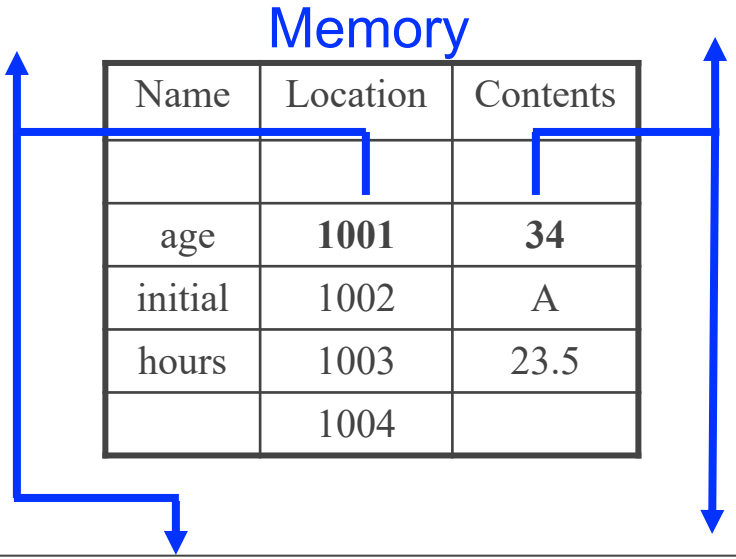
Call-by-Reference Details

- **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-reference

The function call:
f(age);

void f(int& ref_par);



Call-by-value

The function call:
f(age);

void f(int var_par);

Example: swapValues

```
void swap(int& variable1, int& variable2)
{
    int temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Mixed Parameter Lists

- **Call-by-value and call-by-reference parameters can be mixed in the same function**

- **Example:**
  void goodStuff(int& par1, int par2, double& par3);
  - par1 and par3 are call-by-reference formal parameters
    - Changes in par1 and par3 change the argument variable
  - par2 is a call-by-value formal parameter
    - Changes in par2 do not change the argument variable

Testing and Debugging

Testing and Debugging Functions

- Each function should be tested as a separate unit
  - Driver programs allow testing of individual functions

DISPLAY 5.10 Driver Program

```
1   //Driver program for the function getInput.
2   #include <iostream>
3
4   void getInput(double& cost, int& turnover);
5   //Precondition: User is ready to enter values correctly.
6   //Postcondition: The value of cost has been set to the
7   //wholesale cost of one item. The value of turnover has been
8   //set to the expected number of days until the item is sold.
9
10  int main( )
11  {
12      using namespace std;
13      double wholesaleCost;
14      int shelfTime;
15      char ans;
16
17      cout.setf(ios::fixed);
18      cout.setf(ios::showpoint);
19      cout.precision(2);
20      do
21      {
22          getInput(wholesaleCost, shelfTime);
23
24          cout << "Wholesale cost is now $"
25               << wholesaleCost << endl;
26          cout << "Days until sold is now "
27               << shelfTime << endl;
28
29          cout << "Test again?"
30               << " (Type y for yes or n for no): ";
31          cin >> ans;
32          cout << endl;
33      } while (ans == 'y' || ans == 'Y');
34
35      return 0;
36  }
```

```
37  //Uses iostream:
38  void getInput(double& cost, int& turnover)
39  {
40      using namespace std;
41      cout << "Enter the wholesale cost of item: $";
42      cin >> cost;
43      cout << "Enter the expected number of days until sold: ";
44      cin >> turnover;
45  }
```

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** are usually provide values for testing rather than perform the intended calculation
  - Stubs should be so simple that you have confidence they will perform correctly

DISPLAY 5.11 Program with a Stub *(part 1 of 2)*

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

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

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

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

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

20  int main( )
21  {
22      double wholesaleCost, retailPrice;
23      int shelfTime;

24      introduction( );
25          getInput(wholesaleCost, shelfTime);
26      retailPrice = price(wholesaleCost, shelfTime);
27      giveOutput(wholesaleCost, shelfTime, retailPrice);
28        return 0;
29  }
```

Stubs (Cont.)

- 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** are usually provide values for testing rather than perform the intended calculation
  - Stubs should be so simple that you have confidence they will perform correctly

```
30  //Uses iostream:
31  void introduction( )                                   <-- fully tested function
32  {
33      using namespace std;
34      cout << "This program determines the retail price for\n"
35           << "an item at a Quick-Shop supermarket store.\n";
36  }
37  //Uses iostream:                                       <-- fully tested function
38  void getInput(double& cost, int& turnover)
39  {
40      using namespace std;
41      cout << "Enter the wholesale cost of item: $";
42      cin >> cost;
43      cout << "Enter the expected number of days until sold: ";
44      cin >> turnover;
45  }
                                                           <-- function being tested
46  //Uses iostream:
47  void giveOutput(double cost, int turnover, double price)
48  {
49      using namespace std;
50      cout.setf(ios::fixed);
51      cout.setf(ios::showpoint);
52      cout.precision(2);
53      cout << "Wholesale cost = $" << cost << endl
54           << "Expected time until sold = "
55           << turnover << " days" << endl
56           << "Retail price= $" << price << endl;
57  }
                                                           <-- stub
58  //This is only a stub:
59  double price(double cost, int turnover)
60  {
61      return 9.99; //Not correct, but good enough for some testing.
62  }
```

General Debugging Techniques

- **Use a debugger (e.g., gdb)**
  - 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!!

Assert Example

```
// Approximates the square root of n using Newton's
// Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
    double answer = 1;
    int i = 0;

    assert((n > 0) && (num_iterations> 0));
    while (i <num_iterations)
    {
        answer = 0.5 * (answer + n / answer);
        i++;
    }
    return answer;
}
```

Copyright Notice

- The contents of this slide deck are taken from the textbook (Problem Solving with C++, Walter Savitch).

- See your textbook for more details.

- Redistribution of this slide deck is not permitted.

NEXT ?

I/O Streams