CHAPTER 4

PARAMETERS AND OVERLOADING

LEARNING OBJECTIVES

- Parameters
  - Call-by-value
  - **Call-by-reference**
  - Mixed parameter-lists
- Overloading and Default Arguments
  - Examples, Rules
- Testing and Debugging Functions
  - assert Macro
  - Stubs, Drivers

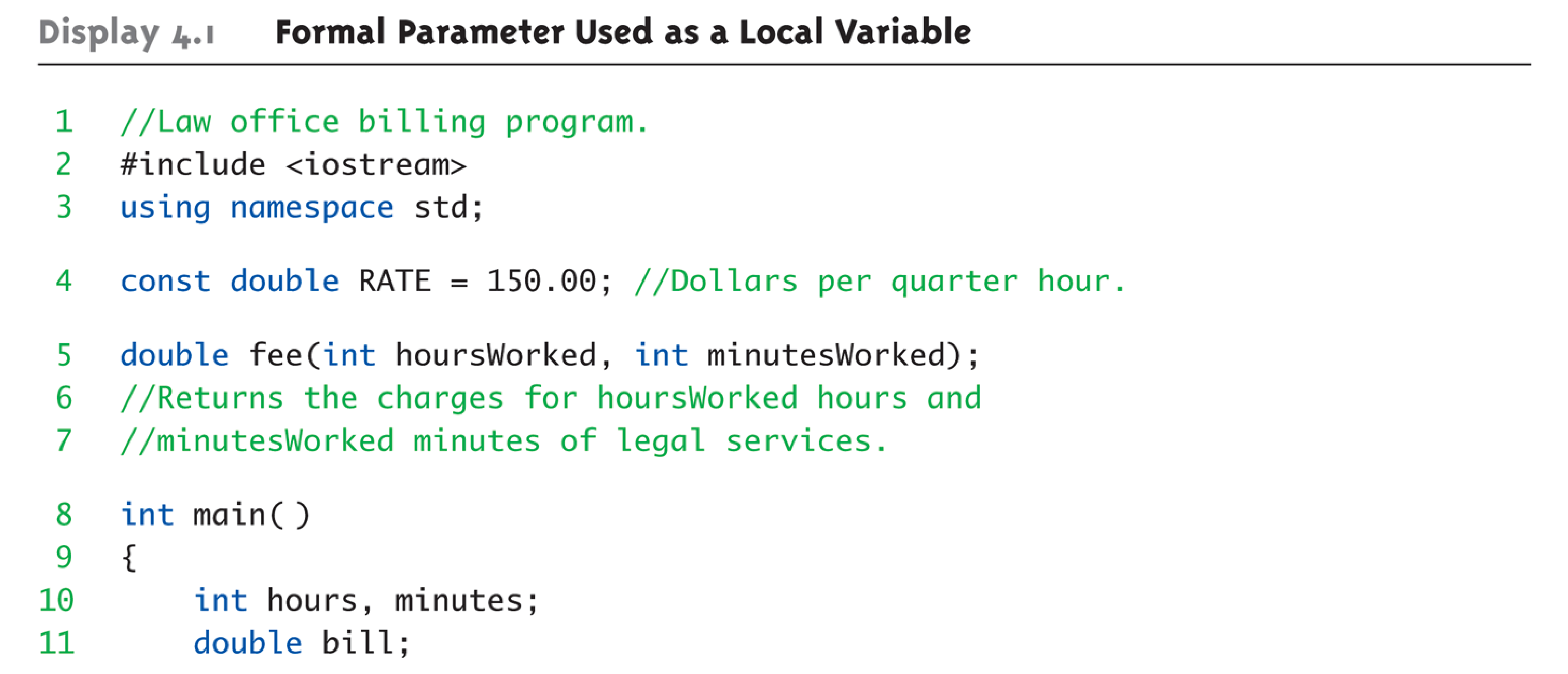
PARAMETERS

- Two methods of passing arguments as parameters
- Call-by-value
  - "copy" of value is passed
- Call-by-reference
  - "address of" actual argument is passed

CALL-BY-VALUE PARAMETERS

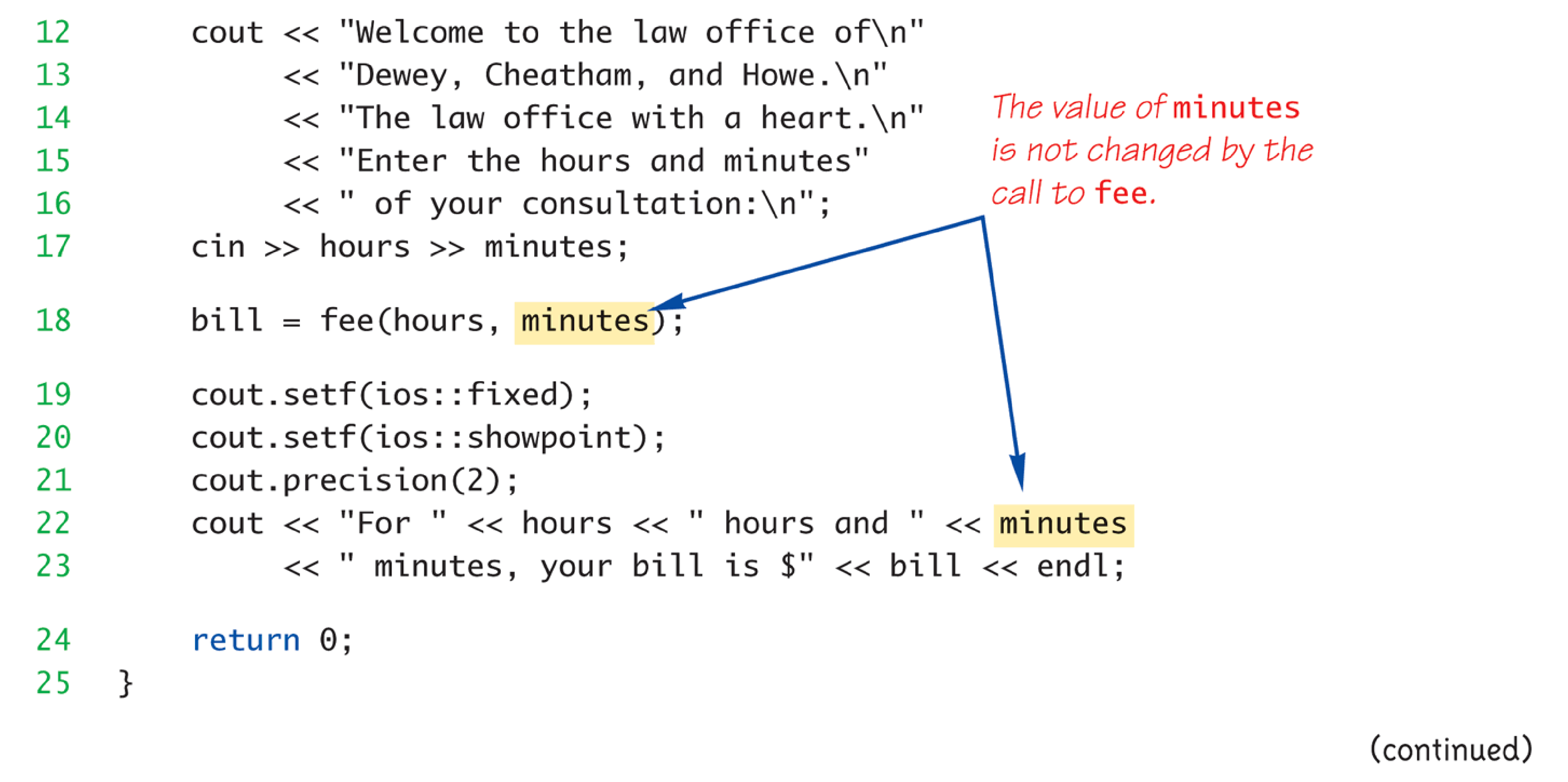
- Copy of actual argument passed
- Considered "local variable" inside function
- If modified, only "local copy" changes
  - Function has no access to “actual argument” from caller
- This is the default method
  - Used in all examples thus far

CALL-BY-VALUE EXAMPLE:
DISPLAY 4.1 FORMAL PARAMETER USED AS A LOCAL VARIABLE



Display 4.1 **Formal Parameter Used as a Local Variable**

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

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

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

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

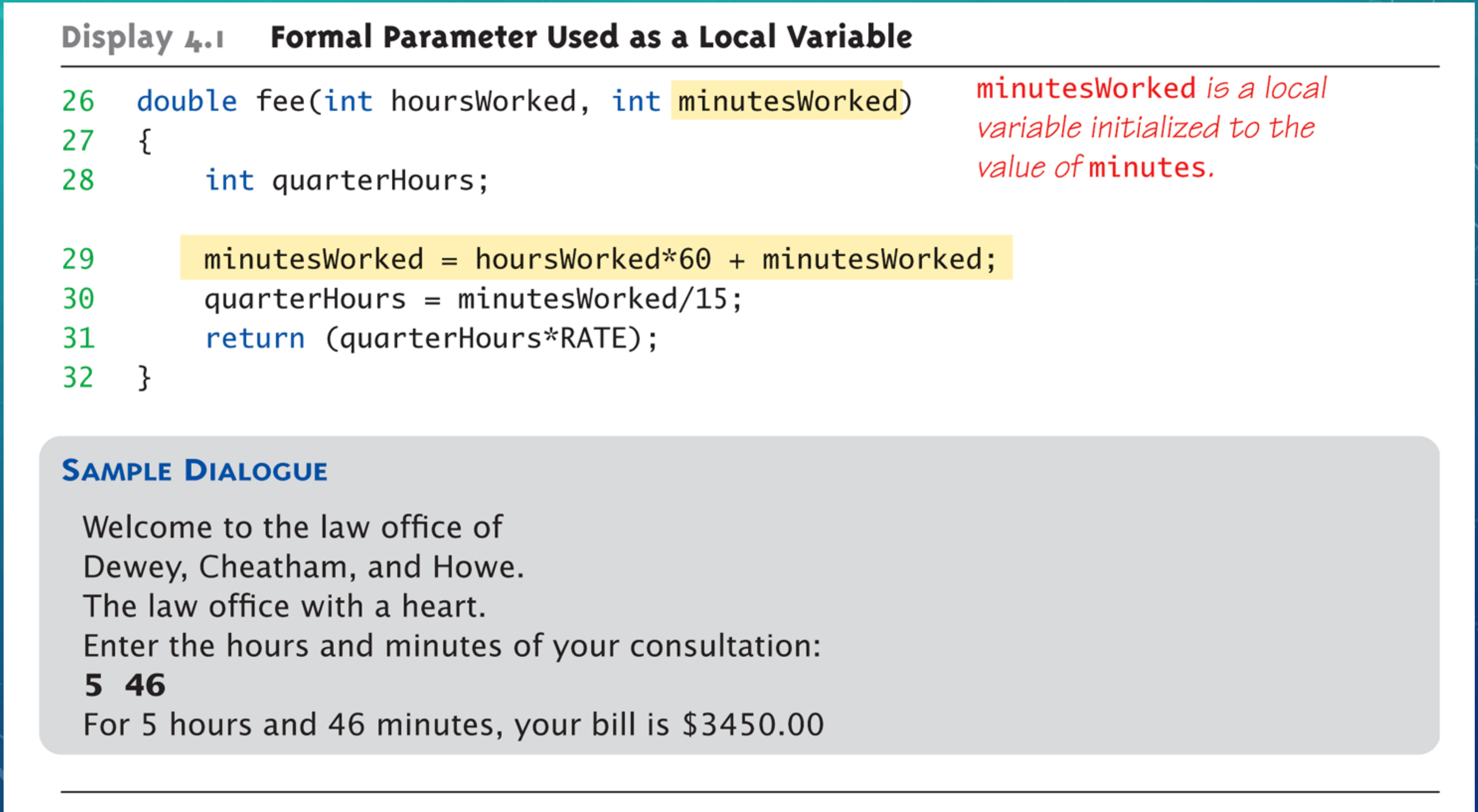


```
    cout << "Welcome to the law office of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    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;
}
```

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

(continued)

CALL-BY-VALUE EXAMPLE: **DISPLAY 4.1** FORMAL PARAMETER USED AS A LOCAL VARIABLE (3 OF 3)



Display 4.1 **Formal Parameter Used as a Local Variable**

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;

    minutesWorked = hoursWorked*60 + minutesWorked;
    quarterHours = minutesWorked/15;
    return (quarterHours*RATE);
}
```

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

Sample Dialogue

```
Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5  46
For 5 hours and 46 minutes, your bill is $3450.00
```

CALL-BY-VALUE PITFALL

- Common Mistake:
  - Declaring parameter "again" inside function:

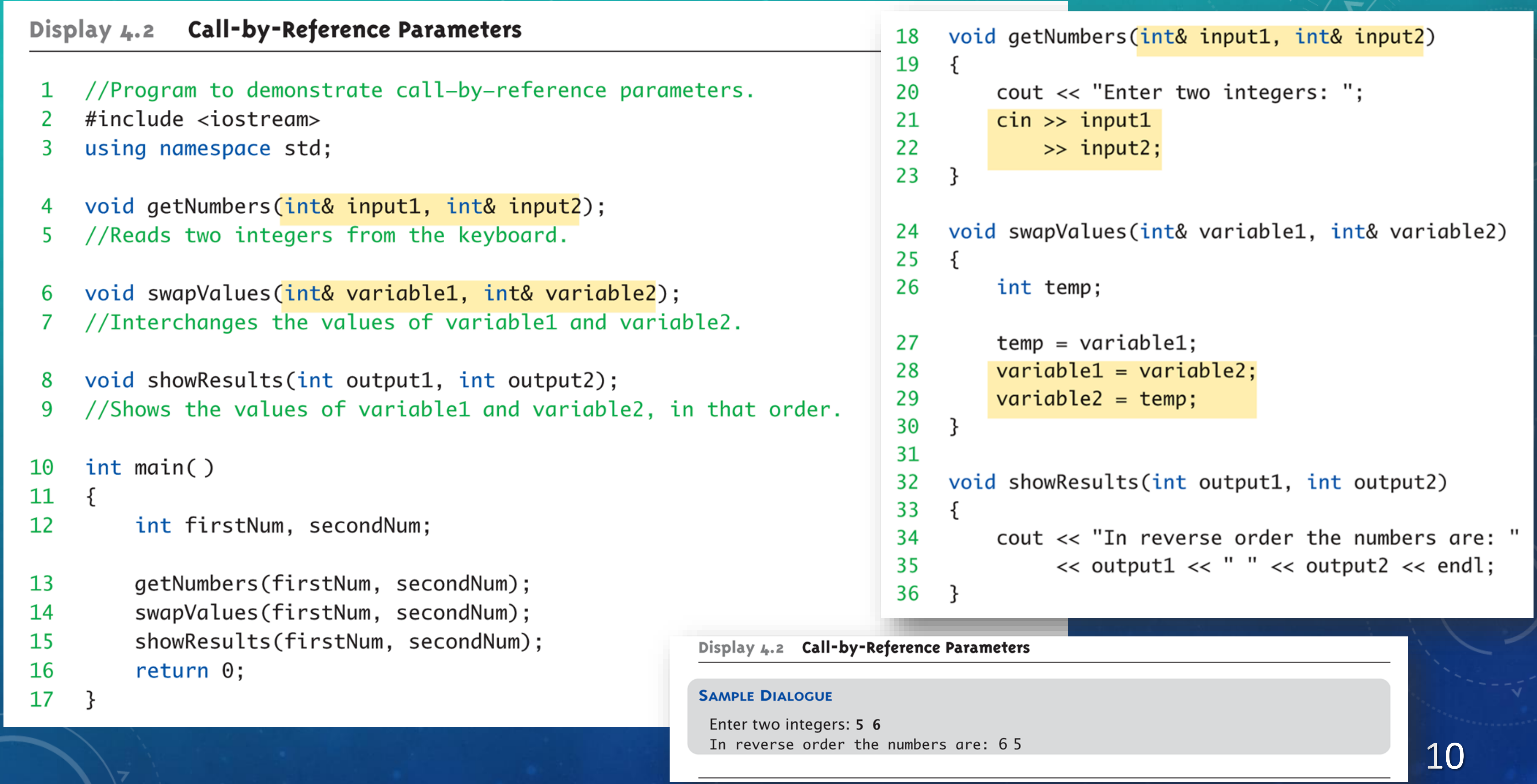
```
double fee(int hoursWorked, int minutesWorked)
{
  int quarterHours;        // local variable
  int minutesWorked              // NO!
}
```

  - Compiler error results
    - "Redefinition error…"
- Value arguments ARE like "local variables"
  - But function gets them "automatically"

CALL-BY-REFERENCE PARAMETERS

- Used to provide access to caller’s actual argument
- Caller’s data can be modified by called function!
- Typically used for input function
  - To retrieve data for caller
  - Data is then "given" to caller
- Specified by ampersand, &, after type in formal parameter list

CALL-BY-REFERENCE EXAMPLE: **DISPLAY 4.1** CALL-BY-REFERENCE PARAMETERS



Display 4.2 **Call-by-Reference Parameters**

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Display 4.2 **Call-by-Reference Parameters**

SAMPLE DIALOGUE

```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```

CALL-BY-REFERENCE DETAILS

- What’s really passed in?
- A “reference” back to caller’s actual argument!
  - Refers to memory location of actual argument
  - Called “address”, which is a unique number referring to distinct place in memory

CONSTANT REFERENCE PARAMETERS

- Reference arguments inherently "dangerous"
  - Caller’s data can be changed
  - Often this is desired, sometimes not
- To "protect" data, & still pass by reference:
  - Use const keyword
    - void sendConstRef(const int &par1, const int &par2);
    - Makes arguments "read-only" by function
    - No changes allowed inside function body

Why “const”?

PARAMETERS AND ARGUMENTS

- Confusing terms, often used interchangeably
- True meanings:
  - Formal parameters
    - In function declaration and function definition
  - Arguments
    - Used to "fill-in" a formal parameter
    - In function call (argument list)
  - Call-by-value & Call-by-reference
    - Simply the "mechanism" used in plug-in process

MIXED PARAMETER LISTS

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  void mixedCall(int & par1, int par2, double & par3);
  - Function call:
    mixedCall(arg1, arg2, arg3);
    - arg1 must be integer type, is passed by reference
    - arg2 must be integer type, is passed by value
    - arg3 must be double type, is passed by reference

CHOOSING FORMAL PARAMETER NAMES

- Same rule as naming any identifier:
  - Meaningful names!

- Functions as "self-contained modules"
  - Designed separately from rest of program
  - Assigned to teams of programmers
  - All must "understand" proper function use
  - OK if formal parameter names are same as argument names

- Choose function names with same rules

OVERLOADING

- Same function name
- Different parameter lists
- Two separate function definitions
- Function "**signature**"
  - **Function name & parameter list**
  - Must be "unique" for each function definition
- Allows same task performed on different data

OVERLOADING EXAMPLE: AVERAGE

- Function computes average of 2 numbers:

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

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
  return ((n1 + n2+ n3) / 2.0);
}
```

- **Same name, two functions**

OVERLOADED AVERAGE() CONT’D

- Which function gets called?
- Depends on function call itself:
  - avg = average(5.2, 6.7);
    - Calls "two-parameter average()"
  - avg = average(6.5, 8.5, 4.2);
    - Calls "three-parameter average()"
- Compiler resolves invocation based on signature of function call
  - "Matches" call with appropriate function
  - Each considered separate function

OVERLOADING PITFALL

- Only overload "same-task" functions
  - A mpg() function should always perform same task, in all overloads
  - Otherwise, unpredictable results
- C++ function call resolution:
  - 1st: looks for exact signature
  - 2nd: looks for "compatible" signature

OVERLOADING RESOLUTION

- 1st: Exact Match
  - Looks for exact signature
    - Where no argument conversion required
- 2nd: Compatible Match
  - Looks for “compatible” signature where **automatic type conversion** is possible:
    - 1st with promotion (e.g., int→double)
      - No loss of data
    - 2nd with demotion (e.g., double→int)
      - Possible loss of data

OVERLOADING RESOLUTION EXAMPLE

- Given following functions:
  - 1. void f(int n, double m);
    2. void f(double n, int m);
    3. void f(int n, int m);
  - These calls:

```
f(98, 99);     → Calls #3
f(5.3, 4); → Calls #2
f(4.3, 5.2);   → Calls ???
```

- Avoid such confusing overloading

AUTOMATIC TYPE CONVERSION AND OVERLOADING

- Numeric formal parameters typically made "**double**" type
- Allows for "any" numeric type
  - Any "subordinate" data automatically promoted
    - int → double
    - float → double
    - char → double*More on this later!
- Avoids overloading for different numeric types

AUTOMATIC TYPE CONVERSION AND OVERLOADING EXAMPLE

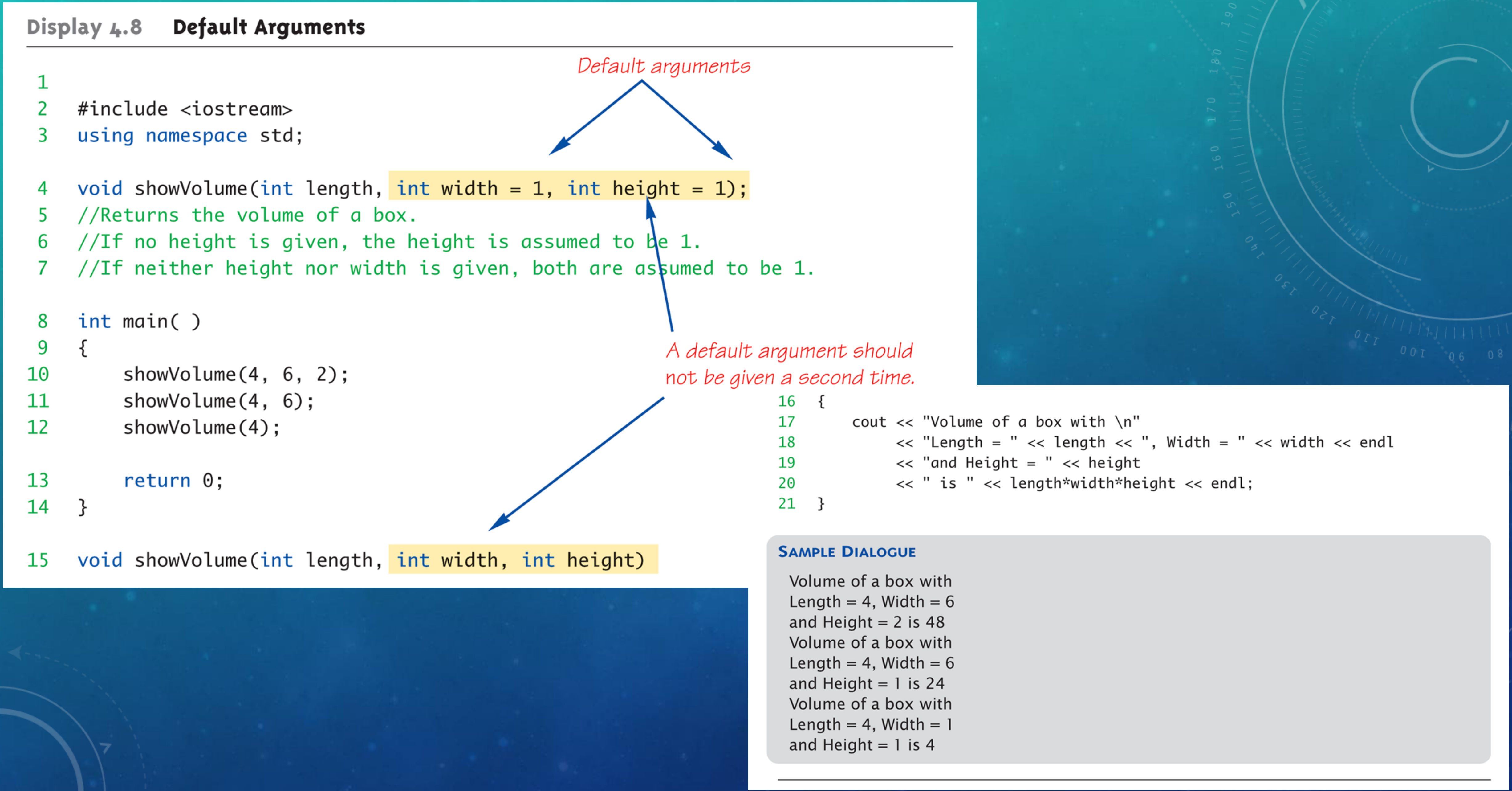
-
  ```
  double mpg(double miles, double gallons)
  {
    return (miles/gallons);
  }
  ```
- Example function calls:
  - mpgComputed = mpg(5, 20);
    - Converts 5 & 20 to doubles, then passes
  - mpgComputed = mpg(5.8, 20.2);
    - No conversion necessary
  - mpgComputed = mpg(5, 2.4);
    - Converts 5 to 5.0, then passes values to function

DEFAULT ARGUMENTS

- Allows omitting some arguments
- Specified in function declaration/prototype

  -
    ```
    void showVolume(  int length,
                      int width = 1,
                      int height = 1);
    ```
    - Last 2 arguments are defaulted
  - Possible calls:
    - showVolume(2, 4, 6); //All arguments supplied
    - showVolume(3, 5); //height defaulted to 1
    - showVolume(7); //width & height defaulted to 1

DEFAULT ARGUMENTS EXAMPLE: **DISPLAY 4.1** DEFAULT ARGUMENTS



Display 4.8 **Default Arguments**

```

#include <iostream>
using namespace std;

void showVolume(int length, int width = 1, int height = 1);
//Returns the volume of a box.
//If no height is given, the height is assumed to be 1.
//If neither height nor width is given, both are assumed to be 1.

int main( )
{
    showVolume(4, 6, 2);
    showVolume(4, 6);
    showVolume(4);

    return 0;
}

void showVolume(int length, int width, int height)
{
    cout << "Volume of a box with \n"
         << "Length = " << length << ", Width = " << width << endl
         << "and Height = " << height
         << " is " << length*width*height << endl;
}
```

Default arguments

A default argument should not be given a second time.

Sample Dialogue

```
Volume of a box with
Length = 4, Width = 6
and Height = 2 is 48
Volume of a box with
Length = 4, Width = 6
and Height = 1 is 24
Volume of a box with
Length = 4, Width = 1
and Height = 1 is 4
```

TESTING AND DEBUGGING FUNCTIONS

- Many methods:
  - **Lots of cout statements**
    - In calls and definitions
    - Used to "trace" execution
  - Compiler Debugger
    - Environment-dependent
  - assert Macro
    - Early termination as needed
  - **Stubs and drivers**
    - Incremental development

THE ASSERT MACRO

- Assertion: a true or false statement
- Used to document and check correctness
  - Preconditions & Postconditions
    - Typical assert use: confirm their validity
  - Syntax:
    assert(<assert_condition>);
    - No return value
    - Evaluates assert_condition
    - Terminates if false, continues if true
- Predefined in library <cassert>
  - Macros used similarly as functions

AN ASSERT MACRO EXAMPLE

- Given Function Declaration:

```
void computeCoin(      int coinValue,  int& number,   int& amountLeft);
```

- //Precondition: 0 < coinValue < 100, 0 <= amountLeft <100
  //Postcondition: number set to max. number   of coins
- Check precondition:
  - **assert** ((0 < currentCoin) && (currentCoin < 100)
                && (0 <= currentAmountLeft) && (currentAmountLeft < 100));
  - If precondition not satisfied → condition is false → program execution **terminates**!

AN ASSERT MACRO EXAMPLE CONT'D

- Useful in debugging

- Stops execution so problem can be investigated

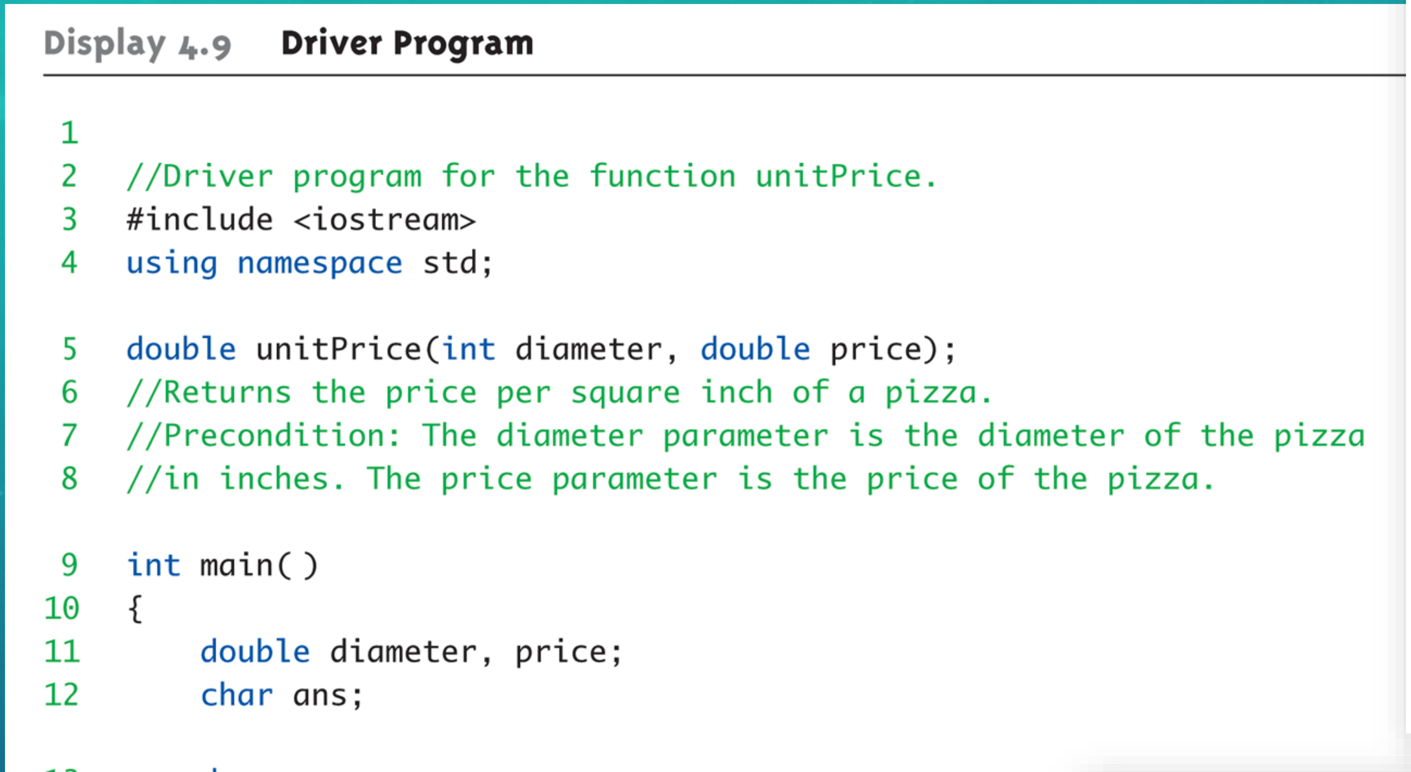
ASSERT ON/OFF

- Preprocessor provides means
- #define **NDEBUG**
  #include <cassert>
- Add "#define" line before #include line
  - Turns OFF all assertions throughout program
- Remove "#define" line (or comment out)
  - Turns assertions back on

STUBS AND DRIVERS

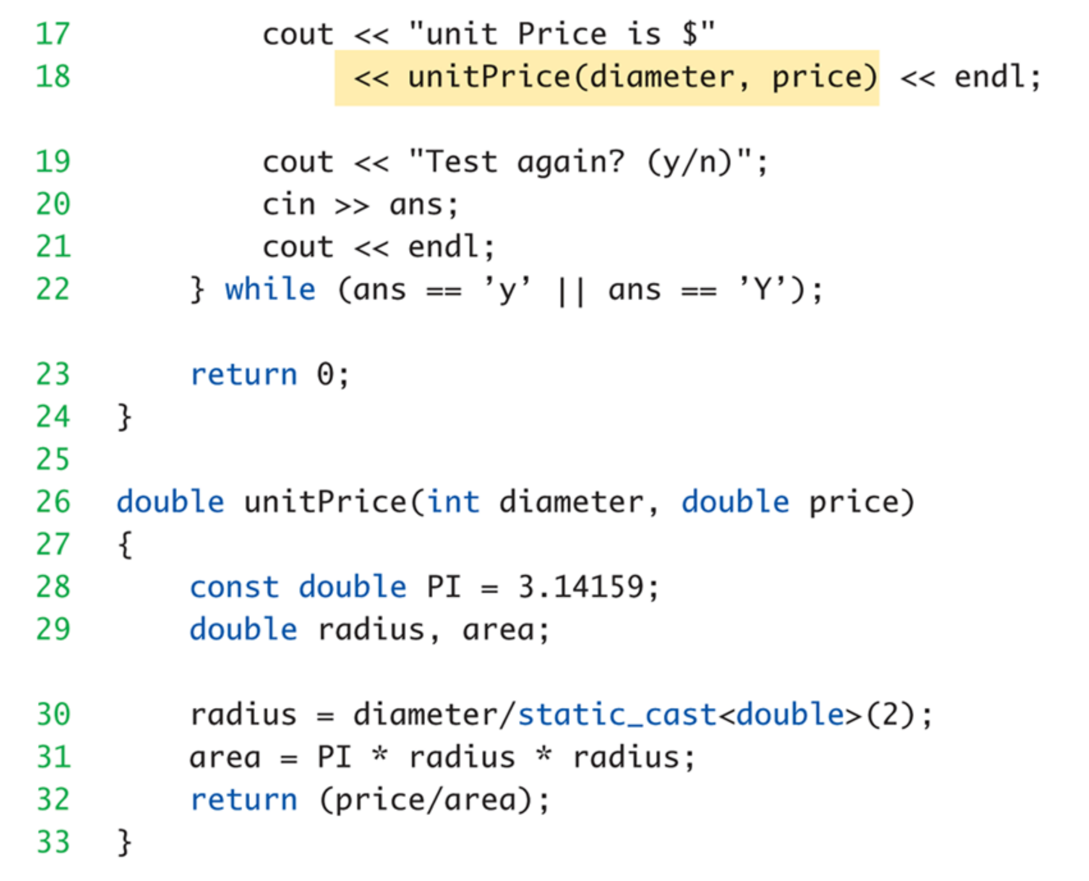
- Separate compilation units
  - Each function designed, coded, tested separately
  - Ensures validity of each unit
  - Divide & Conquer
    - Transforms one big task → smaller, manageable tasks
- But how to test independently?
  - Driver programs

DRIVER PROGRAM EXAMPLE:
DISPLAY 4.9 DRIVER PROGRAM

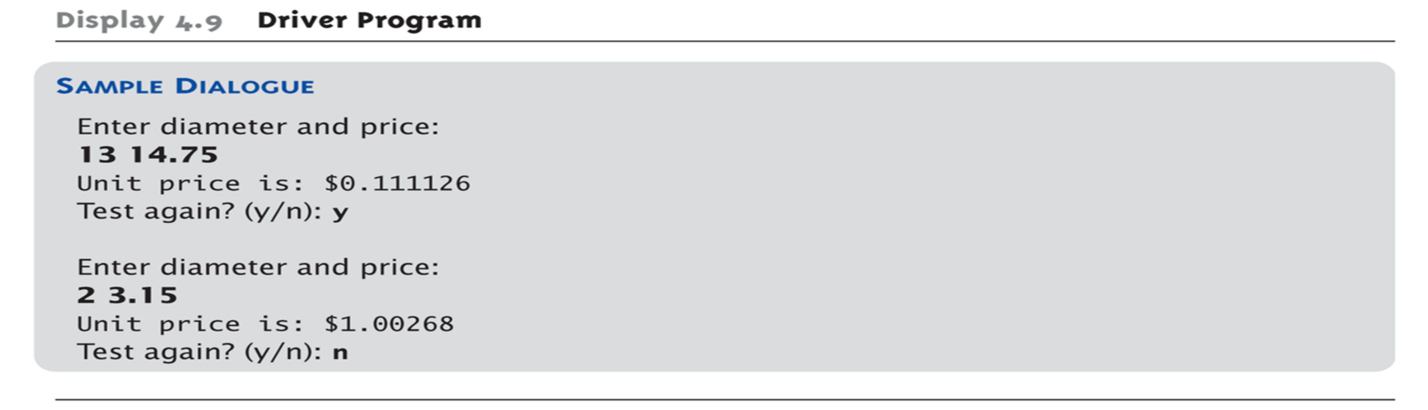


Display 4.9 Driver Program

```
//Driver program for the function unitPrice.
#include <iostream>
using namespace std;

double unitPrice(int diameter, double price);
//Returns the price per square inch of a pizza.
//Precondition: The diameter parameter is the diameter of the pizza
//in inches. The price parameter is the price of the pizza.

int main( )
{
    double diameter, price;
    char ans;

    do
    {
        cout << "Enter diameter and price:\n";
        cin >> diameter >> price;
```



```
        cout << "unit Price is $"
             << unitPrice(diameter, price) << endl;

        cout << "Test again? (y/n)";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}

double unitPrice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

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



Display 4.9 Driver Program

Sample Dialogue

```
Enter diameter and price:
13 14.75
Unit price is: $0.111126
Test again? (y/n): y

Enter diameter and price:
2 3.15
Unit price is: $1.00268
Test again? (y/n): n
```

STUBS

- Develop incrementally
- Write "big-picture" functions first
  - Low-level functions last
  - "Stub-out" functions until implementation
  - Example:

    ```
    double unitPrice(int diameter, double price)
    {
        return (9.99);// not valid, but noticeably a "temporary" value
    }
    ```
  - Calls to function will still "work"

FUNDAMENTAL TESTING RULE

- To write "correct" programs
- Minimize errors, "bugs"
- Ensure validity of data
  - Test every function in a program where every other function has already been fully tested and debugged
  - Avoids "error-cascading" & conflicting results

SUMMARY 1

- Formal parameter is placeholder, filled in with actual argument in function call
- Call-by-value parameters are "local copies" in receiving function body
  - Actual argument cannot be modified
- Call-by-reference passes memory address of actual argument
  - Actual argument can be modified
  - Argument MUST be variable, not constant

SUMMARY 2

- Multiple definitions of same function name possible: called overloading
- Default arguments allow function call to "omit" some or all arguments in list
    - If not provided → default values assigned
- assert macro initiates program termination if assertions fail
- Functions should be tested independently
    - As separate compilation units, with drivers