



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,
page 136

Display 4.1 Formal Parameter Used as a Local Variable (part 1 of 2)

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

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

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

8 int main()
9 {
10     int hours, minutes;
11     double bill;

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

18     bill = fee(hours, minutes);
19
20     cout.setf(ios::fixed);
21     cout.setf(ios::showpoint);
22     cout.precision(2);
23     cout << "For " << hours << " hours and " << minutes
24         << " minutes, your bill is $" << bill << endl;

25     return 0;
26 }
```

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



Call-by-Value Example Cont'd

Display 4.1, page 136

Display 4.1 Formal Parameter Used as a Local Variable (part 2 of 2)

```
26 double fee(int hoursWorked, int minutesWorked)
27 {
28     int quarterHours;

29     minutesWorked = hoursWorked*60 + minutesWorked;
30     quarterHours = minutesWorked/15;
31     return (quarterHours*RATE);
32 }
```

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 4.2,
page 138

Display 4.2 Call-by-Reference Parameters (part 1 of 2)

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

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

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

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

10 int main()
11 {
12     int firstNum, secondNum;

13     getNumbers(firstNum, secondNum);
14     swapValues(firstNum, secondNum);
15     showResults(firstNum, secondNum);
16     return 0;
17 }

18 void getNumbers(int& input1, int& input2)
19 {
20     cout << "Enter two integers: ";
21     cin >> input1
22         >> input2;
23 }

24 void swapValues(int& variable1, int& variable2)
25 {
26     int temp;

27     temp = variable1;
28     variable1 = variable2;
29     variable2 = temp;
30 }

31

32 void showResults(int output1, int output2)
33 {
34     cout << "In reverse order the numbers are: "
35         << output1 << " " << output2 << endl;
36 }
```



Call-By-Reference Example Cont'd

Display 4.2, page 138

Display 4.2 Call-by-Reference Parameters (*part 2 of 2*)

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



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) / 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 4.8,
page 160

Display 4.8 Default Arguments

```
1
2 #include <iostream>
3 using namespace std;
4 void showVolume(int length, int width = 1, int height = 1);
5 //Returns the volume of a box.
6 //If no height is given, the height is assumed to be 1.
7 //If neither height nor width is given, both are assumed to be 1.
8 int main( )
9 {
10     showVolume(4, 6, 2);
11     showVolume(4, 6);
12     showVolume(4);
13
14     return 0;
15 }
16 void showVolume(int length, int width, int height)
17 {
18     cout << "Volume of a box with \n"
19         << "Length = " << length << ", Width = " << width << endl
20         << "and Height = " << height
21         << " is " << length*width*height << endl;
22 }
```

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,
page 163

Display 4.9 Driver Program (part 1 of 2)

```
1
2 //Driver program for the function unitPrice.
3 #include <iostream>
4 using namespace std;

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

9 int main()
10 {
11     double diameter, price;
12     char ans;

13     do
14     {
15         cout << "Enter diameter and price:\n";
16         cin >> diameter >> price;

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

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

23     return 0;
24 }

25
26 double unitPrice(int diameter, double price)
27 {
28     const double PI = 3.14159;
29     double radius, area;

30     radius = diameter/static_cast<double>(2);
31     area = PI * radius * radius;
32     return (price/area);
33 }
```



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