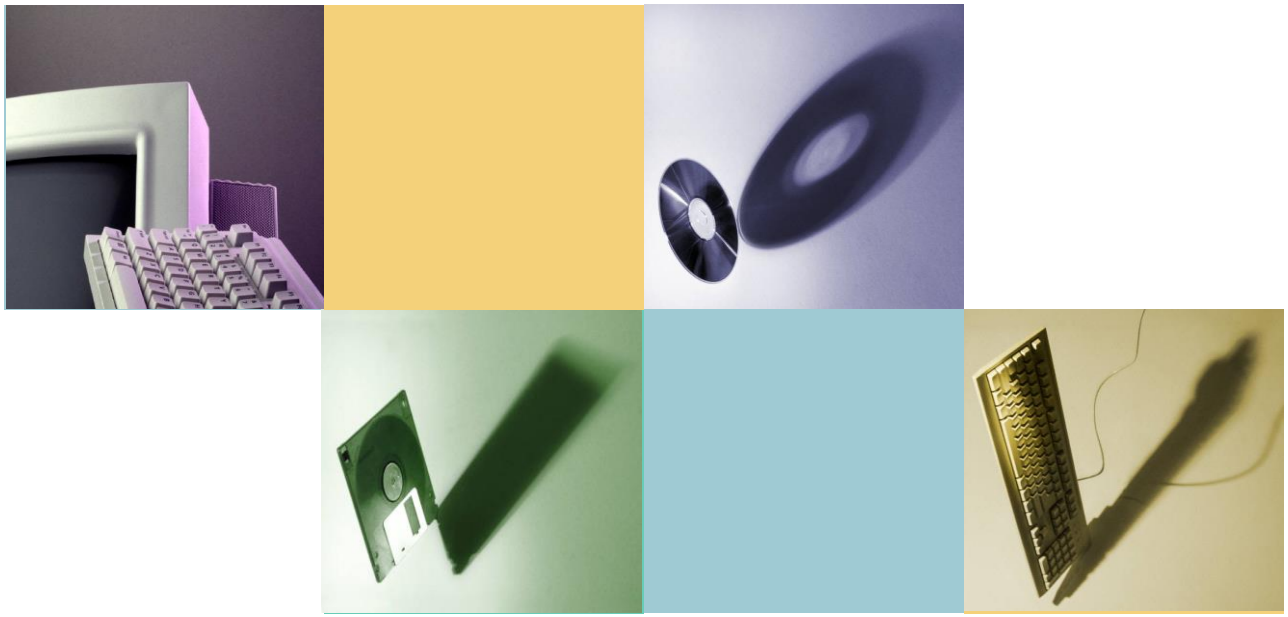
Object-Oriented Programming



Chuan-Kang Ting

Dept of Computer Science and Information Engineering
National Chung Cheng University

Chapter 4

Parameters and Overloading



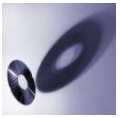
Outlines

- **Parameters**
- **Overloading and Default Arguments**
- **Testing and Debugging Functions**



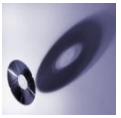
Outlines

- **Parameters**
- Overloading and Default Arguments
- Testing and Debugging Functions


Parameter vs. Argument

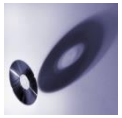
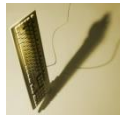
- **Parameter**
  - As a placeholder to stand in for the argument
    - Listed in the function declaration
    - Used in the body of the function definition
- **Argument**
  - An argument is something that is used to fill in a formal parameter


Parameters

- **Call-by-value parameters**
  - Only the **value** of the argument is plugged in
    - The variable's value is not changed by the function call
  - Corresponding arguments can be values or variables
- **Call-by-reference parameters**
  - The argument is a variable; the **variable** itself is plugged in
    - The variable's value can be changed by the function invocation
    - Indicated by appending the ampersand sign "&"
  - Corresponding arguments MUST be variables
  - e.g.

```
void getInput(double& var1, int& var2);
```

Call-by-Value Parameters

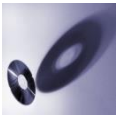
- **A call-by-value parameter is actually a local variable**
  - When the function is invoked, the value of a call-by-value argument is computed; then the corresponding call-by-value parameter (a local variable) is **initialized** to this value

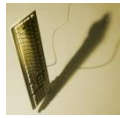
```
12      cout << "Welcome to the law office of\n"
13           << "Dewey, Cheatham, and Howe.\n"
14           << "The law office with a heart.\n"
15           << "Enter the hours and minutes"
16           << " of your consultation:\n";
17      cin >> hours >> minutes;
18      bill = fee(hours, minutes);
19      cout.setf(ios::fixed);
20      cout.setf(ios::showpoint);
21      cout.precision(2);
22      cout << "For " << hours << " hours and " << minutes
23           << " minutes, your bill is $" << bill << endl;
24      return 0;
25  }
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  }
```

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

Work as if **initializing**:
int minutesWorked(minutes)

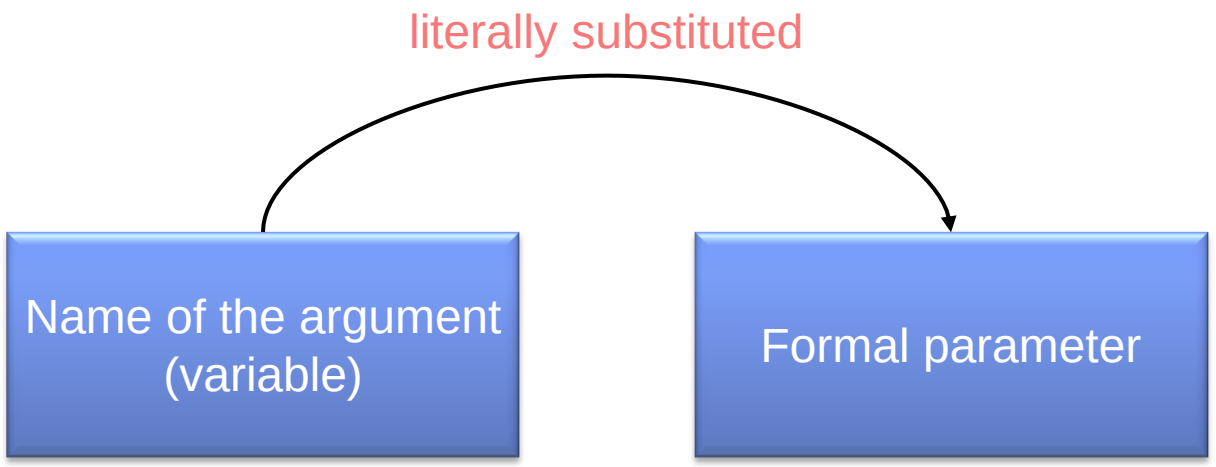
minutesWorked *is a local variable initialized to the value of* **minutes**.

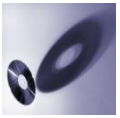




Call-by-Reference Parameters (0)

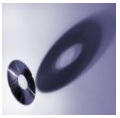
- **Works as if**
  - the name of the variable given as the function argument were **literally substituted** for the call-by-reference formal parameter




Call-by-Reference Parameters (1)

- **Characteristics**
  - Used to provide access to caller's actual argument
  - Caller's data can be modified by called function!
- **Reference?**
  - What's really passed in?
  - A reference back to caller's actual argument!
  - Refers to memory location (address) of actual argument


Call-by-Reference Parameters (2)

- **Scenario**
  - Considering the following function:

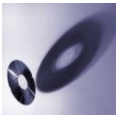
Don't forget "&"!!!

```
void getNumbers(int& input1, int& input2);
```

and a function call

```
getNumbers(firstNum, secondNum);
```

  - When the function called is executed, the function is not given the argument names `firstNum` and `secondNum`. Instead it is given a list of the memory locations associated with each name:
    - 1010 (firstNum)
    - 1012 (secondNum)

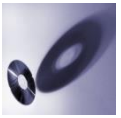




Call-by-Reference Parameters (3)

- **Scenario (cont'd)**

```
firstNum  → 1010 → input1
secondNum → 1012 → input2
```

  - Whatever the function body says to do to a formal parameter is actually done to the variable in the memory location associated with that formal parameter
  - Thus, whatever the function instructs the computer to do to `input1` and `input2` is actually done to the variables `firstNum` and `secondNum`

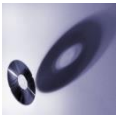

Call-by-Reference Example

Display 4.2 Call-by-Reference Parameters

```
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 }
```

Call-by-reference:

works as if the argument variables were **literally substituted** for parameters

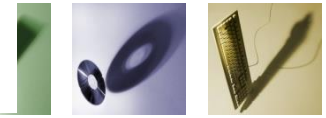

Call-by-Reference Example

```
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  }
```

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

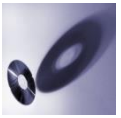
SAMPLE DIALOGUE

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



Unchangeable Parameters

- **Constant Reference Parameters**
  - Placing a `const` before a call-by-reference parameter's type makes that parameter **cannot** be changed
  - No advantages for general use


Outlines

- Parameters
- **Overloading and Default Arguments**
- Testing and Debugging Functions



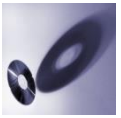
Overloading (0)

- **A way to give two (or more) different function definitions to the same function name**

```
double ave(double a, double b)
{
  return ((a+b)/2.0);
}
```

```
double  ave(double a, double b, double c)
{
  return ((a+b+c)/3.0);
}
```

```
int     ave(int a, int b)
{
  return ((a+b)/2);
}
```


Overloading (1)

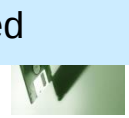
- **Overloading**
  - Have **the same** function name
  - Must have different specifications for their arguments
    - Different numbers of formal parameters
    - OR at least one parameter with different type
  - CANNOT
    - Overload a function name by giving two definitions that differ only in the type of the value returned
    - Overload based only on `const` OR only on call-by-value vs. call-by-reference parameters

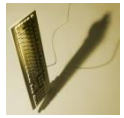
Function's **Signature**

= function's name + sequence of types in parameter list

(excluding `const` and ampersand `&`)

→ Two functions must have different signature, even if overloaded





Overloading (2)

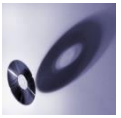
- **Resolution rules**
  1. Exact match
     - Both the number and types of arguments match a definition (without automatic type conversion)
  2. Compatible match
     - There is no exact match but there is a match using *automatic type conversion*

Given:

```
1.  void f(int n, double m);
2.  void f(double n, int m);
```

These calls:

```
f(32, 31.3);     → calls #1
f(30.1, 33);     → calls #2
f(32, 33);       → calls ??
```

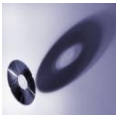

Overloading (3)

- **Rule of thumb**
  - Numeric formal parameters typically made "double" type
    - Allows for "any" numeric type (by automatic type conversion)
      - int → double
      - float → double
  - Avoid overloading for different numeric types



Default Arguments

- **For call-by-value parameters**
  - If the corresponding argument is omitted, then it is replaced by the default argument
  - All the default argument positions must be in the rightmost positions


Default Arguments (cont'd)

Display 4.8 Default Arguments

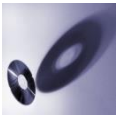
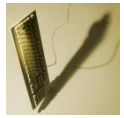
```
1
2  #include <iostream>
3  using namespace std;

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     return 0;
14 }

15 void showVolume(int length, int width, int height)
16 {
17     cout << "Volume of a box with \n"
18          << "Length = " << length << ", Width = " << width << endl
19          << "and Height = " << height
20          << " is " << length*width*height << endl;
21 }
```

Default arguments

A default argument should not be given a second time.

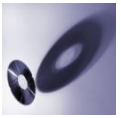
SAMPLE DIALOGUE

```
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
```

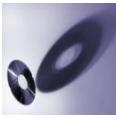
Outlines

- Parameters
- Overloading and Default Arguments
- **Testing and Debugging Functions**


How to Test and Debug?

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

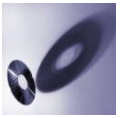

The `assert` Marco

- **The `assert` macro**
  - A compact way for error checks
  - Is used like a `void` function with one call-by-value parameter of type `bool`
  - Syntax:

    `assert(`*boolean expression***`);`**

    - If false, the program ends and an error message is issued
  - e.g.

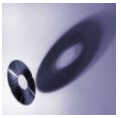
```
#include <cassert>

int main()
{
   ...
   assert((x < 10) && (y > -2));
}
```


The `assert` Macro (cont'd)

- **Use of `assert`**
  - In debugging and checking
  - To investigate the suspicious part of program
  - Turn off `assert`:

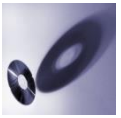
```
#define NDEBUG
#include <cassert>

int main()
{
  ...
  assert((x < 10) && (y > -2));
}
```


Stubs and Drivers

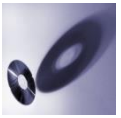
- **Separate compilation units**
  - Each function should be designed, coded, tested separately
  - Ensures validity of each unit
  - Divide & Conquer
    - Transforms one big task → smaller, manageable tasks
- **But how to test independently?**
  - Driver programs
    - Temporary and minimal programs to test a function separately
  - Stubs
    - Used to deal with the interlacing of functions

Test-Driven Development (TDD)


Drivers

- **Need**
  - Obtain reasonable values for the function arguments in as simple as way as possible
    - User inputs → execute the function → show the result
    - Predefined test criteria
- **Need NOT**
  - Fancy input
  - Perform all the calculations the final program will perform


Driver Program (1)

Display 4.9 Driver Program

```
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;
```



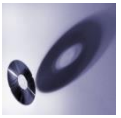
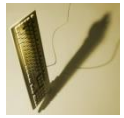
Driver Program (2)

```
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  }
```

SAMPLE DIALOGUE

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

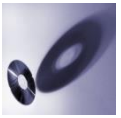
Enter diameter and price:
2 3.15
Unit price is: $1.00268
Test again? (y/n): **n**

Stubs

- **Idea:**
  - Sometimes impossible to test a single function without using others that have not been written or tested yet
  - Use a simplified version for those missing or untested
    → Stub
    - Simplified untested functions that suffice for testing
    - Not necessarily perform correct calculation

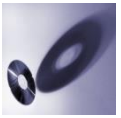
```
double unitPrice(int diameter, double price)
{
  return (9.99);        //not valid, but noticeably a “temporary” value
}
```


Fundamental Testing Rule

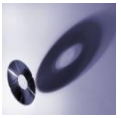
Test every function in a program where *every other* function has already been fully tested and debugged.

- Develop incrementally
  - Write "big-picture" functions first
    - Program outline + stubs
  - Replace stubs on at a time
- To write "correct" programs
  - Minimize errors, "bugs"
  - Ensure validity of data
  - Avoids error-cascading and conflicting results


Summary (1)

- **Call-by-value parameters**
  - Only the **value** of the argument is plugged in
- **Call-by-reference parameters**
  - The argument is a variable; the **variable** itself is plugged in
  - Provide access to caller's actual argument
  - Refers to memory location (address) of actual argument


Summary (2)

- **Overloading**
  - A way to give two (or more) different function definitions to the same function name
    - Different numbers of formal parameters
    - At least one parameter with different type
- **Default arguments**
  - If the corresponding argument is omitted, then it is replaced by the default argument
  - For call-by-value parameters
- **Testing and debugging functions**
  - The `assert` macro
  - Stubs and drivers