Chapter 8
Friends
and
Overloaded
Operators

Overview

• Friend Function (8.1)
• Overloading Operators (8.2)

Program Example:
An Equality Function

• The DayOfYear class from Chapter 6 can be enhanced to include an equality function
• An equality function tests two objects of type DayOfYear to see if their values represent the same date
• Two dates are equal if they represent the same day and month

Declaration of
The equality Function

• We want the equality function to return a value of type bool that is true if the dates are the same
• The equality function requires a parameter for each of the two dates to compare
• The declaration is
  
  bool equal(DayOfYear date1, DayOfYear date2);

  Notice that equal is not a member of the class DayOfYear

Defining Function equal

• The function equal, is not a member function
• It must use public accessor functions to obtain the day and month from a DayOfYear object
• equal can be defined in this way:

  bool equal(DayOfYear date1, DayOfYear date2)
  {
    return ( date1.get_month() == date2.get_month() 
             &&
             date1.get_day() == date2.get_day() );
  }

Friend Function

• Class operations are typically implemented as member functions
• Some operations are better implemented as ordinary (nonmember) functions
Using The Function equal

- The equal function can be used to compare dates in this manner
  
  ```cpp
  if ( equal(today, bach_birthday) )
  cout << "It's Bach's birthday!";
  ```
  
- A complete program using function equal is found in

<table>
<thead>
<tr>
<th>Display 8.1 (1)</th>
<th>Display 8.1 (2)</th>
<th>Display 8.1 (3)</th>
</tr>
</thead>
</table>

Friend Functions

- Friend functions are not members of a class, but can access private member variables of the class
  
  • A friend function is declared using the keyword friend in the class definition
  
  - A friend function is not a member function
  
  - A friend function is an ordinary function
  
  - A friend function has extraordinary access to data members of the class
  
  - As a friend function, the more efficient version of equal is legal

Is equal Efficient?

- Function equal could be made more efficient
  
  • Equal uses member function calls to obtain the private data values
  
  • Direct access of the member variables would be more efficient (faster)

<table>
<thead>
<tr>
<th>Display 8.2 (1)</th>
<th>Display 8.2 (2)</th>
</tr>
</thead>
</table>

Declaring A Friend

- The function equal is declared a friend in the abbreviated class definition here
  
  ```cpp
class DayOfYear
{
  public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    // The rest of the public members
  private:
    // the private members
};
```  

A More Efficient equal

- As defined here, equal is more efficient, but not legal
  
  ```cpp
  bool equal(DayOfYear date1, DayOfYear date2)
  {
    return (date1.month == date2.month
      &&
    date1.day == date2.day);
  }
  ```
  
  - The code is simpler and more efficient
  
  - Direct access of private member variables is not legal

Using A Friend Function

- A friend function is declared as a friend in the class definition
  
  - A friend function is defined as a nonmember function without using the "::" operator
  
  - A friend function is called without using the '.' operator

<table>
<thead>
<tr>
<th>Display 8.2 (1)</th>
<th>Display 8.2 (2)</th>
<th>Display 8.2 (3)</th>
<th>Display 8.2 (4)</th>
</tr>
</thead>
</table>
Friend Declaration Syntax

- The syntax for declaring friend function is:
  ```cpp
  class class_name
  {
    public:
      friend Declaration_for_Friend_Function_1
      friend Declaration_for_Friend_Function_2
      ...
      Member_Function_Declarations
    private:
      Private_Member_Declarations
  };
  ```

Are Friends Needed?

- Friend functions can be written as non-friend functions using the normal accessor and mutator functions that should be part of the class.
- The code of a friend function is simpler and it is more efficient.

Choosing Friends

- How do you know when a function should be a friend or a member function?
  - In general, use a member function if the task performed by the function involves only one object.
  - In general, use a nonmember function if the task performed by the function involves more than one object.
  - Choosing to make the nonmember function a friend is a decision of efficiency and personal taste.

Program Example: The Money Class (version 1)

- Display 8.3 demonstrates a class called Money:
  - U.S. currency is represented.
  - Value is implemented as an integer representing the value as if converted to pennies.
    - An integer allows exact representation of the value.
    - Type long is used to allow larger values.
    - Two friend functions, equal and add, are used.

Characters to Integers

- Notice how function input (Display 8.3) processes the dollar values entered:
  - First read the character that is a $ or a –.
  - If it is the –, set the value of negative to true and read the $ sign which should be next.
  - Next read the dollar amount as a long.
  - Next read the decimal point and cents as three characters.
  - `digit_to_int` is then used to convert the cents characters to integers.

```
int digit_to_int(char c)
{
  return (int(c) - int('0'));
}
```

- A digit, such as '3' is parameter c.
- This is the character '3' not the number 3.
- The type cast int(c) returns the number that implements the character stored in c.
- The type cast int('3') returns the number that implements the character '0'.
**int( c) – int ('0')?**

- The numbers implementing the digits are in order
  - int('0') + 1 is equivalent to int('1')
  - int('1') + 1 is equivalent to int('2')
- If c is '0'
  - int( c) - int('0') returns integer 0
- If c is '1'
  - int( c) – int ('0') returns integer 1

**Class Parameters**

- It can be much more efficient to use call-by-reference parameters when the parameter is of a class type
- When using a call-by-reference parameter
  - If the function does not change the value of the parameter, mark the parameter so the compiler knows it should not be changed

**Leading Zeros**

- Some compilers interpret a number with a leading zero as a base 8 number
  - Base 8 uses digits 0 – 7
- Using 09 to represent 9 cents could cause an error
  - the digit 9 is not allowed in a base 8 number
- The ANSI C++ standard is that input should be interpreted as base 10 regardless of a leading zero

**const Parameter Modifier**

- To mark a call-by-reference parameter so it cannot be changed:
  - Use the modifier const before the parameter type
  - The parameter becomes a constant parameter
  - const used in the function declaration and definition

**Parameter Passing Efficiency**

- A call-by-value parameter less efficient than a call-by-reference parameter
  - The parameter is a local variable initialized to the value of the argument
  - This results in two copies of the argument
- A call-by-reference parameter is more efficient
  - The parameter is a placeholder replaced by the argument
  - There is only one copy of the argument

**const Parameter Example**

- Example (from the Money class of Display 8.3):
  - A function declaration with constant parameters
    - friend Money add(const Money& amount1, const Money& amount2);
  - A function definition with constant parameters
    - Money add(const Money& amount1, const Money& amount2)
      {
        ...
      }
**const Considerations**

- When a function has a constant parameter, the compiler will make certain the parameter cannot be changed by the function
  - What if the parameter calls a member function?
    ```cpp
    Money add(const Money& amount1, const Money& amount2)
    {   ...
        amount1.input( cin );
    }
    ``
  - The call to input will change the value of amount1!

**Function Declarations With const**

- To declare a function that will not change the value of any member variables:
  - Use const after the parameter list and just before the semicolon
    ```cpp
    class Money
    {
        public:
            ...
            void output (ostream& outs) const ;
    ...
    ```

**const And Accessor Functions**

- Will the compiler accept an accessor function call from the constant parameter?
  ```cpp
  Money add(const Money& amount1, const Money& amount2)
  {   ...
        amount1.output(cout);
    }
  ```
  - The compiler will not accept this code
  - There is no guarantee that output will not change the value of the parameter

**Function Definitions With const**

- To define a function that will not change the value of any member variables:
  - Use const in the same location as the function declaration
    ```cpp
    void Money::output(ostream& outs) const
    {
    // output statements
    }
    ```

**const Modifies Functions**

- If a constant parameter makes a member function call...
  - The member function called must be marked so the compiler knows it will not change the parameter
    - const is used to mark functions that will not change the value of an object
    - const is used in the function declaration and the function definition

**const Problem Solved**

- Now that output is declared and defined using the const modifier, the compiler will accept this code
  ```cpp
  Money add(const Money& amount1, const Money& amount2)
  {   ...
        amount1.output(cout);
    }
  ```
const Wrapup

- Using const to modify parameters of class types improves program efficiency
  - const is typed in front of the parameter’s type
- Member functions called by constant parameters must also use const to let the compiler know they do not change the value of the parameter
  - const is typed following the parameter list in the declaration and definition

Display 8.4

Overloading Operators

- In the Money class, function add was used to add two objects of type Money
- In this section we see how to use the ’+’ operator to make this code legal:
  
  ```cpp
  Money total, cost, tax;
  ...
  total = cost + tax;
  // instead of total = add(cost, tax);
  ```

Use const Consistently

- Once a parameter is modified by using const to make it a constant parameter
  - Any member functions that are called by the parameter must also be modified using const to tell the compiler they will not change the parameter
  - It is a good idea to modify, with const, every member function that does not change a member variable

Operators As Functions

- An operator is a function used differently than an ordinary function
  - An ordinary function call enclosed its arguments in parenthesis
    ```cpp
    add(cost, tax)
    ```
  - With a binary operator, the arguments are on either side of the operator
    ```cpp
    cost + tax
    ```

Section 8.1 Conclusion

- Can you
  - Describe the promise that you make to the compiler when you modify a parameter with const?
  - Explain why this declaration is probably not correct?
    ```cpp
    Class Money
    {
      public:
        void input(istream& ins) const;
        ...
    };
    ```

Operator Overloading

- Operators can be overloaded
  - The definition of operator + for the Money class is nearly the same as member function add
  - To overload the + operator for the Money class
    - Use the name + in place of the name add
    - Use keyword operator in front of the +
    - Example:
      ```cpp
      friend Money operator + (const Money& amount1...
Operator Overloading Rules

- At least one argument of an overloaded operator must be of a class type
- An overloaded operator can be a friend of a class
- New operators cannot be created
- The number of arguments for an operator cannot be changed
- The precedence of an operator cannot be changed
- ., ::, *, and ? cannot be overloaded

Program Example: Overloading Operators

- The Money class with overloaded operators + and == is demonstrated in

Display 8.5 (1)  
Display 8.5 (2)

Automatic Type Conversion

- With the right constructors, the system can do type conversions for your classes
  - This code (from Display 8.5) actually works
    
    ```cpp
    Money base_amount(100, 60), full_amount;  
    full_amount = base_amount + 25;  
    ```  
  - The integer 25 is converted to type Money so it can be added to base_amount!  
  - How does that happen?

Type Conversion Event 1

- When the compiler sees base_amount + 25, it first looks for an overloaded + operator to perform
  
  ```cpp
  Money_object + integer  
  ```
  
  - If it exists, it might look like this
    ```cpp
    friend Money operator +(const Money& amount1,  
    const int& amount2);  
    ```

Type Conversion Event 2

- When the appropriate version of + is not found, the compiler looks for a constructor that takes a single integer
  
  - The Money constructor that takes a single parameter of type long will work
  - The constructor Money(long dollars) converts 25 to a Money object so the two values can be added!

Type Conversion Again

- Although the compiler was able to find a way to add
  
  ```cpp
  base_amount + 25  
  ```
  
  this addition will cause an error
  
  ```cpp
  base_amount + 25.67  
  ```
  
  - There is no constructor in the Money class that takes a single argument of type double
A Constructor For double

• To permit base_amount + 25.67, the following constructor should be declared and defined

class Money
{
    public:
        Money(double amount);
        // Initialize object so its value is $amount
        ...
}

Overloading unary operators

• Unary operators take a single argument
• The unary – operator is used to negate a value
  \[ x = -y \]
• + and - - are also unary operators
• Unary operators can be overloaded
  • The Money class of Display 8.6 can include
    • A binary – operator
    • A unary – operator

Overloading –

• Overloading the – operator with two parameters allows us to subtract Money objects as in

\[ \text{Money amount1, amount2, amount3;} \]
\[ \text{...}\]
\[ \text{amount3 = amount1 – amount2;} \]

• Overloading the – operator with one parameter allows us to negate a money value like this

\[ \text{amount3 = -amount1;} \]

What Does << Return?

• Because << is a binary operator

\[ \text{cout << "I have " << amount << " in my purse.";} \]

seems as if it could be grouped as

\[ (\text{cout << "I have"}) << \text{amount} \]

• To provide cout as an argument for << amount,

\[ \text{(cout << "I have") must return cout} \]

Display 8.6

Overloading << and >>

• The insertion operator << is a binary operator
• The first operand is the output stream
• The second operand is the value following <<

\[ \text{cout << "Hello out there,\n";} \]

Operand 1

Operand 2

Operator

Display 8.7

Replacing Function output

• Overloading the << operator allows us to use << instead of Money's output function
• Given the declaration: Money amount(100);

\[ \text{amount.output( cout);} \]

• can become

\[ \text{cout << amount;} \]
Overloaded << Declaration

- Based on the previous example, << should return its first argument, the output stream
  - This leads to a declaration of the overloaded << operator for the Money class:

```cpp
class Money
{
public:
    ...
    friend ostream& operator << (ostream& outs, const Money& amount);
    ...
```

Overloading >>

- Overloading the >> operator for input is very similar to overloading the << for output
  - >> could be defined this way for the Money class

```cpp
istream& operator >> (istream& ins, Money& amount);
{
    // This part is the same as the body of Money::input in Display 8.3 (except that all_cents is replaced with amount.all_cents)
    return ins;
}
```

Overloaded << Definition

- The following defines the << operator
  - ostream& operator <<(ostream& outs, const Money& amount)

```cpp
    {
        // Same as the body of Money::output in Display 8.3 (except all_cents is replaced with amount.all_cents)
        return outs;
    }
```

Section 8.2 Conclusion

- Can you
  - Describe the purpose of making a function a friend?
  - Describe the use of constant parameters?
  - Identify the return type of the overloaded operators << and >>?

Return ostream& ?

- The & means a reference is returned
  - So far all our functions have returned values
  - The value of a stream object is not so simple to return
    - The value of a stream might be an entire file, the keyboard, or the screen!
  - We want to return the stream itself, not the value of the stream
  - The & means that we want to return the stream, not its value

Chapter 8 -- End
Equality function (part 1 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

// int user month = month;
// int user day = day;

DateMonth( int month, int day )
{
    if( month > 12 || month <= 0 )
        throw new IllegalArgumentException( "Invalid month!");
    if( day > 31 || day <= 0 )
        throw new IllegalArgumentException( "Invalid day!");

    this.month = month;
    this.day = day;
}
```

Sample Dialogue:

```
Enter today's date:
User: 8, 31
Date 8, 31
Enter the month as a number: 3
Enter the day of the month: 31
```

Equality function (part 2 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

DateMonth( String s )
{
    int month, day;
    String[] parts = s.split( " " );
    month = Integer.parseInt( parts[0] );
    day = Integer.parseInt( parts[1] );
    DateMonth( month, day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```

Equality function as a friend (part 3 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

DateMonth( String s )
{
    int month, day;
    String[] parts = s.split( " " );
    month = Integer.parseInt( parts[0] );
    day = Integer.parseInt( parts[1] );
    DateMonth( month, day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```

Comparison of the class DateMonth:

```
public static boolean equal( DateMonth d1, DateMonth d2 )
{
    return ( d1.month == d2.month &&
             d1.day == d2.day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```

Equality function (part 1 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

DateMonth( int month, int day )
{
    if( month > 12 || month <= 0 )
        throw new IllegalArgumentException( "Invalid month!");
    if( day > 31 || day <= 0 )
        throw new IllegalArgumentException( "Invalid day!");

    this.month = month;
    this.day = day;
}
```

Sample Dialogue:

```
Enter today's date:
User: 8, 31
Date 8, 31
Enter the month as a number: 3
Enter the day of the month: 31
```

Equality function as a friend (part 2 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

DateMonth( String s )
{
    int month, day;
    String[] parts = s.split( " " );
    month = Integer.parseInt( parts[0] );
    day = Integer.parseInt( parts[1] );
    DateMonth( month, day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```

Comparison of the class DateMonth:

```
public static boolean equal( DateMonth d1, DateMonth d2 )
{
    return ( d1.month == d2.month &&
             d1.day == d2.day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```

Equality function (part 1 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

DateMonth( int month, int day )
{
    if( month > 12 || month <= 0 )
        throw new IllegalArgumentException( "Invalid month!");
    if( day > 31 || day <= 0 )
        throw new IllegalArgumentException( "Invalid day!");

    this.month = month;
    this.day = day;
}
```

Sample Dialogue:

```
Enter today's date:
User: 8, 31
Date 8, 31
Enter the month as a number: 3
Enter the day of the month: 31
```

Equality function as a friend (part 2 of 3):

```java
// Class DateMonth:
// Input: user's input
// Output: instance of class DateMonth

DateMonth( String s )
{
    int month, day;
    String[] parts = s.split( " " );
    month = Integer.parseInt( parts[0] );
    day = Integer.parseInt( parts[1] );
    DateMonth( month, day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```

Comparison of the class DateMonth:

```
public static boolean equal( DateMonth d1, DateMonth d2 )
{
    return ( d1.month == d2.month &&
             d1.day == d2.day );
}
```

Sample Dialogue:

```
Enter today's date:
User: Today's date: 8, 31
Date 8, 31
```
Display 8.8 (4/4)

Overview: -- end -- part 6

Other output and exercises:

- Output script: Display 8.8.mpl, output exam templates
- Test script: Display 8.8.mpl
- Output script: Display 8.8.mpl
- Test script: Display 8.8.mpl

- Default script: Display 8.8.mpl
- Test script: Display 8.8.mpl

If output of script is 0:
- Start output "0" or "0" in script or "0"
- Stop output "0"
- Stop output "0"

PREVIOUS:

- Definitions of the mock function and other output options go here.
- See Display 8.8, 8.8.5, and 8.8 for applications.

Screen Output:

- 13/13 Initial From F12 Left/Right...