Chapter 11

Friends, Overloaded Operators, and Arrays in Classes

PEARSON

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overview

11.1 Friend Functions

11.2 Overloading Operators

11.3 Arrays and Classes

11.4 Classes and Dynamic Arrays

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

11.1

Friend Functions

PEARSON

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Friend Function

- Class operations are typically implemented as member functions
- Some operations are better implemented as ordinary (nonmember) functions

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Program Example: An Equality Function

- The DayOfYear class from Chapter 10 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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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( ) );
 }
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Using The Function equal

- The equal function can be used to compare dates in this manner

```
if ( equal( today,  bach_birthday) )
    cout << "It's Bach's birthday!";
```

- A complete program using function equal is found in

Display 11.1 (1)
Display 11.1 (2)
Display 11.1 (3)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.1 (1/3)

DISPLAY 11.1 Equality Function *(part 1 of 3)*

```
1  //Program to demonstrate the function equal. The class DayOfYear
2  //is the same as in Self-Test Exercise 23-24 in Chapter 10.
3  #include <iostream>
4  using namespace std;
5  class DayOfYear
6  {
7  public:
8      DayOfYear(int the_month, int the_day);
9      //Precondition: the_month and the_day form a
10     //possible date. Initializes the date according
11     //to the arguments.
12     DayOfYear( );
13     //Initializes the date to January first.
14     void input( );
15     void output( );
16     int get_month( );
17     //Returns the month, 1 for January, 2 for February, etc.
18     int get_day( );
19     //Returns the day of the month.
20 private:
21     void check_date( );
22     int month;
23     int day;
24 };
25
26 bool equal(DayOfYear date1, DayOfYear date2);
27 //Precondition: date1 and date2 have values.
28 //Returns true if date1 and date2 represent the same date;
29 //otherwise, returns false.
30
31 int main( )
32 {
33     DayOfYear today, bach_birthday(3, 21);
34
35         cout << "Enter today's date:\n";
36     today.input( );
37     cout << "Today's date is ";
38     today.output( );
39
40     cout << "J. S. Bach's birthday is ";
41     bach_birthday.output( );
42
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.1 (2/3)

DISPLAY 11.1 Equality Function *(part 2 of 3)*

```
43      if ( equal(today, bach_birthday))
44          cout << "Happy Birthday Johann Sebastian!\n";
45      else
46          cout << "Happy Unbirthday Johann Sebastian!\n";
47      return 0;
48  }
49
50  bool equal(DayOfYear date1, DayOfYear date2)
51  {
52       return ( date1.get_month( ) == date2.get_month( ) &&
53                   date1.get_day( ) == date2.get_day( ) );
54  }
55
56  DayOfYear::DayOfYear(int the_month, int the_day)
57                    : month(the_month), day(the_day)
58  {
59      check_date();
60  }
61
62  int DayOfYear::get_month( )
63  {
64      return month;
65  }
66
67  int DayOfYear::get_day( )
68  {
69      return day;
70  }
71
72  //Uses iostream:
73  void DayOfYear::input( )
74  {
75      cout << "Enter the month as a number: ";
76      cin >> month;
77      cout << "Enter the day of the month: ";
78      cin >> day;
79  }
80
81  //Uses iostream:
82  void DayOfYear::output( )
83  {
84      cout << "month = " << month
85           << ", day = " << day << endl;
86  }
```

Omitted function and constructor definitions are as in Chapter 10, Self-Test Exercises 14 and 24, but those details are not needed for what we are doing here.

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.1 (3/3)

DISPLAY 11.1 Equality Function *(part 3 of 3)*

Sample Dialogue

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

A More Efficient equal

- As defined here, equal is more efficient, but not legal

```
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!

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Declaring A Friend

- The function equal is declared a friend in the abbreviated class definition here

```
class DayOfYear
{
    public:
        friend bool equal(DayOfYear date1,
                          DayOfYear date2);
        // The rest of the public members
    private:
        //  the private members
};
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Display 11.2

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.2

DISPLAY 11.2 Equality Function as a Friend

```
1  //Demonstrates the function equal.
2  //In this version equal is a friend of the class DayOfYear.
3  #include <iostream>
4  using namespace std;
5
6  class DayOfYear
7  {
8  public:
9      friend bool equal(DayOfYear date1, DayOfYear date2);
10     //Precondition: date1 and date2 have values.
11     //Returns true if date1 and date2 represent the same date;
12     //otherwise, returns false.
13     DayOfYear(int the_month, int the_day);
14     //Precondition: the_month and the_day form a
15     //possible date. Initializes the date according
16     //to the arguments.
17     DayOfYear( );
18     //Initializes the date to January first.
19     void input( );
20     void output( );
21     int get_month( );
22     //Returns the month, 1 for January, 2 for February, etc.
23     int get_day( );
24     //Returns the day of the month.
25 private:
26     void check_date( );
27     int month;
28     int day;
29 };
30
31 int main( )
32 {
```

<The main part of the program is the same as in Display 11.1.>

```
33 }
34
35 bool equal(DayOfYear date1, DayOfYear date2)
36 {
37     return ( date1.month == date2.month &&
38                             date1.day == date2.day );
39 }
40
```

Note that the private member variables `month` *and* `day` *can be accessed by name.*

<The rest of this display, including the Sample Dialogue, is the same as in Display 11.1.>

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Friend Declaration Syntax

- The syntax for declaring friend function is

```
class class_name
{
   public:
          friend Declaration_for_Friend_Function_1
           friend Declaration_for_Friend_Function_2
              …
          Member_Function_Declarations
   private:
          Private_Member_Declarations
 };
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Program Example: The Money Class (version 1)

- Display 11.3 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

Display 11.3 (1 – 5)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.3 (1/5)

Money Class—Version 1 (*part 1 of 5*)

```
//Program to demonstrate the class Money.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool equal(Money amount1, Money amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if the amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);
    //Initializes the object so its value represents an amount with the
    //dollars and cents given by the arguments. If the amount is negative,
    //then both dollars and cents must be negative.

    Money(long dollars);
    //Initializes the object so its value represents $dollars.00.

    Money( );
    //Initializes the object so its value represents $0.00.

    double get_value( );
    //Precondition: The calling object has been given a value.
    //Returns the amount of money recorded in the data of the calling object.

    void input(istream& ins);
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file. An amount of money, including a dollar sign, has been
    //entered in the input stream ins. Notation for negative amounts is -$100.00.
    //Postcondition: The value of the calling object has been set to
    //the amount of money read from the input stream ins.
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.3 (2/5)

Money Class—Version 1 (*part 2 of 5*)

```
    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then outs has already been
    //connected to a file.
    //Postcondition: A dollar sign and the amount of money recorded
    //in the calling object have been sent to the output stream outs.
private:
    long all_cents;
};

int digit_to_int(char c);
//Function declaration for function used in the definition of Money::input:
//Precondition: c is one of the digits '0' through '9'.
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.

int main( )
{
    Money your_amount, my_amount(10, 9), our_amount;
    cout << "Enter an amount of money: ";
    your_amount.input(cin);
    cout << "Your amount is ";
    your_amount.output(cout);
    cout << endl;
    cout << "My amount is ";
    my_amount.output(cout);
    cout << endl;

    if (equal(your_amount, my_amount))
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";
    our_amount = add(your_amount, my_amount);
    your_amount.output(cout);
    cout << " + ";
    my_amount.output(cout);
    cout << " equals ";
    our_amount.output(cout);
    cout << endl;
    return 0;
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.3 (3/5)

Money Class—Version 1 (*part 3 of 5*)

```
Money add(Money amount1, Money amount2)
{
    Money temp;

    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

bool equal(Money amount1, Money amount2)
{
    return (amount1.all_cents == amount2.all_cents);
}

Money::Money(long dollars, int cents)
{
    if(dollars*cents < 0) //If one is negative and one is positive
    {
        cout << "Illegal values for dollars and cents.\n";
        exit(1);
    }
    all_cents = dollars*100 + cents;
}

Money::Money(long dollars) : all_cents(dollars*100)
{
    //Body intentionally blank.
}

Money::Money( ) : all_cents(0)
{
    //Body intentionally blank.
}

double Money::get_value( )
{
    return (all_cents * 0.01);
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.3 (4/5)

Money Class—Version 1 (*part 4 of 5*)

```
//Uses iostream, cctype, cstdlib:
void Money::input(istream& ins)
{
    char one_char, decimal_point,
         digit1, digit2; //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if ( one_char != '$' || decimal_point != '.'
         || !isdigit(digit1) || !isdigit(digit2) )
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }
    cents = digit_to_int(digit1)*10 + digit_to_int(digit2);

    all_cents = dollars*100 + cents;
    if (negative)
        all_cents = -all_cents;
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.3 (5/5)

Money Class—Version 1 (*part 5 of 5*)

```
//Uses cstdlib and iostream:
void Money::output(ostream& outs)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;
}

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

Sample Dialogue

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09
One of us is richer.
$123.45 + $10.09 equals $133.54
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Characters to Integers

- Notice how function input (Display 11.3) 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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

digit_to_int (optional)

- digit_to_int is defined as

```
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('0') returns the number that implements the character '0'

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

int(c) – int ('0')?

- The numbers implementing the digits are in 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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

const Parameter Example

- Example (from the Money class of Display 11.3):
  - 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)

```
{
    …
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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?

```
Money add(const Money& amount1,
                  const Money& amount2)
{   …
    amount1.input( cin );
}
```

  - The call to input will change the value of amount1!

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

const
And Accessor Functions

- Will the compiler accept an accessor function call from the constant parameter?

```
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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

```
class Money
 {
      public:
                …
                void output (ostream& outs) const ;
                …
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

```
void Money::output(ostream& outs) const
{
        // output statements
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

const Problem Solved

- Now that output is declared and defined using the const modifier, the compiler will accept this code

- ```
  Money add(const Money& amount1,
                    const Money& amount2)
  {       …
          amount1.output(cout);
  }
  ```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.
  ```

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 11.4

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.4

The Class Money with Constant Parameters

```
//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money add(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool equal(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);
    //Initializes the object so its value represents an amount with the
    //dollars and cents given by the arguments. If the amount is negative,
    //then both dollars and cents must be negative.

    Money(long dollars);
    //Initializes the object so its value represents $dollars.00.

    Money( );
    //Initializes the object so its value represents $0.00.

    double get_value( ) const;
    //Precondition: The calling object has been given a value.
    //Returns the amount of money recorded in the data of the calling object.

    void input(istream& ins);
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file. An amount of money, including a dollar sign, has been
    //entered in the input stream ins. Notation for negative amounts is -$100.00.
    //Postcondition: The value of the calling object has been set to
    //the amount of money read from the input stream ins.

    void output(ostream& outs) const;
    //Precondition: If outs is a file output stream, then outs has already been
    //connected to a file.
    //Postcondition: A dollar sign and the amount of money recorded
    //in the calling object have been sent to the output stream outs.
private:
    long all_cents;
};
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Section 11.1 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?

```
class Money
{  …
  public:
          void input(istream& ins) const;
    …
};
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

11.2

Overloading Operators

PEARSON

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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:

```
Money total, cost, tax;
…
total = cost + tax;
// instead of  total = add(cost, tax);
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Operators As Functions

- An operator is a function used differently than an ordinary function
  - An ordinary function call enclosed its arguments in parenthesis

    add(cost, tax)

  - With a binary operator, the arguments are on either side of the operator

    cost + tax

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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:
    friend Money operator + (const Money& amount1…

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Program Example: Overloading Operators

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

Display 11.5 (1)

Display 11.5 (2)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.5 (1/2)

DISPLAY 11.5 Overloading Operators *(part 1 of 2)*

```
1  //Program to demonstrate the class Money. (This is an improved version of
2  //the class Money that we gave in Display 11.3 and rewrote in Display 11.4.)
3  #include <iostream>
4  #include <cstdlib>
5  #include <cctype>
6  using namespace std;
7
8  //Class for amounts of money in U.S. currency.
9  class Money
10 {
11 public:
12     friend Money operator +(const Money& amount1, const Money& amount2);
13     //Precondition: amount1 and amount2 have been given values.
14     //Returns the sum of the values of amount1 and amount2.

15     friend bool operator ==(const Money& amount1, const Money& amount2);
16     //Precondition: amount1 and amount2 have been given values.
17     //Returns true if amount1 and amount2 have the same value;
18     //otherwise, returns false.

19     Money(long dollars, int cents);

20     Money(long dollars);

21     Money( );

22     double get_value( ) const;

23     void input(istream& ins);

24     void output(ostream& outs) const;
25 private:
26     long all_cents;
27 };
```

Some comments from Display 11.4 have been omitted to save space in this book, but they should be included in a real program.

<Any extra function declarations from Display 11.3 go here.>

```
28 int main( )
29 {
30     Money cost(1, 50), tax(0, 15), total;
31     total = cost + tax;

32     cout << "cost = ";
33     cost.output(cout);
34     cout << endl;
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.5 (2/2)

DISPLAY 11.5 Overloading Operators *(part 2 of 2)*

```
35      cout << "tax = ";
36      tax.output(cout);
37      cout << endl;
38      cout << "total bill = ";
39      total.output(cout);
40      cout << endl;
41      if (cost == tax)
42          cout << "Move to another state.\n";
43      else
44          cout << "Things seem normal.\n";
45      return 0;
46  }
47
48  Money operator +(const Money& amount1, const Money& amount2)
49  {
50      Money temp;
51      temp.all_cents = amount1.all_cents + amount2.all_cents;
52      return temp;
53  }
54
55  bool operator ==(const Money& amount1, const Money& amount2)
56  {
57      return (amount1.all_cents == amount2.all_cents);
58  }
59
```

<The definitions of the member functions are the same as in Display 11.3 except that *const* is added to the function headings in various places so that the function headings match the function declarations in the preceding class definition. No other changes are needed in the member function definitions. The bodies of the member function definitions are identical to those in Display 11.3.>

Output

```
cost = $1.50
tax = $0.15
total bill = $1.65
Things seem normal.
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Automatic Type Conversion

- With the right constructors, the system can do type conversions for your classes
  - This code (from Display 11.5) actually works

```
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?

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Type Conversion Event 1

- When the compiler sees base_amount + 25, it first looks for an overloaded + operator to perform

  Money_object + integer

  - If it exists, it might look like this

    friend Money operator +(const Money& amount1,
    const int& amount2);

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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!

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Type Conversion Again

- Although the compiler was able to find a way to add

  base_amount + 25

  this addition will cause an error

  base_amount + 25.67

  - There is no constructor in the Money class that takes a single argument of type double

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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
                …
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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 11.6 can includes
    - A binary – operator
    - A unary – operator

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading -

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

```
Money  amount1, amount2, amount2;
  …
amount3 = amount1 – amount2;
```

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

```
amount3 =  -amount1;
```

Display 11.6

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.6

DISPLAY 11.6 Overloading a Unary Operator

This is an improved version of the class `Money` *given in Display 11.5.*

```
1  //Class for amounts of money in U.S. currency.
2  class Money
3  {
4  public:
5      friend Money operator +(const Money& amount1, const Money& amount2);

6      friend Money operator -(const Money& amount1, const Money& amount2);
7      //Precondition: amount1 and amount2 have been given values.
8      //Returns amount 1 minus amount2.

9      friend Money operator -(const Money& amount);
10     //Precondition: amount has been given a value.
11     //Returns the negative of the value of amount.

12     friend bool operator ==(const Money& amount1, const Money& amount2);

13     Money(long dollars, int cents);

14     Money(long dollars);

15     Money( );

16      double get_value( ) const;

17     void input(istream& ins);
18     void output(ostream& outs) const;
19 private:
20     long all_cents;
21 };
```

We have omitted the `include` *directives and some of the comments, but you should include them in your programs.*

<Any additional function declarations as well as the main part of the program go here.>

```
22 Money operator -(const Money& amount1, const Money& amount2)
23 {
24     Money temp;
25     temp.all_cents = amount1.all_cents - amount2.all_cents;
26     return temp;
27 }

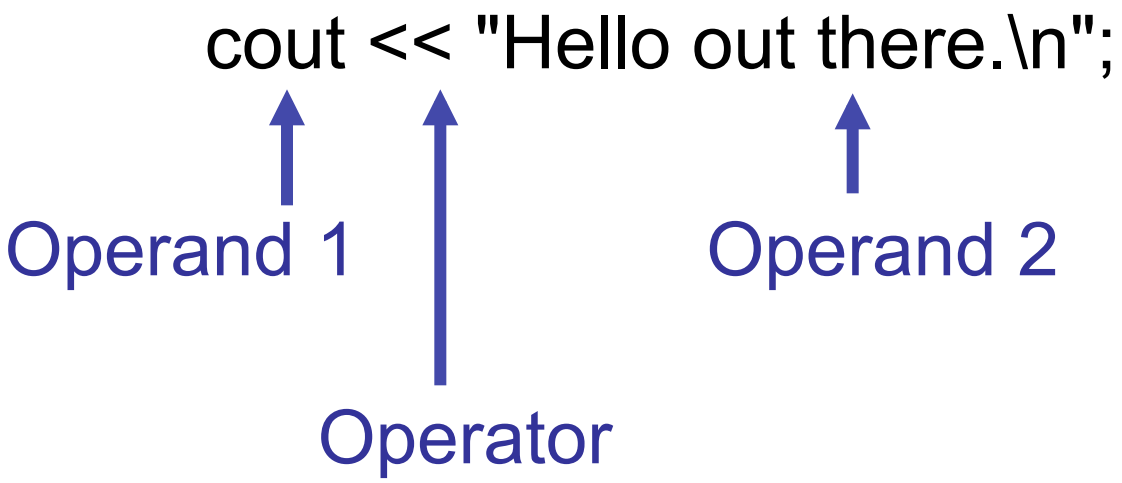
28 Money operator -(const Money& amount)
29 {
30     Money temp;
31     temp.all_cents = -amount.all_cents;
32     return temp;
33 }
```

<The other function definitions are the same as in Display 11.5.>

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading << and >>

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



Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Replacing Function output

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

```
amount.output( cout );
```

can become

```
cout << amount;
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

What Does << Return?

- Because << is a binary operator
  cout << "I have " << amount << " in my purse.";

  seems as if it could be grouped as
  ((cout << "I have") << amount) << "in my purse.";

- To provide cout as an argument for << amount,
  (cout << "I have") must return cout

Display 11.7

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.7

<< as an Operator

```
cout << "I have " << amount << " in my purse.\n";
```

means the same as

```
((cout << "I have ") << amount) << " in my purse.\n";
```

and is evaluated as follows:

First evaluate `(cout << "I have ")`, which returns `cout`:

```
((cout << "I have ") << amount) << " in my purse.\n";
```

and the string "I have" is output.

```
(cout << amount) << " in my purse.\n";
```

Then evaluate `(cout << amount)`, which returns `cout`:

```
(cout << amount) << " in my purse.\n";
```

and the value of amount is output.

```
cout << " in my purse.\n";
```

Then evaluate `cout << " in my purse.\n"`, which returns `cout`:

```
cout << " in my purse.\n";
```

and the string " in my purse.n" is output.

```
cout;
```

Since there are no more << operators, the process ends.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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:

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloaded << Definition

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

```
{
    <Same as the body of Money::output  in
    Display 11.3 (except all_cents is replaced
    with amount.all_cents) >

    return outs;
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading >>

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

```
istream& operator >>(istream& ins,
                     Money& amount);

{
  <This part is the same as the body of
    Money::input in Display 11.3 (except that
    all_cents is replaced with amount.all_cents)>
    return ins;
}
```

Display 11.8 (1-4)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.8 (1/4)

DISPLAY 11.8 Overloading << and >> *(part 1 of 4)*

```
1  //Program to demonstrate the class Money.
2  #include <iostream>
3  #include <fstream>
4  #include <cstdlib>
5  #include <cctype>
6  using namespace std;
7
8  //Class for amounts of money in U.S. currency.
9  class Money
10 {
11 public:
12     friend Money operator +(const Money& amount1, const Money& amount2);
13     friend Money operator -(const Money& amount1, const Money& amount2);
14     friend Money operator -(const Money& amount);
15     friend bool operator ==(const Money& amount1, const Money& amount2);
```

This is an improved version of the class Money *that we gave in Display 11.6.*

Although we have omitted some of the comments from Displays 11.5 and 11.6, you should include them.

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.8(2/4)

DISPLAY 11.8 Overloading << and >> *(part 2 of 4)*

```
16      Money(long dollars, int cents);
17      Money(long dollars);
18      Money( );
19      double get_value( ) const;
20      friend istream& operator >>(istream& ins, Money& amount);
21      //Overloads the >> operator so it can be used to input values of type Money
22      //Notation for inputting negative amounts is as in -$100.00.
23      //Precondition: If ins is a file input stream, then ins has already been
24      //connected to a file.
25      friend ostream& operator <<(ostream& outs, const Money& amount);
26      //Overloads the << operator so it can be used to output values of type Money
27      //Precedes each output value of type Money with a dollar sign.
28      //Precondition: If outs is a file output stream,
29      //then outs has already been connected to a file.
30  private:
31      long all_cents;
32  };
33  int digit_to_int(char c);
34  //Used in the definition of the overloaded input operator >>.
35  //Precondition: c is one of the digits '0' through '9'.
36  //Returns the integer for the digit; for example, digit_to_int('3') returns 3.
37
38  int main( )
39  {
40      Money amount;
41      ifstream in_stream;
42      ofstream out_stream;
43
44      in_stream.open("infile.dat");
45      if (in_stream.fail( ))
46      {
47          cout << "Input file opening failed.\n";
48          exit(1);
49      }
50
51      out_stream.open("outfile.dat");
52      if (out_stream.fail( ))
53      {
54          cout << "Output file opening failed.\n";
55          exit(1);
56      }
57
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.8 (3/4)

DISPLAY 11.8 **Overloading << and >>** *(part 3 of 4)*

```
58      in_stream >> amount;
59      out_stream << amount
60                 << " copied from the file infile.dat.\n";
61      cout << amount
62           << " copied from the file infile.dat.\n";
63
64      in_stream.close( );
65      out_stream.close( );
66
67      return 0;
68  }
69  //Uses iostream, cctype, cstdlib:
70  istream& operator >>(istream& ins, Money& amount)
71  {
72      char one_char, decimal_point,
73           digit1, digit2; //digits for the amount of cents
74      long dollars;
75      int cents;
76      bool negative;//set to true if input is negative.

77      ins >> one_char;
78      if (one_char == '-')
79      {
80          negative = true;
81          ins >> one_char; //read '$'
82      }
83      else
84          negative = false;
85      //if input is legal, then one_char == '$'

86      ins >> dollars >> decimal_point >> digit1 >> digit2;

87      if ( one_char != '$' || decimal_point != '.'
88           || !isdigit(digit1) || !isdigit(digit2) )
89      {
90          cout << "Error illegal form for money input\n";
91          exit(1);
92      }

93      cents = digit_to_int(digit1)*10 + digit_to_int(digit2);

94      amount.all_cents = dollars*100 + cents;
95      if (negative)
96          amount.all_cents = -amount.all_cents;
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.8 (4/4)

DISPLAY 11.8 **Overloading << and >>** *(part 4 of 4)*

```
97      return ins;
98  }
99
100 int digit_to_int(char c)
101 {
102     return ( static_cast<int>(c) - static_cast<int>('0') );
103 }
104 //Uses cstdlib and iostream:
105 ostream& operator <<(ostream& outs, const Money& amount)
106 {
107     long positive_cents, dollars, cents;
108     positive_cents = labs(amount.all_cents);
109     dollars = positive_cents/100;
110     cents = positive_cents%100;
111
112     if (amount.all_cents < 0)
113         outs << "-$" << dollars << '.';
114     else
115         outs << "$" << dollars << '.';
116
117     if (cents < 10)
118         outs << '0';
119     outs << cents;
120
121     return outs;
122 }
123
```

<The definitions of the member functions and other overloaded operators go here.
See Display 11.3, 11.4, 11.5, and 11.6 for the definitions.>

infile.dat
(Not changed by program.)

```
$1.11 $2.22
$3.33
```

outfile.dat
(After program is run.)

```
$1.11 copied from the file infile.dat.
```

Screen Output

```
$1.11 copied from the file infile.dat.
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Section 11.2 Conclusion

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

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

11.3

Arrays and Classes

PEARSON

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Arrays and Classes

- Arrays can use structures or classes as their base types
  - Example:

```
struct WindInfo
{
        double velocity;
        char direction;
 }

WindInfo   data_point[10];
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Accessing Members

- When an array's base type is a structure or a class…
  - Use the dot operator to access the members of anindexed variable
  - Example:

```
for (i = 0; i < 10; i++)
{
        cout << "Enter velocity: ";
        cin >> data_point[i].velocity;
         …
 }
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

An Array of Money

- The Money class of Chapter 11 can be the base type for an array

- When an array of classes is declared
  - The default constructor is called to initialize the indexed variables
- An array of class Money is demonstrated in

Display 11.9 (1-3)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.9 (1/3)

DISPLAY 11.9 Program Using an Array of Money Objects *(part 1 of 3)*

```
1  //This is the definition for the class Money.
2  //Values of this type are amounts of money in U.S. currency.
3  #include <iostream>
4  using namespace std;

5    class Money
6    {
7    public:
8  friend Money operator +(const Money& amount1, const Money& amount2);
9  //Returns the sum of the values of amount1 and amount2.

10 friend Money operator -(const Money& amount1, const Money& amount2);
11 //Returns amount 1 minus amount2.

12 friend Money operator -(const Money& amount);
13 //Returns the negative of the value of amount.

14 friend bool operator ==(const Money& amount1, const Money& amount2);
15 //Returns true if amount1 and amount2 have the same value; false otherwise.

16 friend bool operator < (const Money& amount1, const Money& amount2);
17 //Returns true if amount1 is less than amount2; false otherwise.

18 Money(long dollars, int cents);
19 //Initializes the object so its value represents an amount with
20 //the dollars and cents given by the arguments. If the amount
21 //is negative, then both dollars and cents should be negative.

22 Money(long dollars);
23 //Initializes the object so its value represents $dollars.00.

24 Money( );
25 //Initializes the object so its value represents $0.00.

26 double get_value( ) const;
27 //Returns the amount of money recorded in the data portion of the calling
28 //object.

29 friend istream& operator >>(istream& ins, Money& amount);
30 //Overloads the >> operator so it can be used to input values of type
31 //Money. Notation for inputting negative amounts is as in -$100.00.
32 //Precondition: If ins is a file input stream, then ins has already been
33 //connected to a file.
34
35 friend ostream& operator <<(ostream& outs, const Money& amount);
36 //Overloads the << operator so it can be used to output values of type
37 //Money. Precedes each output value of type Money with a dollar sign.
38 //Precondition: If outs is a file output stream, then outs has already been
39 //connected to a file.
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.9 (2/3)

DISPLAY 11.9 Program Using an Array of Money Objects *(part 2 of 3)*

```
40      private:
41      long all_cents;
42  };
43
```

<The definitions of the member functions and the overloaded operators goes here.>

```
44  //Reads in 5 amounts of money and shows how much each
45  //amount differs from the largest amount.
46  int main( )
47  {
48      Money amount[5], max;
49      int i;
50      cout << "Enter 5 amounts of money:\n";
51      cin >> amount[0];
52      max = amount[0];
53      for (i = 1; i < 5; i++)
54      {
55          cin >> amount[i];
56          if (max < amount[i])
57              max = amount[i];
58          //max is the largest of amount[0],..., amount[i].
59      }
60      Money difference[5];
61      for (i = 0; i < 5; i++)
62          difference[i] = max - amount[i];
63      cout << "The highest amount is " << max << endl;
64      cout << "The amounts and their\n"
65           << "differences from the
66  largest are:\n";
67       for (i = 0; i < 5; i++)
68      {
69          cout << amount[i] << " off by "
70               << difference[i] << endl;
71      }
72      return 0;
73  }
```

Sample Dialogue

```
Enter 5 amounts of money:
$5.00 $10.00 $19.99 $20.00 $12.79
The highest amount is $20.00
The amounts and their
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.9 (3/3)

DISPLAY 11.9 Program Using an Array of Money Objects *(part 3 of 3)*

```
differences from the largest are:
$5.00 off by $15.00
$10.00 off by $10.00
$19.99 off by $0.01
$20.00 off by $0.00
$12.79 off by $7.21
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Arrays as Structure Members

- A structure can contain an array as a member
  - Example:

```
struct Data
{
        double time[10];
        int distance;
}

Data my_best;
```

  - my_best contains an array of type double

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Accessing Array Elements

- To access the array elements within a structure
  - Use the dot operator to identify the array within the structure
  - Use the [ ]'s to identify the indexed variable desired
  - Example: my_best.time[i] references the ith indexed variable of the variable time in the structure my_best

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Arrays as Class Members

- Class TemperatureList includes an array
  - The array, named list, contains temperatures
  - Member variable size is the number of items stored
  -

```
class TemperatureList
  {
    public:
          TemperatureList( );
          //Member functions
     private:
          double   list [MAX_LIST_SIZE];
          int size;
  }
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overview of TemperatureList

- To create an object of type TemperatureList:
  - TemperatureList my_data;
- To add a temperature to the list:
  - My_data.add_temperature(77);
    - A check is made to see if the array is full
- << is overloaded so output of the list is
  - cout << my_data;

Display 11.10 (1-2)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.10 (1/2)

DISPLAY 11.10 Program for a Class with an Array Member *(part 1 of 2)*

```
1  //This is a definition for the class
2  //TemperatureList. Values of this type are lists of Fahrenheit temperatures.
3
4  #include <iostream>
5  #include <cstdlib>
6  using namespace std;
7
8  const int MAX_LIST_SIZE = 50;
9
10 class TemperatureList
11 {
12   public:
13     TemperatureList( );
14     //Initializes the object to an empty list.
15
16     void add_temperature(double temperature);
17     //Precondition: The list is not full.
18     //Postcondition: The temperature has been added to the list.
19
20     bool full( ) const;
21     //Returns true if the list is full; false otherwise.
22
23     friend ostream& operator <<(ostream& outs,
24                                 const TemperatureList& the_object);
25     //Overloads the << operator so it can be used to output values of
26     //type TemperatureList. Temperatures are output one per line.
27     //Precondition: If outs is a file output stream, then outs
28     //has already been connected to a file.
29   private:
30     double list[MAX_LIST_SIZE]; //of temperatures in Fahrenheit
31     int size; //number of array positions filled
32 };
33
34 //This is the implementation for the class TemperatureList.
35
36 TemperatureList::TemperatureList( ) : size(0)
37 {
38    //Body intentionally empty.
39 }
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.10 (2/2)

DISPLAY 11.10 Program for a Class with an Array Member *(part 2 of 2)*

```
40  void TemperatureList::add_temperature(double temperature)
41  {//Uses iostream and cstdlib:
42      if ( full( ) )
43      {
44          cout << "Error: adding to a full list.\n";
45          exit(1);
46      }
47      else
48      {
49          list[size] = temperature;
50          size = size + 1;
51      }
52  }
53  bool TemperatureList::full( ) const
54  {
55       return (size == MAX_LIST_SIZE);
56  }
57  //Uses iostream:
58  ostream& operator <<(ostream& outs, const TemperatureList& the_object)
59  {
60     for (int i = 0; i < the_object.size; i++)
61          outs << the_object.list[i] << " F\n";
62      return outs;
63  }
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Section 11.3 Conclusion

- Can you
  - Declare an array as a member of a class?
  - Declare an array of objects of a class?
  - Write code to call a member function of an element in an array of objects of a class?
  - Write code to access an element of an array of integers that is a member of a class?

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

11.4

Classes and Dynamic Arrays

PEARSON

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Classes and Dynamic Arrays

- A dynamic array can have a class as its base type
- A class can have a member variable that is a dynamic array
  - In this section you will see a class using a dynamic array as a member variable.

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Program Example: A String Variable Class

- We will define the class StringVar
  - StringVar objects will be string variables
  - StringVar objects use dynamic arrays whose size is determined when the program is running
  - The StringVar class is similar to the string class discussed earlier

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The StringVar Constructors

- The default StringVar constructor creates an object with a maximum string length of 100
- Another StringVar constructor takes an argument of type int which determines the maximum string length of the object
- A third StringVar constructor takes a C-string argument and…
  - sets maximum length to the length of the C-string
  - copies the C-string into the object's string value

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The StringVar Interface

- In addition to constructors, the StringVar interface includes:
  - Member functions
    - int length( );
    - void input_line(istream& ins);
    - friend ostream& operator << (ostream& outs,
      const StringVar& the_string);
  - Copy Constructor …discussed later
  - Destructor …discussed later

Display 11.11 (1) **Display 11.11 (2)**

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.11 (1/3)

DISPLAY 11.11 Program Using the StringVar Class *(part 1 of 3)*

```
1  //This is the definition for the class StringVar
2  //whose values are strings. An object is declared as follows.
3  //Note that you use (max_size), not [max_size]
4  //     StringVar the_object(max_size);
5  //where max_size is the longest string length allowed.
6  #include <iostream>
7  using namespace std;
8
9  class StringVar
10 {
11 public:
12     StringVar(int size);
13     //Initializes the object so it can accept string values up to size
14     //in length. Sets the value of the object equal to the empty string.
15
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.11 (2/3)

DISPLAY 11.11 Program Using the `StringVar` Class *(part 2 of 3)*

```
16      StringVar( );
17      //Initializes the object so it can accept string values of length 100
18      //or less. Sets the value of the object equal to the empty string.
19
20      StringVar(const char a[]);
21      //Precondition: The array a contains characters terminated with '\0'.
22      //Initializes the object so its value is the string stored in a and
23      //so that it can later be set to string values up to strlen(a) in length
24
25      StringVar(const StringVar& string_object);
26      //Copy constructor.
27
28      ~StringVar( );
29      //Returns all the dynamic memory used by the object to the freestore.
30
31      int length( ) const;
32      //Returns the length of the current string value.
33      void input_line(istream& ins);
34      //Precondition: If ins is a file input stream, then ins has been
35      //connected to a file.
36      //Action: The next text in the input stream ins, up to '\n', is copied
37      //to the calling object. If there is not sufficient room, then
38      //only as much as will fit is copied.
39      friend ostream& operator <<(ostream& outs, const StringVar& the_string);
40      //Overloads the << operator so it can be used to output values
41      //of type StringVar
42      //Precondition: If outs is a file output stream, then outs
43      //has already been connected to a file.
44
45  private:
46      char *value; //pointer to dynamic array that holds the string value.
47      int max_length; //declared max length of any string value.
48  };
49
50
51  <The definitions of the member functions and overloaded operators go here>
52
53  //Program to demonstrate use of the class StringVar.
54
55  void conversation(int max_name_size);
56  //Carries on a conversation with the user.
57
```

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

A StringVar Sample Program

- Using the StringVar interface of Display 11.11, we can write a program using the StringVar class
  - The program uses function conversation to
    - Create two StringVar objects, your_name and our_name
    - your_name can contain any string max_name_size or shorter in length
    - our_name is initialized to "Borg" and can have any string of 4 or less characters

Display 11.11 (3)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.11 (3/3)

DISPLAY 11.11 Program Using the StringVar Class *(part 3 of 3)*

```
58  int main( )
59  {
60       using namespace std;
61      conversation(30);
62      cout << "End of demonstration.\n";
63      return 0;
64  }
65
66  // This is only a demonstration function:
67  void conversation(int max_name_size)
68  {
69       using namespace std;
70
71      StringVar your_name(max_name_size), our_name("Borg");
72
73      cout << "What is your name?\n";
74      your_name.input_line(cin);
75      cout << "We are " << our_name << endl;
76      cout << "We will meet again " << your_name << endl;
77  }
```

Memory is returned to the freestore when the function call ends.

Determines the size of the dynamic array

Sample Dialogue

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
End of demonstration
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The StringVar Implementation

- StringVar uses a dynamic array to store its string
  - StringVar constructors call new to create the dynamic array for member variable value
  - '\0' is used to terminate the string
  - The size of the array is not determined until the array is declared
    - Constructor arguments determine the size

Display 11.12 (1)

Display 11.12 (2)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.12 (1/2)

DISPLAY 11.12 Implementation of `StringVar` *(part 1 of 2)*

```
1  //This is the implementation of the class StringVar.
2  //The definition for the class StringVar is in Display 11.11.
3  #include <cstdlib>
4  #include <cstddef>
5  #include <cstring>
6
7  //Uses cstddef and cstdlib:
8  StringVar::StringVar(int size) : max_length(size)
9  {
10     value = new char[max_length + 1];//+1 is for '\0'.
11     value[0] = '\0';
12 }
13
14 //Uses cstddef and cstdlib:
15 StringVar::StringVar( ) : max_length(100)
16 {
17     value = new char[max_length + 1];//+1 is for '\0'.
18     value[0] = '\0';
19 }
20
21 //Uses cstring, cstddef, and cstdlib:
22 StringVar::StringVar(const char a[]) : max_length(strlen(a))
23 {
24     value = new char[max_length + 1];//+1 is for '\0'.
25     strcpy(value, a);
26 }
27 //Uses cstring, cstddef, and cstdlib:
28 StringVar::StringVar(const StringVar& string_object)
29                         : max_length(string_object.length( ))
30 {
31     value = new char[max_length + 1];//+1 is for '\0'.
32     strcpy(value, string_object.value);
33 }
34 StringVar::~StringVar( )
35 {
36     delete [] value;
37 }
38
39 //Uses cstring:
40 int StringVar::length( ) const
41 {
42     return strlen(value);
43 }
44
45 //Uses iostream:
```

Copy constructor (discussed later in this chapter)

Destructor

(continued)

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.12 (2/2)

DISPLAY 11.12 Implementation of StringVar *(part 2 of 2)*

```
46   void StringVar::input_line(istream& ins)
47   {
48       ins.getline(value, max_length + 1);
49   }
50
51   //Uses iostream:
52   ostream& operator <<(ostream& outs, const StringVar& the_string)
53   {
54       outs << the_string.value;
55       return outs;
56   }
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Dynamic Variables

- Dynamic variables do not "go away" unless delete is called
  - Even if a local pointer variable goes away at the end of a function, the dynamic variable it pointed to remains unless delete is called
  - A user of the SringVar class could not know that a dynamic array is a member of the class, so could not be expected to call delete when finished with a StringVar object

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Destructors

- A destructor is a member function that is called automatically when an object of the class goes out of scope
  - The destructor contains code to delete all dynamic variables created by the object
  - A class has only one destructor with no arguments
  - The name of the destructor is distinguished from the default constructor by the tilde symbol ~
    - Example: ~StringVar( );

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

~StringVar

- The destructor in the StringVar class must call delete [ ] to return the memory of any dynamic variables to the freestore
  - Example:

```
StringVar::~StringVar( )
{
    delete [ ] value;
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Pointers as Call-by-Value Parameters

- Using pointers as call-by-value parameters yields results you might not expect
  - Remember that parameters are local variables
    - No change to the parameter should cause a change to the argument
  - The value of the parameter is set to the value of the argument (an address is stored in a pointer variable)
    - The argument and the parameter hold the same address
  - If the parameter is used to change the value pointed to, this is the same value pointed to by the argument!

Display 11.13 **Display 11.14**

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.13

DISPLAY 11.13 A Call-by-Value Pointer Parameter

```
1  //Program to demonstrate the way call-by-value parameters
2  //behave with pointer arguments.
3  #include <iostream>
4  using namespace std;
5
6  typedef int* IntPointer;
7
8  void sneaky(IntPointer temp);
9
10 int main( )
11 {
12     IntPointer p;
13
14     p = new int;
15     *p = 77;
16     cout << "Before call to function *p == "
17          << *p << endl;
18     sneaky(p);
19     cout << "After call to function *p == "
20          << *p << endl;
21
22     return 0;
23 }
24
25 void sneaky(IntPointer temp)
26 {
27     *temp = 99;
28     cout << "Inside function call *temp == "
29          << *temp << endl;
30 }
31
```

Sample Dialogue

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

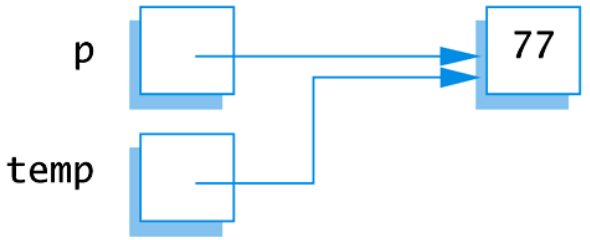
Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Display 11.14

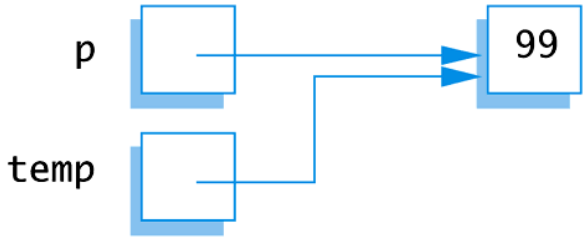
DISPLAY 11.14 The Function Call `sneaky(p);`

1. Before call to `sneaky`:

2. Value of `p` is plugged in for `temp`:



3. Change made to `*temp`:



4. After call to `sneaky`:

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Copy Constructors

- The problem with using call-by-value parameters with pointer variables is solved by the copy constructor.
- A copy constructor is a constructor with one parameter of the same type as the class
  - The parameter is a call-by-reference parameter
  - The parameter is usually a constant parameter
  - The constructor creates a complete, independent copy of its argument

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

StringVar Copy Constructor

- This code for the StringVar copy constructor
  - Creates a new dynamic array for a copy of the argument
    - Making a new copy, protects the original from changes
  - StringVar::StringVar(const StringVar& string_object)
    :
    max_length(string_object.length())
    {
        value = new char[max_length+ 1];
        strcpy(value, string_object.value);
    }

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Calling a Copy Constructor

- A copy constructor can be called as any other constructor when declaring an object
- The copy constructor is called automatically
  - When a class object is defined and initialized by an object of the same class
  - When a function returns a value of the class type
  - When an argument of the class type is plugged in for a call-by-value parameter

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The Need For a Copy Constructor

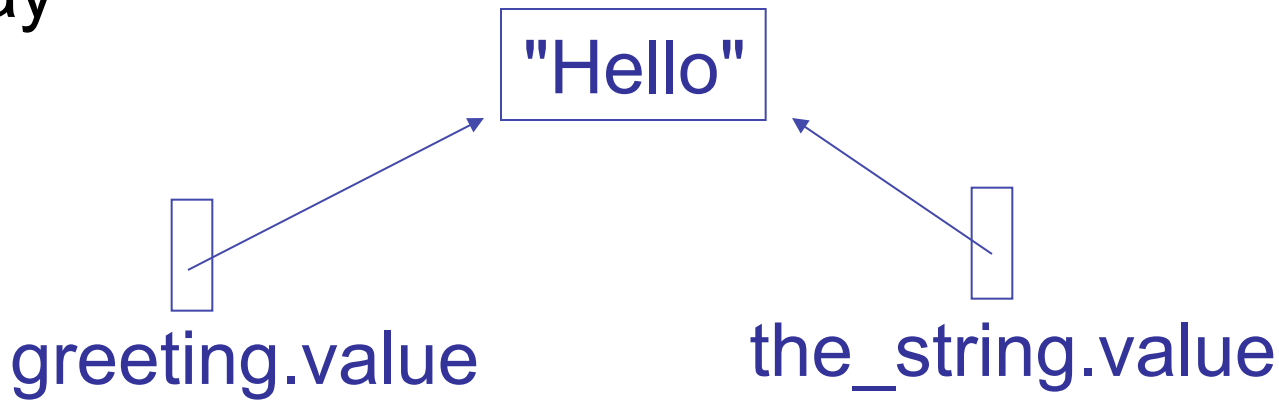
- This code (assuming no copy constructor) illustrates the need for a copy constructor
  - void show_string(StringVar the_string)
    { …}

    StringVar greeting("Hello");
    show_string(greeting);
    cout << greeting << endl;
  - When function show_string is called, greeting is copied into the_string
    - the_string.value is set equal to greeting.value

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The Need For a Copy Constructor (cont.)

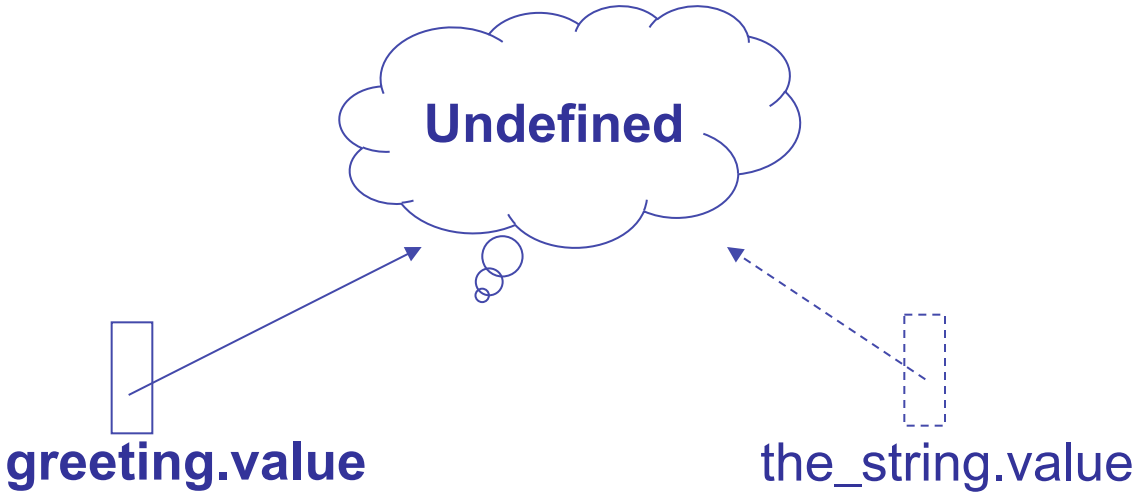
- Since greeting.value and the_string.value are pointers, they now point to the same dynamic array



Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The Need For a Copy Constructor (cont.)

- When show_string ends, the destructor for the_string executes, returning the dynamic array pointed to by the_string.value to the freestore



- greeting.value now points to memory that has been given back to the freestore!

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The Need For a Copy Constructor (cont.)

- Two problems now exist for object greeting
  - Attempting to output greeting.value is likely to produce an error
    - In some instances all could go OK
  - When greeting goes out of scope, its destructor will be called
    - Calling a destructor for the same location twice is likely to produce a system crashing error

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

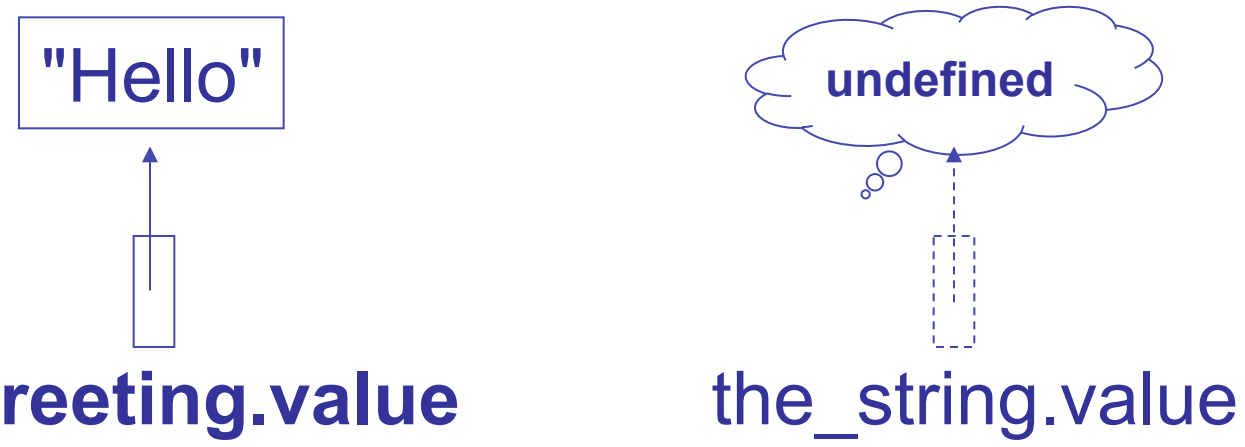
Copy Constructor Demonstration

- Using the same example, but with a copy constructor defined
  - greeting.value and the_string.value point to different locations in memory

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Copy Constructor Demonstration (cont.)

- When the_string goes out of scope, the destructor is called, returning the_string.value to the freestore



  - greeting.value still exists and can be accessed or deleted without problems

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

When To Include a Copy Constructor

- When a class definition involves pointers and dynamically allocated memory using "new", include a copy constructor
- Classes that do not involve pointers and dynamically allocated memory do not need copy constructors

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The Big Three

- The big three include
  - The copy constructor
  - The assignment operator
  - The destructor

- If you need to define one, you need to define all

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

The Assignment Operator

- Given these declarations:

  StringVar string(10), string2(20);

  the statement

  string1 = string2;

  is legal
- But, since StringVar's member value is a pointer, we have string1.value and string2.value pointing to the same memory location

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Overloading =

- The solution is to overload the assignment operator = so it works for StringVar
  - operator = is overloaded as a member function
  - Example: operator = declaration

```
void operator=(const StringVar&  right_side);
```

    - Right_side is the argument from the right side of the = operator

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Definition of =

- The definition of = for StringVar could be:

```
void StringVar::operator=  (const StringVar&   right_side)
{
        int new_length = strlen(right_side.value);
        if (( new_length) > max_length)
            new_length = max_length;

        for(int i = 0; i < new_length; i++)
            value[i] = right_side.value[i];

        value[new_length] = '\0';
}
```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

= Details

- This version of = for StringVar
  - Compares the lengths of the two StringVar's
  - Uses only as many characters as fit in the left hand StringVar object
  - Makes an independent copy of the right hand object in the left hand object

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Problems with =

- The definition of operator = has a problem
  - Usually we want a copy of the right hand argument regardless of its size
  - To do this, we need to delete the dynamic array in the left hand argument and allocate a new array large enough for the right hand side's dynamic array
  - The next slide shows this (buggy) attempt at overloading the assignment operator

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Another Attempt at =

- ```
  void StringVar::operator=  (const StringVar&   right_side)
  {
        delete [ ] value;
        int new_length = strlen(right_side.value);
        max_length = new_length;

        value = new char[max_length + 1];

        for(int i = 0; i < new_length; i++)
            value[i] = right_side.value[i];

        value[new_length] = '\0';
  }
  ```

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.
  ```

A New Problem With =

- The new definition of operator = has a problem
  - What happens if we happen to have the same object on each side of the assignment operator?

    my_string = my_string;
  - This version of operator = first deletes the dynamic array in the left hand argument.
  - Since the objects are the same object, there is no longer an array to copy from the right hand side!

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

A Better = Operator

```
void StringVar::operator = (const StringVar& right_side)
 {
     int new_length = strlen(right_side.value);
     if (new_length > max_length)        //delete value only
       {                                 // if more space
             delete [ ] value;           // is needed
              max_length = new_length;
              value = new char[max_length + 1];
       }

     for (int I = 0; i< new_length; i++)
        value[i] = right_side.value[i];
     value[new_length] = '\0';
}
```

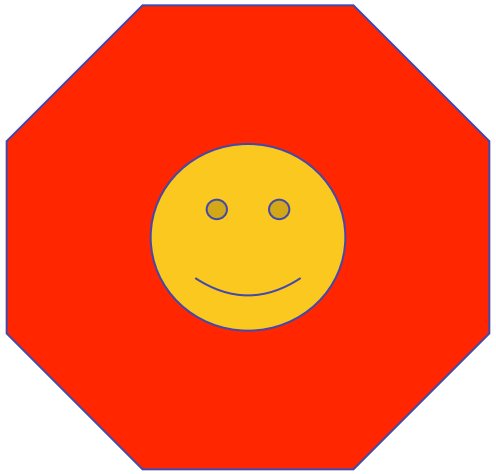
Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Section 11.4 Conclusion

- Can you
  - Explain why an overloaded assignment operator is not needed when the only data consist of built-in types?
  - Explain what a destructor does?
  - Explain when a copy constructor is called?

Copyright © 2014 Pearson Addison-Wesley. All rights reserved.

Chapter 11 -- End



Copyright © 2014 Pearson Addison-Wesley. All rights reserved.