C++ Lecture 14

* Friend Functions and Overloaded Operators
* CIS 251 • Shelby-Hoover Campus

External Functions

* Functions that are not a part of a class normally are not allowed direct access to the private members of that class
* Usually a function that deals with two or more objects given equal standing is not made a member function
* Because such a function must interact with the object through public methods, the syntax can become cumbersome:  
    
  bool equal(DayOfYear date1, DayOfYear date2)  
  {  
   return ((date1.get\_month() == date2.get\_month()) &&  
   (date1.get\_day() == date2.get\_day()));  
  }

Friend Functions

* When you make an external function a **friend function** of a class, it is allowed to access the private members of that class directly without the function call requiring a calling object
* This can be useful when dealing with two or more objects of the class type
* To make an external function a friend function, add a statement to the class definition that includes the keyword friend followed by the prototype of the function
* The prototype may be listed in either the private or the public section of the class definition, but it will be public regardless, so the public section is more appropriate

Friend Function Example

* The class definition for DayOfYear with the friend declaration:  
    
  class DayOfYear  
  {  
  public:  
   friend bool equal(DayOfYear date1, DayOfYear date2);  
   // see Display 11.1 in the textbook for other class members  
  };
* The definition of the method equal with friend status:  
    
  bool equal(DayOfYear date1, DayOfYear date2)  
  {  
   return ((date1.month == date2.month) && (date1.day == date2.day));  
  }

Friends and Members

* Not every function that deals with an object should be made a friend of the class, as having direct access to the private members can be dangerous
* Accessors and mutators are still important components of class definitions, as they provide controlled interaction with the private members of the class
* Whether a function should be a member of a class may depend on how many objects are involved
  + If the task involves only one object (or is primarily linked to a single object), it should probably be a member function
  + If the task involves two or more objects, and none of the objects has priority over the rest, it should probably be a nonmember function (perhaps a friend of the class)

Another Friend Example

* If an object stores a numeric quantity such that the values stored in multiple objects could be added together, the function to perform such an operation could be a friend of the class
* The return type of the function would be another object of that type
* Prototype in the class definition (full class in Display 11.3 in the textbook):  
    
  friend Money add(Money amount1, Money amount2);
* Function definition (note the direct access to the private member variable all\_cents):  
    
  Money add(Money amount1, Money amount2)  
  {  
   Money temp;  
    
   temp.all\_cents = amount1.all\_cents + amount2.all\_cents;  
   return temp;  
  }

Constant Parameters

* Pass-by-reference parameters are more efficient than pass-by-value: the function uses the original argument rather than making a duplicate
* A friend function may not need to have the ability to make changes to the members of the original object (and it might be detrimental to give it such access)
* To benefit from pass-by-reference without running the risk of damage to the original objects, use a **constant reference parameter**:  
    
  Money add(const Money& amount1, const Money& amount2)  
  {  
   // body remains as on previous slide

Constant Calling Objects

* When an object is used to call a class method, it works similarly to pass-by-reference: the method may make changes to the calling object as needed
* Some methods do not need access to make changes to the calling object
* The keyword const can be added to the end of a method prototype and header to indicate that the method should not be allowed to make changes to the calling object:  
    
  void output(ostream& outs) const;  
    
  void Money::output(ostream& outs) const  
  {  
   // body omitted
* The compiler will not allow the method to contain code that assigns a new value to a member variable

Overloading Operators

* The class definitions examined thus far contain named methods to perform common operations (e.g., input, output, comparison, addition)
* The types provided by C++ allow the use of operators to perform these common operations
* In order to use such operators on objects of class types, a class must overload the operators to work with objects of that class
* The overloaded operators are friend functions, and their “names” are the key word operator followed by the characters used for the operator

Binary Arithmetic Operators

* When an object stores numeric values, a binary arithmetic operator allows a programmer to calculate the sum, difference, product, quotient, or remainder of the values in two objects as a new object
* These functions require two constant reference parameters of the class type to represent the values on either side of the operator
* The return type of each of these functions is the class name (the function returns a new object of that type)
* Within the function, a new object stores the arithmetic result
* The function returns this new object

Binary Arithmetic Example

* The friend declaration (prototype) within the Money class (Display 11.5 in the textbook) for an addition operator:  
    
  friend Money operator +(const Money& amount1, const Money& amount2);
* The definition of this friend function outside the class:  
    
  Money operator +(const Money& amount1, const Money& amount2)  
  {  
   Money temp;  
   temp.all\_cents = amount1.all\_cents + amount2.all\_cents;  
   return temp;  
  }
* Using the addition operator within main:  
    
  Money cost(1, 50), tax(0, 15), total;  
  total = cost + tax;

Comparison Operators

* To compare the values stored in two objects, a class can overload the comparison operators
* These functions require two constant reference parameters of the class type for the objects on either side of the operator
* The return type of each of these functions is bool (the function returns true or false)
* The function compares as many (or as few) of the member variables as needed (a programmer decides what counts as “equality” between two objects)
* The function returns the bool result

Comparison Example

* The friend declaration (prototype) within the Money class for an equality comparison operator:  
    
  friend bool operator ==(const Money& amount1, const Money& amount2);
* The definition of this friend function outside the class:  
    
  bool operator ==(const Money& amount1, const Money& amount2)  
  {  
   return (amount1.all\_cents == amount2.all\_cents);  
  }
* Using the equality comparison operator within main:  
    
  Money cost(1, 50), tax(0, 15), total;  
  total = cost + tax;  
  // skipping some code in the middle of Display 11.5 in the textbook  
  if (cost == tax)  
   cout << "Move to another state." << endl;  
  else  
   cout << "Things seem normal." << endl;

Constructors as Operators

* Arithmetic between primitive values of different types is performed automatically (e.g., mixing an int and a double in a single arithmetic statement)
* In order to allow a combination of primitive values with class objects in arithmetic statements, define a constructor that accepts an argument of that primitive type
* The program calls this constructor automatically when attempting to use a value of that primitive type in an arithmetic expression with an object of the class type

Unary Arithmetic Operators

* Remember: a unary operator is one in which there is only one operand (e.g., negation)
* To overload a unary operator, provide a single constant reference parameter of the class type
* The return type of each of these functions is the class name (the function returns a new object of that type)
* The function creates a new object that stores the arithmetic result and returns this new object
* Overloading increment (++) and decrement (--) only provides for the prefix versions of these operators; the postfix version is not covered in the textbook

Unary Arithmetic Example

* The friend declaration (prototype) within the Money class (Display 11.6 in the textbook) for a negation operator:  
    
  friend Money operator -(const Money& amount);
* The definition of this friend function outside the class:  
    
  Money operator -(const Money& amount)  
  {  
   Money temp;  
   temp.all\_cents = -amount.all\_cents;  
   return temp;  
  }
* Using the negation operator within main:  
    
  Money revenue(8729, 53), expenses(5906, 14);  
  expenses = -expenses;

Output Operators

* Output is sent to an output stream (cout or an ofstream variable)
* To output multiple items in a single statement, there must be a way to chain the output together
* To overload the output operator for a class, the first parameter should be a reference parameter to an output stream (ostream&), and the second parameter should be a constant reference parameter to an object
* The method’s return type should also be a reference to an output stream (ostream&) so that the stream will be available to other values in the same statement

Output Example

* The friend declaration (prototype) within the Money class (Display 11.8 in the textbook) for an output operator:  
    
  friend ostream& operator <<(ostream& outs, const Money& amount);
* The definition of this friend function outside the class:  
    
  ostream& operator <<(ostream& outs, const Money& amount)  
  {  
   long positive\_cents, dollars, cents;  
   positive\_cents = labs(amount.all\_cents);  
   dollars = positive\_cents / 100;  
   cents = positive\_cents % 100;  
    
   if (amount.all\_cents < 0)  
   outs << "-";  
   outs << "$" << dollars << '.';  
    
   if (cents < 10)  
   outs << '0';  
   outs << cents;  
    
   return outs;  
  }
* Using the output operator within main:  
    
  Money pocket(5);  
  cout << "I have " << pocket << " in my pocket." << endl;

Input Operators

* Input is received from an input stream (cin or an ifstream variable)
* To receive values for multiple variables in a single statement, there must be a way to chain the input together
* To overload the input operator for a class, the first parameter should be a reference parameter to an input stream (istream&), and the second parameter should be a reference parameter to an object (not constant: the operator should be allowed to make changes to the object)
* The method’s return type should also be a reference to an input stream (istream&) so that the stream will be available to other variables in the same statement

Input Example

* The friend declaration (prototype) within the Money class (Display 11.8 in the textbook) for an input operator:  
    
  friend istream& operator >>(istream& ins, Money& amount);
* The definition of this friend function outside the class:  
    
  istream& operator >>(istream& ins, Money& amount)  
  {  
   char one\_char, decimal\_point, digit1, digit2;  
   long dollars;  
   int cents;  
   bool negative;  
    
   ins >> one\_char; // first character is either – or $  
   if (one\_char == '-')  
   {  
   negative = true; // remember that value is negative  
   ins >> one\_char; // read the $ after the -  
   }  
   else  
   negative = false;  
    
   ins >> dollars >> decimal\_point >> digit1 >> digit2;  
    
   if ( (one\_char != '$') || (decimal\_point != '.') ||  
   (!isdigit(digit1)) || (!isdigit(digit2)) )  
   {  
   cout << "Error! Illegal form for money input." << endl;  
   cout << "Value will not be changed." << endl;  
   }  
   else  
   {  
   cents = digit\_to\_int(digit1)\*10 + digit\_to\_int(digit2);  
    
   amount.all\_cents = dollars\*100 + cents;  
   if (negative)  
   amount.all\_cents = -amount.all\_cents;  
   }  
    
   return ins;  
  }  
    
  int digit\_to\_int(char c)  
  {  
   return ( static\_cast<int>(c) – static\_cast<int>('0') );  
  }
* Using the input operator within main:  
    
  Money pocket;  
  cout << "How much money do you have in your pocket? ";  
  cin >> pocket;