BlogEntry ADT

Objectives

In this laboratory, you

- gain additional experience implementing ADTs in C++.
- use previously defined C++ classes via composition.
- are introduced to the C++ exception mechanism.
- learn how to perform C++ member initialization.
- learn how to use static class methods.

ADT Overview

You probably know what a blog is, but in case your professor doesn't, we are going to briefly describe a blog. A blog is like an online diary where an author periodically posts short entries. The entries are typically arranged in reverse chronological order so that you can read the newest entries first and read your way backward toward the oldest entries.

The BlogEntry ADT encapsulates a single blog entry. It contains author, textual contents, and entry creation and modification dates. Since we conveniently have an implemented Text ADT, we will use it to represent the author and contents of the entry. The process of including one class within another is a form of code reuse called composition. To eliminate internal redundancy, we will develop a third ADT for the date information; we will also use the Date ADT through composition.

A common pattern for accessing data members in objects is to use member functions to set and get data values. These are commonly called getters and setters based on their functionality. We are following the common naming convention of using the prefixes set and get in the setter/getter function names, e.g., getAuthor. Not all data members require both a setter and a getter. For instance, the BlogEntry ADT only provides a getter for the dates. We do this because we want to enforce date integrity for these members by only allowing the ADT to modify them internally.

C++ Concepts Overview

We introduce the following new C++ concepts in this lab.

Member initialization: Including other objects in a class introduces an initialization problem. The default constructors for those objects will run automatically when the containing object's constructor runs. For instance, if a BlogEntry object is created, the default constructors for the Text and Date classes will automatically run. However, some objects may not have a default constructor, or the default constructor may not be the appropriate constructor in that case. Member initialization allows the programmer to specify which constructor to use. For consistency, C++ allows the programmer to use member initialization on the built-in data types, as well.

Exceptions: The standard C++ method for dealing with bad parameters and other difficult—or impossible—situations is to throw an exception. Throwing an exception causes the currently active function to stop execution and return to the calling function. Unless that function or one of its callers takes special steps to handle the exception, the program will be halted. The code that called the function can decide what to do when an exception is thrown. Common responses to an exception include one or more of the following: 1) print out a helpful explanation of what went wrong, 2) try to work around the problem, and 3) halt the program.

Static methods: It would sometimes be useful to call a class member function without requiring an instance of that class to exist. For example, the Date ADT provides a static method, isLeapYear, that determines whether a year is a leap year. We don't want to create a complete date object with a fictional day and month in order to determine

whether the year is a leap year. Static methods can be invoked in the absence of a class instance.

Friends: The protection mechanisms of C++ are too coarse grained. We occasionally need to provide full access to our ADT to a limited set of external functions. By making those functions friends of our ADT, they can access the ADT's internal data structures. For all non-friends, the standard protection mechanisms hold. The most common use of friends is 1) to extend IOStream functionality to new ADTs, and 2) to develop highly coupled classes that may need access to each other's internal data. In this lab book, we use friends only to extend IOStream functionality.

BlogEntry ADT Specification

Data items

The author, the entry contents, and the creation and modification dates.

Structure

The structure is via composition of the Text and Date ADTs.

Operations

`BlogEntry ( )`

Requirements:
None

Results:
Default constructor. Creates an entry with unnamed author and empty contents. Uses the default constructors for all data items.

```
BlogEntry ( const Text& initAuthor, const Text& initContents )
```

Requirements:
None

Results:
Constructor. Creates an entry initialized to the specified author and contents. Uses the default constructors for both Date objects.

```
Text getAuthor ( ) const
```

Requirements:
None

Results:
Getter for author. Returns value of `author` object.

`Text getContents ( ) const`

Requirements:
None

Results:
Getter for contents. Returns value of `contents` object.

`Date getCreateDate ( ) const`

Requirements:
None

Results:
Getter for creation date. Returns value of `created` object.

`Date getModifyDate ( ) const`

Requirements:
None

Results:
Getter for modification date. Returns value of `modified` object.

`void setAuthor ( const Text& newAuthor )`

Requirements:
None

Results:
Setter for author. Sets value of `author` object.

`void setContents ( const Text& newContents )`

Requirements:
None

Results:
Setter for contents. Sets value of `contents` object.

`void showStructure ( ) const`

Requirements:
None

Results:
Outputs the contents of the BlogEntry. Note that this operation is intended for testing/debugging purposes only.

Date ADT Specification

Data items

A day, month, and year. All data values for these will start counting at 1, as is normal for dates.

Structure

All members are integer values.

Operations

`Date ( )`

Requirements:
None

Results:
Default constructor. Creates a date that represents the current date.

`Date ( int day, int month, int year ) throw ( logic_error )`

Requirements:
Parameters must represent a valid date.

Results:
Constructor. Creates a date that represents the specified date.

`int getDay ( ) const`

Requirements:
None

Results:
Getter for day of month. Returns the value of `day`.

`int getMonth ( ) const`

Requirements:
None

Results:
Getter for month. Returns the value of `month`.

`int getYear ( ) const`

Requirements:
None

Results:
Getter for year. Returns the value of `year`.

```
static bool isLeapYear( int year )
```

Requirements:
Year is greater than 1901 A.D.

Results:
Static method. If the specified year is a leap year, returns `true`. Else returns `false`.

```
static int daysInMonth ( int month, int year )
```

Requirements:
Year is greater than 1901 A.D. (The formula we provide uses 1901 as a basis for calculation.)

Results:
Static method. Returns the number of days in the specified month.

```
friend ostream& operator<< ( ostream& out, const Date& date )
```

Requirements:
None

Results:
Outputs the date in the same format as the showStructure function to the appropriate `ostream`. Note: Technically, this is not part of the Date ADT specification, but it is closely tied because it is a friend.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the day, month and year. Note that this operation is intended for testing/debugging purposes only.

Implementation Notes

Member initialization: This is performed in the constructor definition (implementation) after the parameter and exception lists, but before the constructor body's opening brace. The syntax is to begin with a colon (':'), followed by a comma-separated list of initialization elements. Each object initialization element is actually a call to a constructor for a specific member variable. Therefore, the syntax of an initialization element looks like a call to a constructor for that member variable. For example, here is an implementation for the BlogEntry ADT two-parameter constructor.

```
BlogEntry::BlogEntry(const Text& initAuthor,
                     const Text& initContents)
    : author(initAuthor), contents(initContents)

{
    // No additional code necessary
}
```

This code calls the Text constructor for the `author` object with `initAuthor` as a parameter. It then calls the Text constructor for the `contents` object with `initContents` as a parameter. Since no constructor is explicitly listed for the Date objects in the member initialization list, the Date object default constructors will be used. This covers all the BlogEntry member data, so no additional work needs to be done inside the BlogEntry constructor body.

Exceptions: The C++ exception handling mechanism is quite complicated. For the purposes of this book, 1) you will always be throwing a `logic_error` exception, and 2) we only discuss how to generate exceptions—not how to handle them after they are thrown.

Following are the steps for using exceptions in your data structure.

1. Include <stdexcept> in your C++ program file. We already included this in the header files.
2. When the function requirements (also known as prerequisites) are violated, throw an exception. The generic syntax is

   ```
   if ( condition ) throw logic_error("explanation string");
   ```

   The code in the program that deals with the exception can access the string and use it to improve error messages and interaction with the user.

   For instance, in the Date constructor, you can deal with an invalid month by writing

   ```
   if ( month < 1 || month > 12 )
       throw logic_error("month not in valid range");
   ```

3. The last step is to declare that the function throws an exception and which exceptions it may throw. This should be done in both the class declaration file—e.g., *Date.h*—and in the class definition file—e.g., *Date.cpp*. After the function parameter parentheses, write "`throw (logic_error)`". The syntax is the same for both cases.

Static methods: A class member function is made static by placing the keyword `static` in the function declaration. Note: there are many uses of the keyword `static` in C++. Do not use `static` in the function definition because it means something completely different. The syntax for using a static method is to use the class name, followed by the scope resolution operator ("::"), followed by the function name. E.g., "`if( Date::isLeapYear(2000) ) {...}`".

Friends: To make a function a friend of a particular class, use the keyword `friend` in the class declaration, followed by the friend function prototype. Then use the named function as normal; it will have full access to anything in the class, as though it were a class member function. For instance,

```
friend ostream& operator<<(ostream& out, const Date& date);
```

Composition detail: Typically, each class has its own declaration file (the ".h" file). When using composition in a class, you need to `#include` the class declaration files for each of the classes used in composition. This gives the compiler enough information to process the code correctly.

Extending iostream functionality: The iostream class cannot possibly know about all the new classes that you write. By default, it only knows how to deal with the built-in data types. We often would like to use that same functionality with classes. This is accomplished by overloading the << and >> operators. These operators take two parameters: 1) a stream object, and 2) an object of the relevant class. To support compact input and output, (e.g., "`cout << obj1 << obj2;`", instead of two separate statements) the iostream class returns a reference to the stream object in these overloaded operators. To overload the << operator, use code similar to the following example for the Date implementation.

```
ostream& operator<<( ostream& out, const Date& date ) {
    return out << date.month << "/" << date.day << "/" << date.year;
}
```

The key to extending the iostream functionally successfully is 1) using a stream reference for both return type and stream parameter, 2) passing the proper values to the stream, and 3) returning the updated stream.

Step 1: Implement the operations in the two ADTs. Base your implementation on the declarations for the two classes as provided in *Date.h* and *BlogEntry.h*. The standard C++ library functions `time` and `localtime` can be used to access the necessary time and date information. You may need help from your instructor to get this working.

(Note: all header files for this book are included in the student file set. We included a copy of the header file in the first lab in this book for your convenience while getting introduced to implementing ADTs with C++. Throughout the rest of this lab text, please view the header files in the lab files distribution.)

Step 2: Save your implementation of the Date ADT in the file *Date.cpp*. Save your implementation of the BlogEntry ADT in the file *BlogEntry.cpp*. Be sure to document your code.

Compilation Directions

Compile your implementation of the Date ADT in the file *Date.cpp*, your implementation of the BlogEntry ADT in the file *BlogEntry.cpp*, and the test program in the file *test2.cpp*. Compilation details will depend on your programming environment.

Testing

Test your implementation of the Date and BlogEntry ADTs by using the program in the file *test2.cpp*.

Step 1: Download the online test plans for Lab 2.

Step 2: Complete the Test Plan for Test 2-1 by filling in the expected result for each date.

Step 3: Execute Test Plan 2-1. If you discover mistakes in your implementation of the Date ADT, correct them and execute the test plan again.

Step 4: Complete the test plan for Test 2-2 by filling in the expected result for each getter.

Step 5: Execute Test Plan 2-2. If you discover mistakes in your implementation of the Date ADT, correct them and execute the test plan again.

Step 6: Complete the test plan for Test 2-3 by filling in the expected result for each leap year.

Step 7: Execute Test Plan 2-3. If you discover mistakes in your implementation of the Date ADT, correct them and execute the test plan again.

Step 8: Complete the test plan for Test 2-4 by filling in the expected result for each month.

Step 9: Execute Test Plan 2-4. If you discover mistakes in your implementation of the Date ADT, correct them and execute the test plan again.

Step 10: Complete the test plan for Test 2-5 by filling in the expected output for each date.

Step 11: Execute Test Plan 2-5. If you discover mistakes in your implementation of the Date ADT, correct them and execute the test plan again.

Step 12: Complete the test plan for Test 2-6 by filling in the expected result for each blog entry.

Step 13: Execute Test Plan 2-6. If you discover mistakes in your implementation of the BlogEntry ADT, correct them and execute the test plan again.

Step 14: Complete the test plan for Test 2-7 by filling in the expected result for each getter and setter.

Step 15: Execute Test Plan 2-7. If you discover mistakes in your implementation of the BlogEntry ADT, correct them and execute the test plan again.

Programming Exercise 1

The Web has become integral to computing. Many applications now use a web browser as the standard interface. Consequently, it is helpful to be able to generate output in HTML. It is also natural to generate HTML for web-based information delivery systems, such as blogs.

```
void printHTML ( ostream& out ) const
```

Requirements:
None

Results:
Sends output to the stream `out` in the format specified below.

Output Format Specification

```
<html>
<body>
<h1>author here</h1>
<p>
contents here
</p>
<p>
Created: creation date here
</p>
<p>
Last modified: modification date here
</p>
</body>
</html>
```

Note that the output web page is just begging for enhancement (e.g., Title, CSS styling, etc.). A lot of the extra information would belong in an expanded BlogEntry ADT.

Step 1: Add this function to your implementation of the BlogEntry ADT.

Step 2: Activate Test 8 in the test program *test2.cpp* by changing the definition of LAB2_TEST8 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete the test plan for Test 2-8 by filling in the expected result for each test.

Step 4: Execute Test Plan 2-8. If you discover mistakes in your implementation of `printHTML`, correct them and execute the test plan again.

Programming Exercise 2

In order to produce a calendar for a given date, you need to know on which day of the week the date occurs. We are going to enhance the Date ADT to provide access to this information by adding the following member function.

```
int getDayOfWeek( ) const
```

Requirements:
None

Results:
Returns a value between 0 (Sunday) and 6 (Saturday) indicating the day of the week.

The day of the week corresponding to a date can be computed using the following formula:

$$dayOfWeek = (1 + nYears + nLeapYears + nDaysToMonth + day) \% 7$$

where *nYears* is the number of years since 1901, *nLeapYears* is the number of leap years since 1901, and *nDaysToMonth* is the number of days from the start of the year to the start of `month`.

This formula yields a value between 0 (Sunday) and 6 (Saturday) and is accurate for any date from January 1, 1901 through at least December 31, 2099. You can compute the value *nDaysToMonth* dynamically using a loop. Alternatively, you can use an array to store the number of days before each month in a nonleap year and add a correction for leap years when needed.

Step 1: Add this function to your implementation of the Date ADT.

Step 2: Activate Test 9 in the test program *test2.cpp* by changing the definition of LAB2_TEST9 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete the test plan for Test 2-9 by filling in the expected result for each test.

Step 4: Execute Test Plan 2-9. If you discover mistakes in your implementation of `getDayOfWeek`, correct them and execute the test plan again.

Programming Exercise 3

Most applications that use dates will at some point need to sort them in chronological order. It is much easier if the relational operators are overloaded to support date comparison.

```
bool operator== ( const Date& other ) const
```

Requirements:
None

Results:
Returns `true` if this object represents the same date as `other`. Otherwise, returns `false`.

```
bool operator< ( const Date& other ) const
```

Requirements:
None

Results:
Returns `true` if this object represents an earlier date than `other`. Otherwise, returns `false`.

```
bool operator> ( const Date& other ) const
```

Requirements:
None

Results:
Returns `true` if this object represents a later date than `other`. Otherwise, returns `false`.

Step 1: Implement the relational operations described above. Add your implementation of these operations to the file *Date.cpp*. Prototypes for these operations are included in the declaration of the Date class in the file *Date.h*.

Step 2: Activate Test 10 in the test program *test2.cpp* by changing the definition of LAB2_TEST10 from 0 to 1 in *config.h*.

Step 3: Complete the test plan for Test 2-10 by filling in the expected result for each test. Add your own test cases to Test Plan 2-10 so it is complete.

Step 4: Execute Test Plan 2-10. If you discover mistakes in your implementation of the relational operations, correct them and execute the test plan again.

Analysis Exercise 1

A full-page version of this exercise with space for writing in answers is available in the online supplements for Lab 2.

Part A

Design another operation for the BlogEntry ADT and give its specification below. You need not implement the operation, simply describe it.

Function prototype:

Requirements:

Results:

Part B

Describe an application in which you might use your new operation.

Analysis Exercise 2

A full-page version of this exercise with space for writing in answers is available in the online supplements for Lab 2.

The BlogEntry class name suggests that it would be used by composition in a Blog class. Design the Blog ADT, specifying data items, structure, and operations. For each operation, indicate the prototype, any requirements, and the results.

Data Items

Structure

Operations

We provide one example operation to indicate the format and level of detail expected.

```
BlogEntry& operator[]( int entryNumber ) throw ( logic_error )
```

Requirements:
`entryNumber` must represent a valid entry.

Results:
Returns a reference to the specified blog entry.