ECE 71/191T – Data Structures and Algorithms

Dr. Gregory R. Kriehn, Fresno State
C++ Homework Assignment: Chapter 16

Code Due By: Midnight on Thu, Mar 23
Write-up Due By: Office Fri, Mar 24

HOMEWORK #31 – Unordered and Ordered Linked Lists

Implement the `linkedListType` (see page 1091), `unorderedLinkedList`, and `orderedLinkedList` template classes. This will also require use of the `linkedListIterator` class, and implementing `nodeType` as a template structure. The `nodeType` template structure, `linkedListIterator` class, and `linkedListType` class can all be placed in the header file "`linkedList.h`". Use separate files for `unorderedLinkedList` and `orderedLinkedList`.

Finally, extend the class `linkedListType` by adding the following function:

```
rotate();
// Function to remove the first node of a linked list and
// put it at the end of the linked list
```

Write a program to test your function. Use the class `unorderedLinkedList` and `orderedLinkedList` to create linked lists for testing purposes.

NOTE: When trying to access a data member from a template base class (i.e., the `first`, `last`, and `count` protected members from `linkedListType<Type>`), the derived classes `unorderedLinkedList` and `orderedLinkedList` are unable to determine if the data member is really coming from the base class. This is because the data members are known as nondependent names, even though their types are dependent upon the base class. For example:

```
nodeType<Type> *first;   // first declared as a pointer to
                         // nodeType<Type> within
                         // linkedListType<Type>.
```

`first` is declared within the base template class, but when `first` is used elsewhere in a derived class, it is still considered a nondependent name, even though its data type within the base class (`nodeType<Type>`) is a dependent name. Trying to compile your code with a statement like:

```
first = current;
```

in your derived class will result in a very obscure compiler error since, again, `first` is viewed locally by the compiler within the derived class as a nondependent name.

The rule for the compiler is that it does not look in dependent base classes (like `linkedListType<Type>`) when looking up nondependent names like `first`, `last`, and `count` – even though they are inherited from the base class. However, since these data members are still dependent upon `linkedListType<Type>`, we need to be explicit about what they depend upon.

Workarounds include:

1. Change `first` to `linkedListType<Type>::first`. This resolves the scope of `first`. If `first` referred to a virtual member function of `linkedListType<Type>`, this would not work since the scope operator inhibits the virtual dispatch mechanism. However, in our case `first` is a data member, not a member function, so it can be safely used.

2. Change `first` to `this->first`. Since the pointer to a class object `this` is always implicitly dependent in a template, `this->first` is now considered a dependent name and the lookup is therefore deferred until the template is actually instantiated, at which point all base classes are considered.

3. Insert `using linkedListType<Type>;` prior to using `first`. This ensures that the compiler examines `linkedList<Type>` when trying to resolve class objects, member functions, or data members.

See:

https://isocpp.org/wiki/faq/templates#nondependent-name-lookup-members

for details. In particular, look at:

"Why am I getting errors when my template-derived-class uses a member it inherits from its template-base-class?"

Newer C++ compilers (such as GCC V6.3.1, which we are using on the server), enforce these rules regarding nondependent and dependent names.

HOMEWORK #32 – Splitting Linked Lists

Derive a class `intLinkedList` from the class `unorderedLinkedList`:

```
class intLinkedList: public unorderedLinkedList<int>
{
     Public:
          Void splitEvensOddsList (
                    intLinkedList &evensList,
                    intLinkedList &oddsList);
```

```
            // Function to rearrange the nodes of the linked
            // list so that evensList consists of even
            // integers and oddsList consists of odd
            // integers.
            // Postcondition:  evensList consists of even
            // integers.  oddsList consists of odd integers.
            // The original list is empty
};
```

Write the function for `splitEvensOddsList()`. Note that this function does not create any new nodes – it only arranges the nodes of the original list so that nodes with even integers are in `evensList` and nodes with odd integers are in `oddsList`.

Write a program that uses class `intLinkedList` to create a linked list of integers and then uses the function `splitEvensOddsList` to split the list into two sublists.