Programming Assignment #1

Objective

This assignment will provide practice with a doubly-linked list, along with practice on the basic concept of an iterator over a container.

Task

For this assignment you will implement a templated doubly-linked list class, along with an associated iterator class for helping with generic list traversals. In a doubly-linked list, each node has a pointer to the next node AND a pointer to the previous node. The following starter files are provided for you:

- `tnode.h` -- fully defines a templated node class for use in a doubly-linked list. Note that the member data includes a data element, as well as pointers to the next and previous nodes.
- `tlist.h` -- provides the **declarations** of the template classes `TList` (the linked list class) and `TListIterator` (an associated class to help with list traversal).
- `driver.cpp` -- contains some sample test calls to the linked list, but should not be considered a complete or thorough set of tests. You will need to create more test cases to fully test your class features. This driver is provided to help illustrate some of the basic class features and concepts, including the use of iterators.
- `driver output.txt` -- contains the output from running the above sample driver program

Your job will be to finish this templated set of classes by defining each of the functions in the `TList` and `TListIterator` classes. These should be defined in a file called:

```
tlist.hpp
```

Note that there is already a `#include` at the bottom of `tlist.h` where your definition file will be brought in. This illustrates a pretty standard format for setting up a templated class.

Iterators

The small class called `TListIterator` is a helper class that can be used in conjunction with the linked list class. This is a common feature used in container classes like this. The purpose of an iterator is to provide a common and non-implementation-specific way of traversing a container, so that mulitple containers could potentially use common algorithms (like sorting and searching functions, for example). This will be explored more in the course. For the iterator class in this assignment, here is a brief sample use:

```
  // suppose that L is a linked list storing ints, and it has
  //   already been populated with the values 3, 6, 9, 12, 15, 18, 21

  // this call would retrieve a list iterator over the container L
  TListIterator<int> itr = L.GetIterator();

  // at this point, itr currently is positioned at the first element in
  // the list (the 3).

  int x = itr.GetValue();             // x would now store 3
  itr.Next();                         // itr has advanced to the 6
  int y = itr.GetValue();             // y would now store 6
  itr.Next();
  itr.Next();                         // we have now advanced to the 12
  int z = itr.GetValue();             // z now stores 12

  itr.Previous();                     // now we have moved backwards, to the
9
  int a = itr.GetValue();             // a stores 9.    etc.
```

This class essentially helps us walk through the linked list in a fairly easy way, with calls to `Next()` and `Previous()` to move around.

Program Details

Here are general descriptions of the two classes you are to define, along with a general description of each function's task.

1) class TList

The member data of this class consists of pointers to the first and last nodes, a size variable, and a dummy variable of type T that can be used for error-checking situations. Specifically, some of the functions specify to return a stored data item, but if you encounter a situation like an empty list or other situation where there would not BE a valid data item, you can return a reference to the dummy object instead. This is needed because some such functions are pass-by-reference (so that the retrieved item can be modified by the caller under normal situations).

Function descriptions

- **Default constructor** -- creates an empty linked list

- **TList(T val, int num)** -- creates a linked list containing "num" copies of the data element "val"

- **Clear** -- clear out the list, resetting it to an empty list

- **Big Five**
  - Destructor -- appropriate clean-up of list, no memory leaks
  - Copy constructor -- deep copy
  - Copy assignment operator -- deep copy
  - Move constructor -- constructor with standard move semantics
  - Move assignment operator -- assignment with standard move semantics
- **Accessors**
  - **IsEmpty** -- returns true if the list is empty, false otherwise
  - **GetSize** -- returns the size (number of data elements) in the list
  - **GetFirst** -- returns the data element in the first node (by reference)
  - **GetLast** -- returns the data element in the last node (by reference)

  Note that error situations in the last two functions would occur if the list was empty (this what the "dummy" item is for).

- **Endpoint insert/removes**
  - **InsertFront** -- insert the data (parameter) as the first node in the list
  - **InsertBack** -- insert the data (parameter) as the last node in the list
  - **RemoveFront** -- remove the first element in the list. If the list is empty, just leave it empty
  - **RemoveBack** -- remove the last element in the list. If the list is empty, just leave it empty
- **Iterator retrieval**
  - **GetIterator** -- create and return an iterator that is positioned on the first node in the list. If list is empty, return default iterator
  - **GetIteratorEnd** -- create and return an iterator that is positioned on the last node in the list. If list is empty, return default iterator

- **Insert (2 parameters)**
  The new data element (second parameter) should be inserted into the linked list just *before* the position given by the iterator (the first parameter). If the list is empty, just insert the single item. If the iterator doesn't refer to a node, insert the item at the end of the list.

- **Remove (1 parameter)**
  This function should remove the data item that is given by the iterator (the parameter). The function should return an iterator to the node that is **after** the one that was just deleted. If the initial list was empty, there's nothing to delete -- so just leave it empty and return a default iterator.

- **Print**
  Should print the entire list contents, front to back, separated by the delimiter given in the second parameter. This function may assume that the stored type T has an available insertion << operator available for printing. Print to the stream given in the first parameter.

- **`operator+`**
  This is a standalone function that should return a TList object that is the result of concatenating two TList objects together -- in parameter order. See driver.cpp program for examples.

2) class TListIterator

The TListIterator class has only one member data variable -- a pointer to the Node to which it currently refers (we'll call this the current node in the function descriptions below).

- **Default constructor** -- a default iterator should just store the nunll pointer internally.
- **HasNext** -- returns true if there is another node *after* the current node, false otherwise
- **HasPrevious** -- returns true if there is another node *before* the current node, false otherwise
- **Next** -- advances the iterator to the next node after the current one. Returns an iterator to the new position.
- **Previous** -- moves the iterator to the previous node (before the current one). Returns an iterator to the new position.

- **GetData** -- return the *data item* at the current node. If the iterator is not pointing to a node (i.e. null pointer), you can use the "dummy" that was defined previously. Note that this is a return by reference.

Define all of the functions for these two classes in the file `tlist.hpp`

3) Test Program

Create a test program of your own in the file **`mydriver.cpp`**. You can use my provided driver.cpp as a general model of how to populate a linked list. Your test program should contain the following tests/illustrations at a minimum:

- Tests of all of the functions
- Tests that involve Linked Lists of at least two different stored types. Suggested types: int, double, char, string
- At least 10 tests of each of the following
    - InsertFront
    - InsertBack
    - RemoveFront
    - RemoveBack
    - Insert (iterator-based)
    - Remove (iterator-based)

    The tests for each of these should not be all in a row for any single one. I.e. for best testing, make sure that insert/remove calls of a single function type are frequently interspersed between other types. (This way, a mistake in one will often be revealed by later calls to others)

- Clear illustrations of list contents with Print before and after major sets of insert/delete tests. Make your outputs readable and easy to follow for best testing/results.
- At least one test that uses an iterator to traverse the list front to back
- At least one test that uses an iterator to traverse the list back to front

4) makefile

Create a makefile that configures a build of both my provided driver (driver.cpp) and your test program (mydriver.cpp). i.e. when you type "make" in the directory, it should compile and build both executables. Make your executables named "driver.x" and "mydriver.x", respectively.

5) General Requirements

- Document your code appropriately so that it is readable and easy to navigate
- You may use standard C++ I/O libraries, as well as class libraries like `string`. You may **NOT** use any of the container class libraries from the STL
- If you wish to add any helper functions to the TList class, you may modify the tlist.h file for this purpose. But do NOT change any of the expected interface function prototypes or add extra member data. Any helper functions you write should be in the private section
- Make sure your files compile and run on linprog.cs.fsu.edu with g++, using the C++11 standard compilation flag. This is where they will be tested and graded

---

Deliverables and submitting

These are the deliverable files you should submit:

```
tlist.h
tlist.hpp
mydriver.cpp
makefile
```

To submit, package up your files in a compressed tar archive and upload this at the assignment submission link on the Canvas course site (in the "Assignments" section). Your tar file should be named in this format, all lowercase:

lastname_firstname_p1.gz.tar

Example: My tar file would be: **gaitros_david_p1.gz.tar**

Note that in addition to the provided test cases, we will also test your program/classes using additional test programs. Your program must be able to pass all such test cases to obtain a full score for the corresponding components.

This is an example of how to tar/zip your files if the command is executed in the directory where they are located:

tar -cvzf Gaitros_david_p1.gz.tar tlist.h tlist.hpp mydriver.cpp makefile

Grading Criteria:

- Critical Requirements – Grading cannot be done unless these are satisfied:
  - Uncorrupted _____.gz.tar file properly submitted on canvas
  - The program compiles using submitted makefile. If the program does not compile no further grading can be accomplished. Programs that do not compile will receive a zero.

- (25 Points) The program executes without exception and produces output.
- (25 Points) The program produces the correct output.
- (25 Points – 5 Points each) The program specifications are followed.
    - tlist.cpp complies with the requirements of the assignment
    - mydriver.cpp performs a non-trivial test and performs a test on all the member functions.
    - Lists correctly inserted
    - All parameters passed in must be passed by reference (address/pointers)
    - destructor properly implemented
- (10 Points)The program is documented (commented) properly.
- (5 Points)Use constants when values are not to be changed
- (5 Points)Use proper indentation
- (5 Points)Use good naming standards