Templates and STL Vectors

Let's start with a singly-linked-list class of strings. The code to free the list from memory is missing.

Here is the relevant code:

```
class Node
{
  private:
        string name;
        Node* next;
  public:
        Node();
        Node(string theName, Node* nextNode);
        string getName();
        void setName(string newName);
        Node* getNext();
        void setNext(Node *newNext);
};

Node::Node() : name(""), next(NULL)
{
}
Node::Node(string theName, Node* nextNode) :
        name(theName), next(nextNode)
{
}
string Node::getName()
{
        return name;
}
void Node::setName(string newName)
{
        name = newName;
}
Node* Node::getNext()
{
        return next;
}
void Node::setNext(Node *newNext)
{
        next = newNext;
}
```

```
class LinkedList
{
  private:
        Node* head;             // Points to the first thing in the list
  public:
        LinkedList();
        ~LinkedList();
        void add(string s); // Adds to front of list
        void remove(string s); // Deletes first occurrence of s in the list
        void output();  // Output everything in the list
};

LinkedList::LinkedList()
{
        head = nullptr;
}
LinkedList::~LinkedList()
{
        // TO DO: Free up the memory allocated to this linked list
}

// Adds a new node with this string to the front of the linked list
void LinkedList::add(string s)
{
        if (head == nullptr)
        {
                // New list, make head point to it
                head = new Node(s, nullptr);
        }
        else
        {
                // Make a new node that points to head
                Node *temp = new Node(s, head);
                // Set head to the new node
                head = temp;
        }
}

// Deletes first occurrence of s in the list
void LinkedList::remove(string s)
{
        Node *temp = head;
              Node *prev = nullptr;
        while (temp != nullptr)
        {
                if (temp->getName() == s)
                {
                        if (prev == nullptr)
                                        {
                                                // We're deleting the head of the list
                                                head = head->getNext();
                                                delete temp;
                                        }
                                        else
                                        {
                                                // We're deleting past the head
                                                prev->setNext(temp->getNext());
                                                delete temp;
```

```
                                        }
                        break;
                }
                                prev = temp;
                temp = temp->getNext();
        }
}

// Output everything in the list
void LinkedList::output()
{
        Node *temp = head;
        while (temp != nullptr)
        {
                cout << temp->getName() << endl;
                temp = temp->getNext();
        }
        cout << endl;
}


int main()
{
  LinkedList mylist;

  mylist.add("Andy");
  mylist.add("Bobo");
  mylist.add("Chuck");
  mylist.add("Drogo");
  mylist.add("Erin");
  mylist.add("Frodo");

  cout << "Output entire list" << endl;
  mylist.output();

  cout << "Output after removing Chuck" << endl;
  mylist.remove("Chuck");
  mylist.output();
  cout << "Output after removing Frodo" << endl;
  mylist.remove("Frodo");
  mylist.output();

  cout << "Output after removing Andy" << endl;
  mylist.remove("Andy");
  mylist.output();

  system("pause");
  return 0;
}
```

This program will create the list then output the contents after deleting Chuck, Frodo, and Andy.

This might be just what we want – a list of Strings – but what if we want a linked list of some other type, like an integer? Or how about a class, like Transaction or Pet or any of the other classes we've written

so far? We would have to change the Node class and the LinkedList class so it uses a different data type. But the data type would basically be the only change, how inefficient is that!

There is a solution, and it is to use **templates**.

If we put:

```
template <class T>
```

In front of a class, where T is a name we choose, but is typically T, then this is saying that we want to have a generic type T used in the class that we will fill in later when an instance variable of the class is created. Here we use "T" for the Node class. "string" was changed to "data" since it may not be a string:

```
template <class T>
class Node
{
  private:
        T data;
        Node* next;
  public:
        Node();
        Node(T theData, Node* nextNode);
        T getData();
        void setData(T newData);
        Node* getNext();
        void setNext(Node *newNext);
};

template<class T>                  // Have to insert this and use Node<T> as the type
Node<T>::Node() : next(NULL)
{
}
template<class T>
Node<T>::Node(T theData, Node* nextNode) :
        data(theData), next(nextNode)
{
}
template<class T>
T Node<T>::getData()
{
        return data;
}
template<class T>
void Node<T>::setData(T newData)
{
        data = newData;
}
template<class T>
Node<T>* Node<T>::getNext()
{
        return next;
}
template<class T>
void Node<T>::setNext(Node<T> *newNext)
{
        next = newNext;
}
```

When we create an instance of the Node class we can now stick the data type we want for the node in the declaration! Here is an example main:

```
Node<int> n1(3, nullptr);
Node<double> n2(34.10, nullptr);

cout << n1.getData() << endl;
cout << n2.getData() << endl;
```

This makes a Node that stores an int, and a separate node that stores a double. The rest of the code works the same!

There is an important caveat to make this code compile. There are problems separating out template classes into a separate header and implementation. The usual solution is to put all the template implementation into the header, or to include the .cpp file instead.

In our case in the main.cpp we could take out

#include “Node.h”

And replace it with

#include “Node.cpp”

Let’s modify the LinkedList class now so we can use the new Node class:

```
#include "Node.cpp"
#include <string>
using namespace std;
template<class T>
class LinkedList
{
  private:
        Node<T>* head;             // Points to the first thing in the list
  public:
        LinkedList();
        ~LinkedList();
        void add(T s); // Adds to front of list
        void remove(T s); // Deletes first occurrence of s in the list
        void output();  // Output everything in the list
};



#include "LinkedList.h"           // Implementation File for LinkedList.cpp
#include <iostream>
using namespace std;
template <class T>
LinkedList<T>::LinkedList()
{
        head = nullptr;
}
```

```
template <class T>
LinkedList<T>::~LinkedList()
{
	// TO DO: Free up the memory allocated to this linked list
}

// Adds a new node with this string to the front of the linked list
template <class T>
void LinkedList<T>::add(T s)
{
	if (head == nullptr)
	{
		// New list, make head point to it
		head = new Node<T>(s, nullptr);
	}
	else
	{
		// Make a new node that points to head
		Node<T> *temp = new Node<T>(s, head);
		// Set head to the new node
		head = temp;
	}
}

// Deletes first occurrence of s in the list
template <class T>
void LinkedList<T>::remove(T s)
{
	Node<T> *temp = head;
	      Node<T> *prev = nullptr;
	while (temp != nullptr)
	{
			// The code below requires == to be overrided for any class T
		if (temp->getData() == s)
		{
			if (prev == nullptr)
					{
						// We're deleting the head of the list
						head = head->getNext();
						delete temp;
					}
					else
					{
						// We're deleting past the head
						prev->setNext(temp->getNext());
						delete temp;
					}
			break;
		}
				prev = temp;
		temp = temp->getNext();
	}
}
```

```
// Output everything in the list
template <class T>
void LinkedList<T>::output()
{
        Node<T> *temp = head;
        while (temp != nullptr)
        {
                        // The code below requires << to be overridden for class T
                cout << temp->getData() << endl;
                temp = temp->getNext();
        }
        cout << endl;
}
```

Now our main looks like this:

```
  LinkedList<string> mylist;

  mylist.add("Andy");
  mylist.add("Bobo");
  mylist.add("Chuck");
  mylist.add("Drogo");
  mylist.add("Erin");
  mylist.add("Frodo");

  cout << "Output entire list" << endl;
  mylist.output();
```

If we want a linked list of int's we could do the following:

```
  LinkedList<int> intList;
  intList.add(34);
  intList.add(100);
  intList.add(25);
  intList.remove(100);
  intList.output();
```

This outputs 25 and 34.

In our example we had only one template parameter, but you can have more and separate them by a comma, for example, template<class T1,T2>.

Vectors

It turns out there is a huge library of useful data types called the Standard Template Library, or STL. You have already been using one of the classes in the STL, the string class. There is another super-useful class called vector. Internally, this uses something like a linked list but you can access it like an array. You can think of it as a dynamic array that can grow or shrink as needed, and it also lets you insert or remove items from the vector.

In its most basic operation, use push_back to add to a vector, and use [] to access a vector's contents. You need to have added to a vector before trying to access it with []. Here is an example:

To use the vector class:

```
#include <vector>
```

Here is a sample that demonstrates typical use:

```
vector<int> v1;
for (int i =0; i < 5; i++)
      v1.push_back(i);
for (int i = 0; i < v1.size(); i++)
      cout << v1[i] << " ";
cout << endl;

vector<string> v2;
v2.push_back("hello");
v2.push_back("there");
for (int i = 0; i < v2.size(); i++)
      cout << v2[i] << " ";
cout << endl;
```

The output is

```
0 1 2 3 4

hello there
```

We can put our own classes inside a vector if we want, e.g.:

```
vector<Transaction> v;

Transaction t;
v.push_back(t);
```

If we put a template class into a vector then you might try this:

```
vector<Node<string>> v3;
Node<string> n("hello",nullptr);
v3.push_back(n);
cout << v3[0].getData() << endl;
```

This runs fine in a C++11 compiler, but in older compilers you must put a space between the >>:

```
vector<Node<string> > v3;
```

This is one of those syntax idiosyncrasies that was finally fixed in C++!

There are other things we can do with vectors using iterators. An iterator is like a pointer to an element in a vector. v.begin() returns an iterator to the first element, while v.end() is an iterator to the end of the vector (past the last element). Use +1 to get to the next element, +2 to get two elements, etc.

For example, *(v.begin() +2) will reference the third element in the vector.

v.erase(v.begin() + 2) will erase the third element.

v.erase(v.begin(), v.end()) will erase everything in the vector.

Maps

The map class is another very useful STL class. It allows you to create an associative array, which is essentially a little database. Here is an example:

```
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main( )
{
    map<string, string> planets;

    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile per hour winds";
    planets["Pluto"] = "Dwarf planet";

    cout << "Entry for Mercury - " << planets["Mercury"]
         << endl << endl;

    if (planets.find("Mercury") != planets.end())
      cout << "Mercury is in the map." << endl;
    if (planets.find("Ceres") == planets.end())
      cout << "Ceres is not in the map." << endl << endl;

    // Iterator outputs planets in order sorted by key
    cout << "Iterating through all planets: " << endl;
    map<string, string>::const_iterator iter;
    for (iter = planets.begin(); iter != planets.end(); iter++)
    {
      cout << iter->first << " - " << iter->second << endl;
    }

    return 0;
}
```

Note that it is rather unwieldy to declare the type for the iterator. This is where **auto** comes into play. Instead of:

```
      map<string, string>::const_iterator iter;
      for (iter = planets.begin();  ...
```

We can use:

```
   for (auto iter = planets.begin(); …
```

This is also a good place to use the for each loop:

```
for (auto planet : planets)
{
    cout << planet.first << " " << planet.second << endl;
}
```

Other STL classes include:

set	- stores a set
pair	- 2-tuple of elements
list	- doubly linked list (vector is a dynamic array that grows/shrinks)
slist	- singly linked list
queue	- FIFO queue
stack	- LIFO
deque	- double ended queue, vector with insert/erase at beginning or end