Linked Implementation Of A Stack, Reading

The arrayed implementation of a stack is really quite easy and efficient. The only disadvantage is all that **excess capacity**, and pausing to double it when it becomes full. Also once a stack reaches its maximum size, with whatever capacity it takes to accommodate that size, that's the capacity for the rest of the stack's life. It does not auto-shrink the way it auto-grows, and if it did, it could very well spend too much time growing and shrinking in a real application.

Another way to store values in a stack is a **singly-linked list**. It would have **no excess capacity**, adding and removing nodes as necessary. Since there's no square bracket operator, there's no need to traverse and count in order to find a value at a given index, so a linked structure may offer a suitable alternative.

Singly-Linked Lists

If you're not up on what you learned about linked lists in your previous Comsc-110 and -165 classes, right now would be a good time to go back and check your notes and past assignments, because we expect that you know about nodes and head pointers and next pointers and the traversing for-loops with the "p" and optional "prev" pointers. *Refer to chapter 14 of our optional Burns textbook.*

In any case, we'll be applying linked list coding in the context of a **containing class**. That is, the node struct and the head pointer will all be *contained inside this templated class*. The public interface specification and its use in the main program is all the programmer knows or needs to know about the template. The public interface for linked stack and the arrayed stack would be expected to be *exactly the same* as each other.

Nodes

The node struct should be visible and useable only by the class. So you'll see a new thing now -- it's how to make a struct a (private) member of a class:

```
template <typename V>
class Stack
{
  struct Node
  {
    V value;
    Node* next;
  };

  Node* firstNode; // the head pointer
  int siz; // tracking the number of nodes

  public:
  ... // exactly the same as the arrayed version
};
```

A few things are different from what you probably studied previously about linked lists. First, the struct for the Node is inside a class. Then the name of the struct is made generic -- "Node". Finally, the "attributes" are not listed. Instead the object with *its* attributes is embedded in the Node. To **compare and contrast** with prerequisite knowledge:

```
struct Student
{
  string name;
  int studentID;
  Student* next;
};
```

```
// Stack.h H file

...

template <typename V>
Stack<V>::Stack( )
```

```
// main program's CPP file

#include "Stack.h"

struct Student
{
```

```
int main( )
{
  // create an empty list
  Student* firstStudent = 0;
  ...

  // create and add a Student

  Student* temp = new Studen
t;
  temp->name = ...
  temp->studentID = ...
  temp->next = firstStudent;
  firstStudent = temp;
  ...
}
```

becomes

```
{
  firstNode = 0;
  ...
};

template <typename V>
void Stack<V>::push(const V& valu
e)
{
  Node* temp = new Node;
  temp->value = value;
  temp->next = firstNode;
  firstNode = temp;
  ...
}
```

```
  string name;
  int studentID;
};

int main( )
{
  Stack<Student> students;
  ...

  Student x = {"Joe", 123456
7};
  students.push(x);
  ...
}
```

With a generic node struct, the template's H does not need to know the attributes of whatever object it stores. In fact, it does not even have to be an object -- this works with **Stack<double> temperatures;**. Further, the main program does not have to deal with dynamic memory allocation of its nodes, because its objects get *copied* into dynamically-allocated nodes in **class Stack**. The Student struct in the main program no longer even has a next pointer -- the main program has *no idea* that its Stack object is using a linked list!

That takes care of the private data members and the public interface (which is the same as the arrayed stack). All that's left is to rewrite the templated member functions so that they use the linked list instead of the array.

Main Constructor

The constructor no longer has a reason for a parameter, because capacity is no longer a concept. But it is still needed because it has to (a) set the head pointer to null and (b) initialize the siz to zero.

```
Stack( ); // prototype
```

Destructor

There is no need for a **delete [ ]** statement since there's no **new [ ]** anywhere. But there are **new Node** statements, and each "new" needs a "delete". Before (in your prerequisite course) it was the main program's job to deallocate nodes. Now it's the destructor's job. To compare and contrast:

```
  // in the main program CPP
  while (firstStudent)
  {
    Student* p = firstStudent;
    firstStudent = firstStudent->next;
    delete p;
  }
```

becomes

```
template <typename V>
Stack<V>::~Stack( )
{
  while (firstNode)
  {
    Node* p = firstNode;
    firstNode = firstNode->next;
    delete p;
  }
}
```

Sorry -- it's now **too long to be written inline** as it was for the arrayed stack. In any case there's nothing to call in the main program, because the destructor is called *automatically* when the stack object itself goes out of scope or the program ends.

Copy Constructor And Assignment Operator

It's not all that instructive to go into the development of the algorithm and code for these functions, so we'll take a different approach. They apply to any singly-linked structure, with slight modifications for the naming of data members:

```
template <typenameV>
Stack<V>::Stack(const Stack<V>& original)
{
  firstNode = 0;
  Node* lastNode = 0; // temporary tail
  siz = original.siz;
  for (Node* p = original.firstNode; p; p = p->
next)
  {
    Node* temp = new Node;
    temp->value = p->value;
    temp->next = 0;
    if (lastNode) lastNode->next = temp;
    else firstNode = temp;
    lastNode = temp;
  }
}
```

In the copy constructor above, the original list is traversed in order to access the data stored in its nodes. Then new nodes are allocated, filled with copies of the stored values, and added to the *end* of the list, so that the order of data values is preserved. Note the introduction of a *temporary tail pointer* to facilitate the appending of nodes.

The assignment operator (to the right) basically combines the destructor and the copy constructor. Again, the key to the whole thing is that *temporary tail*. The stack's private data members do not include a tail because it's not needed anywhere but in these functions.

```
template <typename V>
Stack<V>& Stack<V>::operator=(const Stack<V>& original)
{
  if (this != &original)
  {
    // deallocate existing list
    while (firstNode)
    {
      Node* p = firstNode->next;
      delete firstNode;
      firstNode = p;
    }

    // build new queue
    Node* lastNode = 0; // temporary tail
    for (Node* p = original.firstNode; p; p = p->next)
    {
      Node* temp = new Node;
      temp->value = p->value;
      temp->next = 0;
      if (lastNode) lastNode->next = temp;
      else firstNode = temp;
      lastNode = temp;
    }
    siz = original.siz;
  }
  return *this;
}
```

Pop And Clear

These are simple one-liners in the arrayed version, but there are nodes to deallocate in the linked version, so **no more inline functions** for these two. The easier one is Stack::clear, because we already wrote the destructor and Stack::clear does exactly what the destructor does -- plus one detail -- it has to set "siz" to zero.

Actually, we could **just have the destructor call Stack::clear**, couldn't we...

Stack::pop is actually just like Stack::clear, except that instead of deallocating *all* the nodes, it just deallocates the head, if there is one. For that, we just have to replace "while" with "if".

```
template <typename V>
void Stack<V>::clear( )
```

```
template <typename V>
void Stack<V>::pop( )
```

```
{
  while (firstNode)
  {
    Node* p = firstNode;
    firstNode = firstNode->next;
    delete p;
    --siz;
  }
}
```

```
{
  if (firstNode)
  {
    Node* p = firstNode;
    firstNode = firstNode->next;
    delete p;
    --siz;
  }
}
```

Note that in **Stack::clear**, there's no need to set the *head pointer* to zero -- it comes out of the while-loop as zero. **siz** ends up zero too. That's convenient!

Push

Didn't we write this one already? Inside the copy constructor and assignment operator loops? All that's missing is the adjustment to the size counter:

```
++siz;
```

Peek

The peek function should return a mutable reference to the "data" attribute of the head node, like this:

```
return firstNode->value;
```

But just in case the stack is empty, and "firstNode" is null, we need a backup plan. There's no index zero to return as there was with the arrayed stack, so we're back to the dummy that we used in the StaticArray and the DynamicArray.

```
if (firstNode == 0) return dummy;
```

Remember to add "dummy" to the list of private data members -- and *resist any urge to initialize it to zero* in the main constructor!

Capacity

As we said earlier in this reading, there is no longer a concept of capacity. So any reference to a capacity tracking private data member, or to setters and getters, go away. No **int cap;**, no **Stack::capacity**.