Project 1: Llama Stacks

Due: Tuesday, October 4, 8:59:59pm

Corrections and Addenda

* [09/15/2016] pop() should throw a LlamaUnderflow exception if the stack is empty.
* [09/15/2016] Add to Llama.h
* #include <string>
* [09/19/2016] Depending on how you implement the assignment operator, your output for driver2.cpp and driver3.cpp may be different from the sample output shown. In particular, you have to decide what to do with the existing nodes in the linked list of the left hand side of the assignment operator. Do keep in mind that the left hand side and the right hand side may have different number of nodes in the linked list. I.e., it is perfectly legitimate to have the assignment:
* LlamaStackOne = LlamaStackTwo ;

where LlamaStackOne has 14 nodes and LlamaStackTwo has 6 nodes or where LlamaStackOne has 4 nodes and LlamaStackTwo has 24 nodes.

How you decide to handle these situations can change the number of LlamaNodes created and destroyed. It's OK if your numbers are different after assignment. It is much more important that your program does not seg fault and does not leak memory.

* [9/19/2016] No, you can't use vector or list or any other data structure from STL. You can use string that's it.
* [9/19/2016] No, you can't use C++11 and C++14 features. Your program must compile on GL using g++ without any flags.

Objectives

The objective of this programming assignment is to have you review C++ programming using following features: object-oriented design, dynamic memory allocation, pointer manipulation, exceptions and templates.

Introduction

The premise of this assignment is that you are using a system where calls to new and delete for dynamic memory allocation and de-allocation are very slow. They are slow enough that it is worth your while to consider data structures that reduce the number of calls to new and delete.

The data structure we are considering is a stack. This can be implemented as an array or as a linked list with advantages and disadvantages to both. However, your application does not allow you to determine the maximum size of the stack. If you implemented the stack using an array, you may run out of storage if the stack exceeds the size of the array. You can dynamically allocate a bigger array and copy the old stack, but the time for copying is prohibitively slow. Using a standard linked list would involve a call to new or delete for each push or pop instruction. This is also too slow.

Instead of a simple linked list, we will use a linked list where each node contains an array of items. The array will be of fixed size, say for example, 100. This linked list will start out with a single node that holds an array of 100 items. So, a single node can hold up to 100 items of the stack. That means we can push items on the stack and pop items off the stack without calling new or delete until the stack grows larger than 100. If that happens, we will add a node to the linked list. The result is a stack that holds 200 items. Adding space for 100 more items involved just one call to new.

There's a catch. Suppose our stack has 101 items and we do a pop. Now we only have 100 items, what do we do with the linked list? We can remove and deallocate the node that used to hold items 101 through 200. If we do that, what happens if we push right after the pop? We would have to add a node to the linked list again. If there's a sequence of push and pop instructions that causes the stack size to flip between 100 and 101, then we are left with the situation that every push and pop instruction results in a call to new or delete. Then there would be no advantage to our new data structure.

The solution is that we should not immediately remove a node when our stack size drops from 101 to 100. Instead, we should wait until the stack size drops much further, until it drops to 50 items (which requires at least 50 pop instructions to achieve). When the stack size is between 100 and 51, we will keep the "extra" node around in case the stack size grows to 101 again. Then we can use the space in the extra node and bypass the need to call new.

To recap, if our stack size reaches 101 for the first time, we add a node to the linked list. The stack size can increase and decrease, but we do not deallocate a node until the stack size drops to 50. Also, we won't have to allocate a new node until the stack size increases past 200. In general, we follow this scheme whenever a node is filled --- i.e., when the stack size grows past 200, 300, 400, ... This strategy prevents us from having to call new or delete very often. It also allows our stack to grow and shrink without having to copy the entire stack to a new location.

Let's call this data structure a Llama Stack (short for Linked-List Array Mixed Amalgamated Stack).

Assignment

Your assignment is to implement *and test* a templated Llama Stack in C++. You should use the following two header files:

#ifndef \_LLAMANODE\_H\_

#define \_LLAMANODE\_H\_

/\* File: LlamaNode.h

UMBC CMSC 341 Fall 2016 Project 1

This file has the class declaration for the LlamaNode class

for Project 1. See project description for details.

This file should not be modified in ANY way.

When your program is graded, it will be compiled with the original

version of this file. Your program must work with the original.

\*/

#include <iostream>

using namespace std ;

template <class T, int LN\_SIZE> // forward class declaration

class Llama ;

template <class T, int LN\_SIZE>

class LlamaNode {

friend class Llama<T,LN\_SIZE> ;

public:

LlamaNode() ;

~LlamaNode() ;

static void report() ;

private:

static int newCount ; // # of times constructor was called

static int deleteCount ; // # of times desctructor was called

T arr[LN\_SIZE] ;

LlamaNode \*m\_next ;

} ;

#include "LlamaNode.cpp"

#endif

#ifndef \_LLAMA\_H\_

#define \_LLAMA\_H\_

/\* File: Llama.h

UMBC CMSC 341 Fall 2016 Project 1

This file has the class declaration for the LlamaNode class

for Project 1. See project description for details.

You may add public and private data members to the Llama class.

You may add public and private member functions to the Llama class.

\*/

#include <stdexcept>

#include <string>

#include "LlamaNode.h"

using namespace std ;

class LlamaUnderflow : public std::out\_of\_range {

public:

LlamaUnderflow(const string& what) : std::out\_of\_range(what) { }

} ;

template <class T, int LN\_SIZE>

class Llama {

public:

Llama() ;

Llama(const Llama<T,LN\_SIZE>& other) ; // copy constructor

~Llama() ;

int size() ;

void dump() ;

void push(const T& data) ;

T pop() ;

void dup() ; // (top) A B C D -> A A B C D

void swap() ; // (top) A B C D -> B A C D

void rot() ; // (top) A B C D -> C A B D

T peek(int offset) const ;

// overloaded assignment operator

//

const Llama<T,LN\_SIZE>& operator=(const Llama<T,LN\_SIZE>& rhs) ;

//

// Add your public member functions & public data mebers here:

//

private:

//

// Add your private member functions & private data mebers here:

//

} ;

#include "Llama.cpp"

#endif

The implementation of the LlamaNode class is trivial and has been done for you. (See [LlamaNode.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/LlamaNode.cpp).) In fact, you are not allowed to change the LlamaNode files in any way.

Your assignment is to implement the member functions of the Llama class. You will need to add data members and member functions to the Llama class, but the rest of the header file should not be changed. You must implement the following member functions:

* Llama() ;
* This is the default constructor for the Llama class. Since you are picking the data members for this class, much of this code is determined by you. At the very least, the default constructor must create a Llama Stack that holds one node.
* Llama(const Llama<T,LN\_SIZE>& other) ;
* This is the copy constructor for the Llama class. It should make a complete copy of the Llama Stack given in the parameter. The target of the copy is the host object.

You should not call the assignment operator from the copy constructor (or vice versa). The objective of these two member functions are sufficiently different that you should implement them separately. If you do not want to duplicate similar code in the two functions, make a third function that you can call from the copy constructor and from the assignment operator.

* ~Llama() ;
* This is the destructor for the Llama class. All dynamically allocated memory associated with the host object must be deallocated. You should use valgrind on GL to check for memory leaks.
* int size() ;
* This function returns the number of items in the Llama Stack.
* void dump() ;
* This member function is used for debugging. It should print out all pertinent information regarding the host Llama stack to cerr, including the number of LlamaNodes that have ever been created and destroyed. It should also print out the address of each node and whether there is currently an "extra" node in the data structure.

See the sample outputs below for the suggested format of the output of the dump() function.

* void push(const T& data) ;
* The push() member function adds data to the top of the stack. A copy of data should be made.
* T pop() ;
* The pop() member function removes and returns the item at the top of the Llama Stack.

If the stack is empty, pop() should throw a LlamaUnderflow exception. (The LlamaUnderflow class is defined in [Llama.h](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/Llama.h).)

* void dup() ;
* The dup() member function duplicates the top of the stack. For example, suppose the stack originally held A B C D (where A is at the top). Then, after calling dup(), the stack should become A A B C D.

If the stack is empty, dup() should throw a LlamaUnderflow exception.

* void swap() ;
* The swap() member function exchanges the top two items of the stack. For example, suppose the stack originally held A B C D (where A is at the top). Then, after calling swap(), the stack should become B A C D.

If there are fewer than two items in the stack, swap() should throw a LlamaUnderflow exception.

* void rot() ;
* The rot() member function permutes the top three items of the stack. For example, suppose the stack originally held A B C D (where A is at the top). Then, after calling rot(), the stack should become C A B D.

Note: "rot" is short for "rotate".

If there are fewer than three items in the stack, rot() should throw a LlamaUnderflow exception.

* T peek(int offset) const ;
* The peek() member function returns the value of an item in the stack. The offset is used to determine which item is retrieved: peek(0) is the top of the stack, peek(1) is the first item below the top, ... The stack should not be modified in any way after a call to peek(). If the offset is too large, peek() should throw a LlamaUnderflow exception.
* const Llama<T,LN\_SIZE>& operator=(const Llama<T,LN\_SIZE>& rhs) ;
* This is the overloaded assignment operator. The assignment operator should handle the case of self-assignment. Any existing dynamically allocated memory in the host object must be deallocated beforehand. A complete copy of the right hand side of the assignment should be placed in the host object. This should handle the possible existence of an "extra" node in rhs.

You should not call the assignment operator from the copy constructor (or vice versa). See note in copy constructor description.

Finally, you must write a main() function that should be placed in a file named Driver.cpp. Your main() function should thoroughly test your implementation.

Files and Sample Runs

Here are some files for you to download:

* [LlamaNode.h](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/LlamaNode.h)     (do not change!)
* [LlamaNode.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/LlamaNode.cpp)     (do not change!)
* [Llama.h](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/Llama.h)

The following three test programs may be used to check the compatibility of your implementation. These programs do not check the *correctness* of your implementation. Even if your implementation compiles and runs correctly with these programs, it does not mean your implementation is error-free. Grading will be done using programs that exercise your implementation much more thoroughly. You must do the testing yourself --- testing is part of programming. Conversely, if your implementation does *not* compile or does *not* run correctly with these test programs, then it is unlikely that it will compile or run correctly with the grading programs.

* [driver1.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver1.cpp): uses Llama Stack with strings.   
  Output from sample run: [driver1.txt](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver1.txt).   
  Output from sample run with valgrind: [driver1v.txt](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver1v.txt).
* [driver2.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver2.cpp): uses Llama Stack with float. Also catches an exception.   
  Output from sample run: [driver2.txt](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver2.txt).   
  Output from sample run with valgrind: [driver2v.txt](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver2v.txt).
* [driver3.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver3.cpp): uses Llama Stack with a class that has dynamically allocated data.   
  Compile with [OvCoInt.h](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/OvCoInt.h) and [OvCoInt.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/OvCoInt.cpp) (Note: OvCoInt = "Overly Complicated Integer").   
  Output from sample run: [driver3.txt](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver3.txt).   
  Output from sample run with valgrind: [driver3v.txt](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver3v.txt).

Implementation Notes

* Yes, you can have int template parameters.
* Do not put a main() function in Llama.cpp. Your main() function should go in a separate file called Driver.cpp.
* Remember that your Llama.cpp file must also be guarded. This is because the Llama.h file includes Llama.cpp (and Llama.cpp also includes Llama.h). This arrangement is necessary for C++ templates.
* Since a LlamaNode can be quite large, your linked list should not use a dummy header node. They are not very useful for stacks anyway.
* Except for dump(), peek(), the copy constructor, the destructor and the assignment operator, all member functions should run in O(1) time. I.e., the running time for the member function should not depend on the size of the stack. (We will consider LN\_SIZE to be a constant.)
* You will need a consistent design philosophy. You will need data members that determine where the top of the stack is, how many items there are in the stack, whether there is currently an "extra" node... You should think a lot about how, when and where these data members are updated and kept consistent.
* There should be at most one extra node in the data structure at any time. (I.e., you can't just never delete LlamaNodes.)
* Recommendation: Use the C/C++ assert macro facility to test your design. To do this:
* #include <assert.h>

In your program, you can use assert() with a Boolean expression that "should" be true. For example, if you think that a pointer, ptr, should always be non-NULL at a particular point in your code, then you can use:

assert( ptr != NULL ) ;

If ptr is indeed not NULL, nothing happens. On the other hand, if ptr is NULL, then the program will stop and print out the line number where the assertion failed.

For example, the third sample driver above, [driver3.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/driver3.cpp), uses the Overly Complicated Integer class (see [OvCoInt.h](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/OvCoInt.h)and [OvCoInt.cpp](https://www.csee.umbc.edu/~chang/cs341.f16/projects/p1files/OvCoInt.cpp)) which has a dynamically allocated int. The pointer to this location is never supposed to be NULL. The constructor for OvCoInt allocates space and it *shouldn't* ever be deallocated until the destructor is invoked. The assert() macro is used throughout to check.

The assert() macro is a debugging tool used during development. It is also an example of defensive programming. After programs are fully debugged, the assert() macro can be turned off by compiling with the -DNDEBUG flag. So, you do not need to worry about efficiency and there is no need to comment out all the assert() calls when you are done debugging.

* Do not implement dump() using peek(). The dump() function should be simple and bug free. Use dump() to debug more complicated functions like peek().
* Approach this assignment using incremental development. Implement push() and dump() first. Make sure that push()and dump() are working. Then implement pop(). Make sure you are handling the "extra" node correctly before working on the copy constructor and the assignment operator.
* Along the lines of incremental development, try working without templates first, because non-templated code is easier to debug. Remove or comment out the template syntax and add the following macros to

Llama.h

#define T string

#define LN\_SIZE 4

After that, the identifiers T and LN\_SIZE will be replaced with "string" and "4" respectively. After verifying your code works with strings, check if it works with int. Then, check it with OvCoInt. (Just change the #defines.) Finally, check if it works with different values of LN\_SIZE. After that, making templates from your code is mostly a typing exercise.

* Test your programs for memory leaks (dynamically allocated memory that was never released) using valgrind. The valgrind command is available on GL. Just compile your test program and run:
* valgrind ./Driver.out

This is assuming that your executable file is named Driver.out. If that run did not leak memory, the output from valgrind will say:

All heap blocks were freed -- no leaks are possible

As usual, the fact that a single run of your implementation did not leak memory does not mean that it will never leak memory.

What to Submit

You must submit the following files to the proj1/src directory.

* Llama.h
* Llama.cpp
* Driver.cpp

You do not need to submit LlamaNode.h and LlamaNode.cpp because those files should not have changed. If you do happen to place a copy of LlamaNode.h and/or LlamaNode.cpp in your submission directory, they will be replaced by a copy of the original version.

If you followed the instructions in the [Project Submission](https://www.csee.umbc.edu/~chang/cs341.f16/projects/submission.shtml) page to set up your directories, you can submit your code using these Unix commands.

mkdir ~/cs341proj/proj1/src

cp Llama.h Llama.cpp Driver.cpp ~/cs341proj/proj1/src/

Note: you only have to use mkdir to make the src subdirectory once. If you submit again, just use the cp command.

Discussion Topics

Here are some topics to think about:

* What if LN\_SIZE is 1?
* What if LN\_SIZE is bigger than the number of items you will ever store in the stack?
* The LlamaNode class does not have a copy constructor or an overloaded assignment operator. Why is this OK?