Assignment 8 – MiniStack

CS 244

Due: April 3, 2014, 11:59 PM

Late penalties will be as described in the syllabus.

Overview

Create a C++ class MiniStack, an auxiliary function file, and a main program to test them. There is a worksheet (ungraded) with this. At the end of the worksheet there is a list of things you must do (with your code). Also complete the below steps and follow all other directions.

General Objectives:

Explore: How to implement a template class MiniStack using the std::vector

First Step

In Linux in your …/Documents/Programs folder
Create a folder named ***A08***

This is necessary as you will compress the folder and its contents for submission.

Second Step Check D2L

There will be starter code for this assignment. Likely posted near where this document was found. It may make things easier, and some of the files are explicitly required.

The files you will need for this assignment MUST be named:

MiniStack.h **MiniFuncs.cpp** **MiniTester.cpp**
MiniFuncs.h **MyExcept.h** **makefile**

with the corresponding classes properly defined and declared in each and your main() function located in MiniTester.cpp. The files: MiniFuncs.h, MyExcept.h and makefile will be provided for you and ***should not*** require any changes. But include them in your turn-in submission.

All files (excluding makefile, MiniFuncs.h, and MyExcept.h) should have comments with the file name, your name, the last modified date, and a description of the file contents. Each function should also be appropriately commented in the header and implementation files. The body of each function may need comments for the less than obvious parts of the code (if any).

This assignment begins with background material and a worksheet (the worksheet will *not* be graded). The program code you turn in will be graded.

-=-=-

Assignment 8 – MiniStack

CS 244

Building new Implementations using Existing Classes

One of the principles of good programming is to *reuse* existing code whenever practical. If you can reuse existing code, you don't need to spend the time to rewrite it. Code used previously has also been debugged, and will likely contain fewer errors. One of the easiest ways to create a container is to leverage an existing data type to build a new abstraction. In this lesson we will illustrate this process by building a **Stack** that will use a **std::vector** to store values as shown in the header:

```
template <typename T>
class MiniStack
{
   public:
         // constructor. create an empty stack
      MiniStack();
         // push: push (insert) item onto the stack.
         // Postcondition: the stack has a new topmost element and
         // the stack size increases by 1
      void push(const T& item);
         // pop: remove and return the item from the top of the stack.
         // Precondition: the stack is not empty.
         // if stack empty, throws the underflowError exception
      T pop();
         // top: return a reference to the element on the top of the stack.
         // Precondition: the stack is not empty.
         // if stack empty, the function throws underflowError exception
      T& top();
         // const top: constant version of top()
      const T& top() const;
         // empty: determine whether the stack is empty (size of 0)
      bool empty() const;
         // size: return the number of elements in the stack
      int size() const;

   private:
         // a private std::vector object maintains the stack items and size
      vector<T> m_stackVector;
};
```

Encapsulation and *information hiding* are two important object-oriented programming strategies that describe the goal of software reuse. Information hiding refers to the purposeful ignoring of details, so that other features can be considered more carefully. In our Stack implementation, we will ignore many vector methods and use only the functions helpful in our implementation.

Encapsulation means to conceal, or encapsulate, details within a data structure so that they do not need to be considered by users of the abstraction. The vector, for example, encapsulates the problems of memory management. The user of the vector need not know that the internal array will sometimes be resized. They can simply view the vector as a convenient storage container. By building a stack with a vector, the user need not worry about resizing the stack.

A stack requires the **operations: push, pop and top** to push an item onto the top of a stack, pop an item off the top of the stack, and obtain an item on top of the stack.

PLAN BEFORE YOU CODE

Assignment 8 – MiniStack


```
// push item on the stack by inserting it at
// the rear of the vector
template <typename T>
void MiniStack<T>::push(const T& item)
{









}

// if (empty())   throw underflowError("MiniStack pop(): stack empty");
// pop the stack by REMOVING and RETURNING the item at the rear of the vector
// This may deviate from your book's presentation. Do it this way anyway.
template <typename T>
T MiniStack<T>::pop()
{









}

// if (empty())   throw underflowError("MiniStack top(): stack empty");
// the top of the stack is at the rear of the vector
// return the element at the rear of the vector
template <typename T>
T& MiniStack<T>::top()
{









}
```

Assignment 8 – MiniStack

PLAN BEFORE YOU CODE

```
// constant version of top() with same internal method code
template <typename T>
const T& MiniStack<T>::top() const
{



}

// The constructor has nothing to do as the vector is already constructed.
// The constructor of std::vector initializes m_stackVector to be empty
template <typename T>
MiniStack<T>::MiniStack()
{



}

template <typename T>
bool MiniStack<T>::empty() const
{



}
template <typename T>
int MiniStack<T>::size() const
{



}
```

Based on implementation, fill in the following table of algorithmic execution times:
This may show up later in more important things like tests or quizzes

Operation	Execution time
`void push(const T& item);`	O(
`T pop();`	O(
`T& top();`	O(
`bool empty() const;`	O(
`int size() const;`	O(

Assignment 8 – MiniStack

CS 244

Tasks to complete after downloading starter code and placing it in folder named A08:
Be sure to read entire document for all directions and activities.

1. Implement the MiniStack methods in the header file (MiniStack.h) just after the full header declaration. The header and implementation will both be in file MiniStack.h, as **template classes must be in the same file** (which is strange but is a C++ thing).
2. In the MiniTester.cpp file create/fill in the main() function to test your implementation of the MiniStack class. You may also want to test the functions of part 3 below (#include "MiniFuncs.h" and call the functions from your main() function)
3. In the **MiniFuncs.cpp** file, create 3 functions as follows:
    a. ***bool isPalindrome(string str)*** to see if a string is the same forwards and backwards. This must be determined with the use of your MiniStack class to reverse the string. You should ignore case. So "Step on no Pets" should return true. You will also need to remove whitespace so that the string "no sam mason" returns true.

    b. ***bool balancedParentheses(string equation)*** in the tester file that accepts a mathematic equation and determines if there is a matching right parenthesis for each left parenthesis. The method returns true if they are balanced and false if they are not balanced. You must use your MiniStack class to determine this.

    c. ***bool balancedBlocks(string code)*** that accepts code as a string to see if the { ([]) } are matched using a stack. They should not be intermixed so that this string { ([ )] } should fail.
    If the string fails, return false.
    i.e. if NOT balanced return false
    else return true

    For example: The above string { ([ )] } should return false

    For example: If a brace is not closed, as in {([]), should return false.

    For example: The string: (({ }) [ ]), should return true

    Possible Input: could be of the form

    if(a<b[2] && (c+d) < e) { v.pop(); return (((a+b)*c)-d) }

    with all but the { ([ ]) } characters ignored.

This will be subject to automated grading (at least partially), so function and filenames matter.

Assignment 8 – MiniStack

CS 244

-=-=-=

Grading

100 points possible
At least 80 of these points will be determined by an "automatic grading" program.

The other 20 will be determined from code examination and/or further automated grading.

The majority of the points will be for correctly implementing the template MiniStack class. The functions in MiniFuncs.cpp WILL be tested and count for points. Not as many points will be given for your main() testing function. That function and the majority of the testing cpp file is for you to estimate what your grade will be before turning it in. It is likely a separate testing file created by the instructor will be used to test your class(es). The better you write your own testing program the more likely you will score well on the instructor's.

Bonus Points

There may be in-class work that can be submitted for bonus points to be applied to this assignment. However, total score will not be increased beyond the 100 points possible.

Turn-In Directions

Correctly submitting your work is worth 0 points,
but if not done correctly will likely result in nothing to grade.

Preparation

In Ubuntu Linux browse to your A08 folder
Make sure your source code files are in the folder
Right click on the A08 folder,
Set the file name to be A08_*yourlastname*.tar.gz
where *yourlastname* is your last name
Example: if your last name is Gollygee
then the filename would be A08_Gollygee.tar.gz
Select compress
This should create the file named A08_ *yourlastname*.tar.gz

Submit

Submit the A08_ *yourlastname*.tar.gz file
to the correct course drop box in D2L