cs124 Chapter 5 Stacks Assignment

David Topham

March 26, 2012

Contents

1 5.1 Stack ADT 1

2 5.2 Balanced Parenthesis - Self-Check exercise 2 4

3 5.3 Implementation - linked stack size function 5

4 5.4 Postfix Expressions - unit test expression code 9

1 5.1 Stack ADT

Here is stack.h:

```
#ifndef STACK_H_
#define STACK_H_
/** Definition file for class KW::stack */
// Include directives needed by the implementation
#include <stdexcept>
#ifdef USELIST
#include <cstddef>
#else
#include <vector>
#endif
#include <algorithm>
#include <string>
#include <sstream>
namespace KW {
  /** A stack is a data structure that provides last-in first-out
      access to the items that are stored in it.  Only the most recently
      inserted item is accessible.
  */
  template<typename Item_Type>
    class stack {

    public:
      // Constructor and member functions
      /** Constructs an initially empty stack.  */
      stack();

      /** Pushes an item onto the top of the stack.
          @param item The item to be inserted
      */
      void push(const Item_Type& item);

      /** Returns a reference to the object at the top of the stack
          without removing it.
          @return A reference to the object at the top of the stack
      */
      Item_Type& top();

      /** Returns a const reference to the object at the at the
          top of the stack without removing it.
          @return A const reference to the object at the top of the stack
      */
      const Item_Type& top() const;

      /** Removes the top item from the stack.  */
      void pop();

      /** Determines whether the stack is empty.  */
      bool empty() const;

      /** Returns the number of items in the stack.  */
      size_t size() const;




    private:
      // Data fields
#ifdef USELIST
      // Insert definition of node here
#include "Node.h"
      /** A pointer to the top of the stack */
      Node* top_of_stack;

#else
      /** A sequential container to contain the stack items */
      std::vector<Item_Type> container;
#endif
```

```
  }; // End class stack

  // Insert implementation of member functions here
#ifdef USELIST
#include "Linked_Stack.tc"
#else
#include "Stack.tc"
#endif

} // End namespace KW
#endif
```

2 5.2 Balanced Parenthesis - Self-Check exercise 2

Read the Case Study on “Balanced Parenthesis” in section 5.2, then do Self-Check exercise 2.

paren_checker.cpp

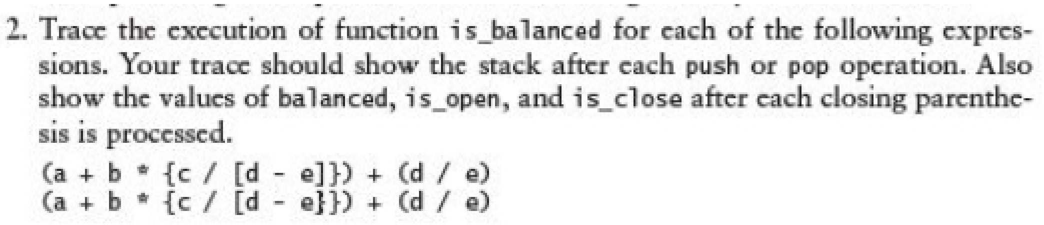
```cpp
/** Program to check an expression for balanced parentheses. */
#include <stack>
#include <string>
#include <iostream>
using namespace std;
// The set of opening parentheses.
const string OPEN = "([{";
// The corresponding set of closing parentheses.
const string CLOSE = ")]}";

bool is_open(char ch) {
  return OPEN.find(ch) != string::npos;
}

bool is_close(char ch) {
  return CLOSE.find(ch) != string::npos;
}

bool is_balanced(const string& expression) {
  // A stack for the open parentheses that haven’t been matched
  stack<char> s;
  bool balanced = true;
  string::const_iterator iter = expression.begin();
  while (balanced && (iter != expression.end())) {
    char next_ch = *iter;
    if (is_open(next_ch)) {
      s.push(next_ch);
    } else if (is_close(next_ch)) {
      if (s.empty()) {
        balanced = false;
      } else {
        char top_ch = s.top();
        s.pop();
        balanced =
          OPEN.find(top_ch) == CLOSE.find(next_ch);
      }
    }
    ++iter;
  }
  return balanced && s.empty();
}
```

Table 1: trace of in_balanced for expression $(a + b * c/[d - e] + (d/e)$

push or pop	balanced	is_open	is_close
push	false	true	false

Main function to test is_balanced

```
test_paren_checker.cpp
.................................................................................
#include <iostream>
using namespace std;
bool is_balanced(const string& expression);
int main() {
  cout << "Enter an expression\n";
  string expression;
  while (getline(cin, expression) && (expression != "")) {
    cout << expression;
    if (is_balanced(expression)) {
      cout << " is balanced\n";
    } else {
      cout << " is not balanced\n";
    }
    cout << "Enter another expression: ";
  }
  return 0;
}
```



2. Trace the execution of function `is_balanced` for each of the following expressions. Your trace should show the stack after each push or pop operation. Also show the values of `balanced`, `is_open`, and `is_close` after each closing parenthesis is processed.

```
(a + b * {c / [d - e]}) + (d / e)
(a + b * {c / [d - e}}) + (d / e)
```

(use latex table to show trace)

(section we worked on in class)

(end section we worked on in class)

3 5.3 Implementation - linked stack size function

Read section 5.3 and do Programming exercise 1 on page 332

The source code files from the author have a file `Test_Stack.cpp` that can be used to test our function. The main file includes `stack.h` and that file includes `Stack.tc`. That one is the template class using a vector.

...and Stack.tc:

```
/*<snippet id="" omit="false">*/
#ifndef STACK_TC_
```

```
#define STACK_TC_

/** Construct an initially empty stack.  */
template<typename Item_Type>
  stack<Item_Type>::stack() { }

/** Pushes an item onto the top of the stack.
    @param item The item to be inserted
*/
template<typename Item_Type>
  void stack<Item_Type>::push(const Item_Type& item) {
    container.push_back(item);
  }

/** Returns a reference to the object at the top of the stack
    without removing it.
    @return A reference to the object at the top of the stack
*/
template<typename Item_Type>
  Item_Type& stack<Item_Type>::top() {
    return container.back();
  }

/** Returns a const reference to the object at the
    top of the stack without removing it.
    @return A const reference to the object at the top of the stack
*/
template<typename Item_Type>
  const Item_Type& stack<Item_Type>::top() const {
    return container.back();
  }

/** Removes the top item from the stack.
 */
template<typename Item_Type>
  void stack<Item_Type>::pop() {
    container.pop_back();
  }

/** Determines whether the stack is empty.  */
template<typename Item_Type>
  bool stack<Item_Type>::empty() const {
    return container.empty();
  }

/** Returns the number of items in the stack.  */
template<typename Item_Type>
```

```
  size_t stack<Item_Type>::size() const {
    return container.size();
  }




#endif
/*</snippet>*/
```

We have to modify `stack.h` so it uses the code in listing 5.7. This is in file `Linked_Stack.tc`. The linked list stack uses `Node.h`. Notice how in `stack.h` there is a conditional compilation statement `#ifdef USELIST`? This allows us to decide which version to use by adding a compiler option.

```
g++ -DUSELIST Unit_Test_Stack.cpp
```

This option will define USELIST so that the code will include `Linked_Stack.tc`.

But first we will modify the driver to use Bruce Eckel's unit test framework. Because of the way that `stack.h` is included, this is not the `std::stack` but Koffman's version. We want to test each of the member functions in the stack class. (stack, push, pop, top, empty, and size).

Unit_Test_Stack.cpp

```
#include <iostream>
#include "stack.h"
#include "test.h"
using std::cout;
using std::endl;
using KW::stack;
class TestStack : public TestSuite::Test
{
public:
  void run()
 {
   stack<int> the_stack;
   test_(the_stack.empty() == true);
   for (int i = 0; i < 10; i++)  the_stack.push(i);
   test_(the_stack.empty() == false);
   test_(the_stack.size() == 10);
   the_stack.pop();
   test_(the_stack.size() == 9);
   test_(the_stack.top() == 8);
 }
};
int main()
{
  TestStack t;
  t.run();
  t.report();
}
```

Test this using the vector stack and the linked list stack. Notice that the testcases for `size` fail since we have not yet implemented that function!

To add our new function to an existing fragment in ProTex, just create a fragment with the same name. I am using `#pragma once` instead of the method shown in textbook to prevent multiple includes because this way there is no `#endif` to interfere with our adding more code to the end of the file. Here is the original:

Linked_Stack.tc

```
#pragma once
template<typename Item_Type>
  stack<Item_Type>::stack() : top_of_stack(NULL) {}

template<typename Item_Type>
void stack<Item_Type>::push(const Item_Type& item) {
    top_of_stack = new Node(item, top_of_stack);
}

template<typename Item_Type>
Item_Type& stack<Item_Type>::top() {
    return top_of_stack->data;
}

template<typename Item_Type>
const Item_Type& stack<Item_Type>::top() const {
    return top_of_stack->data;
}

template<typename Item_Type>
  void stack<Item_Type>::pop() {
    Node* old_top = top_of_stack;
    top_of_stack = top_of_stack->next;
    delete old_top;
  }

template<typename Item_Type>
bool stack<Item_Type>::empty() const {
    return top_of_stack == NULL;
}
```

And here is our new function added in (just missing a few details!):

Linked_Stack.tc +

```
template<typename Item_Type>
  size_t stack<Item_Type>::size() const {
}
```

4 5.4 Postfix Expressions - unit test expression code

Using Bruce Eckel's unit test framework, test each of the ideas mentioned in these paragraphs.

page 338 Write a unit test driver to verify that the Postfix_Evaluator code works correctly.

page 346 Write a unit test driver to verify that the Infix_To_Postfix code works correctly.

page 350 Write a unit test driver to verify that the Infix_To_Postfix with parenthesis code works correctly.