CS1321 Homework 7*

Jeffrey D. Oldham

2000 Mar 30

Due Thursday, 2000 Apr 06, at the beginning of class.

Contents

1 Revisions

2000Apr05: In the section describing how to evaluate a reverse Polish expression, we used \$. This is for expositional purposes only. In your code, the \$ need not be placed on the stack and a \$ will not be at the end of the expression.

2000Apr01: Specified that the user interface for the tautology checker may not be changed.

2 Reading

The textbook's chapter 6 covers defining templatized classes. Chapter 7 discusses stacks. Be sure to read Section 7.4, concentrating on reverse Polish notation expressions.

prototype	example use	explanation
`value_type`	`stack<string>::value_type x;`	type of items on stack
`size_type`	`stack<bool>::size_type n;`	type for size of stack (number of elements)
`stack<T>(void)`	`stack<int> s;`	create a stack of elements with type `T` but no items
`bool empty(void) const;`	`bool b = s.empty();`	returns true if stack has no elements, false otherwise
`size_type size(void) const;`	`stack<int>::size_type sz = s.size();`	returns number of elements currently in stack
`void push(const value_type & x);`	`s.push(x);`	adds `x` to stack
`void pop(void);`	`s.pop();`	pops (removes) top element of stack. Nothing is returned. It is the user's responsibility to ensure the stack is not empty before calling.
`value_type top(void) const;`	`int x = s.top();`	returns top element of stack without changing the stack. It is the user's responsibility to ensure the stack is not empty.

Table 1: `stack` Member Functions and Types

3 Debugger Tip

The `gdb` debugger permits running programs in "slow motion," permitting examination of how a program runs. For information how to use `gdb`, see my short gdb notes or the notes at Stanford.

Here is a quick and dirty introduction to `gdb`.

1. Compile your program using the `-g` compiler flag, e.g.,

   ```
   g++ -g -Wall -pedantic foo.cc -o foo
   ```

   This causes the compiler to write information used by the debugger.

2. Run `gdb` inside `xemacs` or `emacs` by typing "Alt-x gdb" or "ESC x gdb". Type the executable's name.

3. In the `gdb` window, type "run".

4. To obtain help, type "help".

4 Problem Statement

You and possibly a CS1321 classmate are to implement a stack data structure capable of holding any element type. Then you are to write code using the stack to evaluate a Boolean expression written in reverse Polish notation. Combining this with distributed source code yields a program that checks Boolean expressions for tautologies.

4.1 Stacks

A *stack* is a last-in/first-out data structure with objects arranged in linear order. That is, it permits easy access only from one end. Entries can be added or removed only at the rightmost end. For example, the STL `stack` class implements a stack.

Your implementation should support all the operations listed in Table 1. These operations are similar but not identical to those provided by the STL `stack` class.

4.1.1 Stack Implementation

You can choose any implementation strategy you like for your stack class *except* that you may not use the STL `stack` class or any implementation similar to the source code in the textbook. It should be possible to use your templatized class to create and manipulate stacks of `int`s, `double`s, `string`s, `bool`s, etc., with any number of elements.

Your implementation should correctly use dynamic memory (but why implement it using dynamic memory yourself?).

4.1.2 Stack Implementation Tips

1. First implement a stack with a `typedef` statement declaring the type of its elements:

   `typedef string value_type;`

   Then templatize the class and test using a different type. Using a different type will check that all occurrences of `value_type` were actually found.

2. The `typename` C++ keyword provides alternative syntax for defining templates. In the textbook and in class, template parameters are declared using syntax like `<class T>`. An alternative syntax is `<typename T>`.

   The `typename` keyword is also useful whenever the compiler cannot determine that an expression is a type. For example, it is sometimes necessary to write

   `typedef typename stack<T>::size_type foo;`

   for reasons only apparent to people who have compilers inside their heads. Heuristic: If the compiler becomes terribly confused about a type and the type contains a template parameter, try adding the keyword `typename` before the expression.

5 Boolean Expressions and Reverse Polish Notation

A *Boolean expression* consists of variables, `true`, and `false` connected together by Boolean operators `&&`, `||`, `=>`, `!`, and `==` and possibly parentheses. For example,

(x && y) || (!x && !y)

and

!p || true

are Boolean expressions. Using *infix notation*, where the Boolean operators appear between their operands, can require using parentheses. Instead, we will use *reverse Polish notation*. Using this notation, the previous expressions are written as

x y && x ! y ! && ||

and

p ! true ||

Reverse Polish notation first lists the two operands (using reverse Polish notation) and then the operator. For example: In the second example, the first operand is !p, the operator is `||`, and the second operand is `true`. The reverse Polish notation for the first operand is p `!`. Listing the two operands and then the operator yields the expression.

5.1 Well-Formed Expressions

Intuitively, a well-formed expression has the correct number of operands and operators arranged in the correct order. It is defined recursively:

> A *well-formed expression* is either `true`, `false`, a variable, or an expression p q `&&`, p q `||`, p q `=>`, p q `==`, or p `!`, where p and q are well-formed expressions.

5.2 Truth Tables

To evaluate Boolean expressions, we need to be able to evaluate the simplest Boolean expressions.

o_1 o_2 `&&` is true if and only if both o_1 and o_2 are true.

o_1 o_2 `||` is false if and only if both o_1 and o_2 are false.

o_1 `!` is the "opposite" of o_1.

o_1 o_2 `=>` reads as "if o_1, then o_2." It is true if o_1 is false or o_2 is true. It is false if and only if o_1 is true and o_2 is false.

o_1 o_2 `==` is true if o_1 and o_2 have the same truth value.

Truth tables with the first operand on the left side and the second operand on the right side appear in Figure 1.

`&&`	true	false
true	true	false
false	false	false

`||`	true	false
true	true	true
false	true	false

`!`	
true	false
false	true

`=>`	true	false
true	true	false
false	true	true

`==`	true	false
true	true	false
false	false	true

Figure 1: Truth Tables for Boolean Expressions

step	stack	expression
1	$	true false `&&` true `!` false `!` `&&` `||` $
2	$ true	false `&&` true `!` false `!` `&&` `||` $
3	$ true false	`&&` true `!` false `!` `&&` `||` $
4	$ false	true `!` false `!` `&&` `||` $
5	$ false true	`!` false `!` `&&` `||` $
6	$ false false	false `!` `&&` `||` $
7	$ false false false	`!` `&&` `||` $
8	$ false false true	`&&` `||` $
9	$ false false	`||` $
10	$ false	$

Figure 2: Example Evaluation of a Reverse Polish Expression

5.3 Determining A Boolean Expression's Value

Evaluating a reverse Polish notation is easy using a stack. See Figure 2. Initially, the stack is empty; for expositional purposes, we use $ to denote the bottom of the stack so we can tell it is empty. Initially, we start with the entire expression; we mark its end using a $ for expositional purposes only. The rules are:

1. If the next token is `true` or `false`, move it to the stack.
2. If the next token is a binary operator, i.e., one requiring two operands, remove the two operands from the stack, replacing them with the result of applying the operator to them, and remove the operator from the expression. If less than two operands are present on the stack, the Boolean expression is not well-formed. I.e., it has incorrect syntax.
3. If the next token is a unary operator, i.e., one requiring one operand, remove the operand from the stack, replacing it with the result of applying the operator to it, and remove the operator from the expression. If the stack is empty, the Boolean expression is not well-formed. I.e., it has incorrect syntax.
4. If the next token is $, stop. If the stack has one value, it is the expression's value. Otherwise, the expression is not well-formed.

For example, the first two steps move Booleans from the expression to the stack. In the third step, the `&&` operator beginning the expression is removed, the top two Boolean expressions are popped off the stack, and the result is pushed on the stack. In step 10, the entire expression has been processed. Since there is one Boolean on the stack, it is the value of the expression and the expression was well-formed.

5.4 Tautologies

In addition to Boolean expressions involving only `true`, `false`, and the operators described earlier, we can write Boolean expressions involving variables. Such an expression has a value for any assignment of Boolean values to its variables. For example, consider the expression p q `||`. There are 2^2 ways of assigning values to its two variables since each variable can be either `true` or `false`. For each way of assigning values to p and q, we can then evaluate the resulting expression. The result is `false` if both p and q are `false` and `true` for the other three choices.

A *tautology* is a Boolean expression that evaluates to `true` for all possible ways of assigning values to its variables. For example,

`true`

is a tautology, as are

x x `==`

and

x `!` x `||`

and

x x `==` y y `==` `&&`

since all evaluate to `true` for any way of assigning values to their variables. However,

x

and

x y `=>`

are not tautologies, because there is some way of assigning values to their variables that makes them evaluate to `false`.

Given a Boolean expression with N variables, one way of determining whether it is a tautology is to evaluate the 2^N possible expressions resulting from assigning different combinations of Boolean values to the N variables. If all of them evaluate to `true`, the original expression is a tautology; otherwise it is not.

5.5 Programming an Evaluator

Write a function `evaluate` evaluating a Boolean expression without any variables. Given an expression in reverse Polish notation represented as `vector` of `strings`, it should return a pair of Booleans, the first indicating whether the expression was well-formed and, if well-formed, the second indicating the expression's value.

The provided code reads a Boolean expression with variables from the standard input and cycles through all possible variable assignments, invoking `evaluate` to determine the expression's value. If the expression is true for all assignments, the program indicates is a tautology. Otherwise, the program indicates it is not a tautology or is not well-formed.

The user-provided expression must be in reverse Polish notation with all variables, operators, and keywords separated by whitespace. Any whitespace-delimited set of characters that is not an operator, `true`, or `false` is a variable.

6 What Files Do I Need?

Write a templatized stack class. We have provided a minimal stack.h and test file. Add an `evaluate` function and any helper functions to the tautology checker. A prototype for `evaluate` is already included.

7 Rules

7.1 Programming Rules

As for previous homeworks, working with other people during your planning phase is encouraged. For this homework, you are permitted to write the code, i.e., program, with one other person in CS1321. To learn the material, both of you should be actively involved in the programming. Failure to do so will almost certainly hurt your comprehension of the material in the rest of the course.

7.2 Submission Rules

Each one- or two-person team of programmers should submit only its completed implementation, consisting of the files `stack.h` and `tautology-checker.cc`. You do not need to send any other files. Please send only text documents, do *not* send Microsoft Word documents, PDF documents, HTML documents, etc. **Please include both your names and email addresses at the top of your program.**

We will test your stack implementation using our own programs and our own choice of stack elements; we will also test your completed `tautology-checker` program. Please be sure your code compiles without warning when using `g++ -Wall -pedantic`. It should not be necessary to change the given tautology checker code, but, if you do, changing the user-interface will be considered incorrect.

See the submission details for information how to email the programs. Note that the CS computer configuration was recently changed. If you want to receive an automated reply acknowledging your submission, please read the submission WWW page. If a team of two are in different sections, submit exactly once to one of the two permissible email addresses.

8 Running Time of the Tautology Checker

Given a Boolean expression with v variables and n operators, our tautology checker requires $O(2^v n)$ time. While exponential running times are acceptable for small values of v, they quickly become infeasible. (This Perl program generates a tautology with the number of variables specified as its command-line argument. (After downloading it, make the file executable using `chmod +x generate-tautology.pl`.) Try using it to generate input for your tautology checker. To time the tautology checker program, use timer.h.)

Can we find a faster algorithm? No one has yet been successful. There is a family of *NP-complete problems* all of which are currently thought difficult to solve. We can prove that, if any of these problems could be solved in polynomial time, i.e., $O(n^k)$, for some fixed k, then all these problems can be solved in polynomial time. If we could prove one problem required more than polynomial time, all of them would. Satisfiability, i.e., "Is there an assignment making the Boolean expression true?," is the most famous NP-complete problem. The tautology problem is at least as hard or harder than satisfiability so do not be frustrated by not finding a faster algorithm. For more information, read *Foundations of Computer Science*, by Alfred V. Aho and Jeffrey D. Ullman, ISBN 0-7176-8233-2, p. 649.