CSCI 1321 (Principles of Algorithm Design II), Spring 2001

Homework 7[1]

Assigned: April 6, 2001.

Due: April 14, 2001, at 5pm.

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Reading **1**

2 Overview **2**

3 Stacks **2**
- 3.1 Introduction . . . 2
- 3.2 Stack implementation . . . 2
- 3.3 Stack implementation tips . . . 3

4 Boolean expressions **3**
- 4.1 Boolean expressions and reverse Polish notation . . . 3
- 4.2 Well-formed expressions . . . 4
- 4.3 Truth tables . . . 4
- 4.4 Determining a Boolean expression's value . . . 5
- 4.5 Tautologies . . . 5
- 4.6 Programming an evaluator . . . 6

5 What files do I need? **7**

6 What to turn in **7**

7 Hints, tips, etc. **8**
- 7.1 Running time of the tautology checker . . . 8
- 7.2 Program debugging . . . 8

1 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.

[1] © 2001 Jeffrey D. Oldham (oldham@cs.stanford.edu) and Berna L. Massingill (bmassing@cs.trinity.edu). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

2 Overview

You 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 tautology-checker.cpp[2] yields a program that checks Boolean expressions for tautologies.

3 Stacks

3.1 Introduction

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[3] class implements a stack.

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

Function prototype	Example use	Explanation
`value_type`	`value_type x;`	type of items on stack.
`size_type`	`size_type n;`	type for size of stack (number of elements).
`stack<T>(void)`	`stack<int> s;`	creates 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 this function.
`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 before calling this function.

3.2 Stack implementation

You can choose any implementation strategy you like for your stack class *except* that you may not use the STL `stack` class. 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

[2] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/tautology-checker.cpp`

[3] `http://www.sgi.com/tech/stl/stack.html`

implementation should correctly use dynamic memory (i.e., deep rather than shallow copies, no memory leaks, etc.). Observe, however, that you may be able to achieve this goal with very little effort, if you implement your class using a class that already uses dynamic memory correctly (as we did when we defined a double-ended queue class deque.h[4] using our doubly-linked-list (`dll`) class). You may similarly use any class we have defined in lecture, or any STL class (except for `stack`).

3.3 Stack implementation tips

- You may want to begin by implementing 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.

- The `typename` C++ keyword provides alternative syntax for defining templates. In the textbook, template parameters are declared using syntax such as `<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, as in the following example:

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

  Exactly when and why this is necessary is apparently only obvious 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.

4 Boolean expressions

4.1 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 ||
```

[4] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/SamplePrograms/deque.h`

Reverse Polish notation first lists the two operands (using reverse Polish notation, if they are expressions) 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.

4.2 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.

4.3 Truth tables

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

- 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 (true if o_1 is false, false if o_1 is true).
- 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.

We could also express these rules in the form of truth tables as follows:

&&	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

4.4 Determining a Boolean expression's value

Evaluating an expression in reverse Polish notation is easy using a stack, as in the following example.

Step	Stack	Expression left to scan
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	$

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 $. The rules are:

- If the next token is `true` or `false`, push it onto the stack.
- If the next token is a binary operator `op`, i.e., one requiring two operands, remove the top two elements of the stack (saving the first as `operand2` and the second as `operand1`), apply operator `op` to `operand1` and `operand2`, and push the result back onto the stack. If it is impossible to remove two elements from the stack (i.e., it has fewer than two elements), the Boolean expression is not well-formed (i.e., it has incorrect syntax).
- If the next token is a unary operator `op`, i.e., one requiring one operand, remove the top element of the stack (saving it as `operand1`), apply operator `op` to `operand1`, and push the result back onto the stack. If it is impossible to remove an element from the stack (i.e., it is empty), the Boolean expression is not well-formed (i.e., it has incorrect syntax).
- 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.

4.5 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.

4.6 Programming an evaluator

In this part of the assignment, you are to add code to program tautology-checker.cpp[5].

Specifically, you are to write a function `evaluate()` evaluating a Boolean expression without any variables. The function is to take as input an expression in reverse Polish notation, represented as a `vector<string>`; it is to 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 that it is a tautology. Otherwise, the program indicates that 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 sequence of characters other than an operator, `true`, or `false` is considered to be a variable. Here are some examples of possible input expressions, each involving two variables:

[5] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/tautology-checker.cpp`

```
x y ! ||
hello goodbye &&
hello goodbye
hello goodbye && &&
```

Observe that the last two expressions are not well-formed. This should be detected by your `evaluate()` function.

Notice that although input to the program can include variables, input to your `evaluate()` function will consist of `true`, `false`, and operators only. The end of the expression is indicated by the end of the vector; that is, there is no explicit marker $ as there was in the example shown earlier.

5 What files do I need?

For the first part of the assignment (writing a templatized stack class), you may start from scratch, or you may make use of the following files:

- stack.h[6] provides a very minimal class definition.
- test-stack.cpp[7] provides a very minimal test program.

For the second part of the assignment (completing the tautology-checker program), you will need the following file:

- tautology-checker.cpp[8].

Add to this file an `evaluate()` function and any needed helper functions. A prototype for `evaluate()` is already included. You should not need to make any changes in this program other than adding code for the `evaluate()` function and possibly some helper functions.

6 What to turn in

Submit the following two source-code files:

- Your implementation of the stack class (`stack.h`).
- Your revised/completed version of the tautology-checker program (`tautology-checker.cpp`).

Submit these files as described in the Guidelines for Programming Assignments[9]. For this assignment use a subject line of "cs1321 hw 7".

You do not need to submit a test program for the stack class.

[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/stack.h`

[7] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/test-stack.cpp`

[8] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/tautology-checker.cpp`

[9] `http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Notes/pgmguidelines/index.html`

7 Hints, tips, etc.

7.1 Running time of the tautology checker

Given a Boolean expression with v variables and n operators, our tautology checker requires time roughly proportional to $2^v n$ time. While exponential running times are acceptable for small values of v, they quickly become infeasible. If you want to see this in action, you can use Perl program generate-tautology.pl[10] to generate input for the tautology checker. It takes one command-line argument specifying the number of variables. (To use this program, save it into a file and make the file executable with the command `chmod +x generate-tautology.pl`.) To time the tautology checker program, you can use the `timer()` function in timer.h[11].

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 to be difficult to solve. We can prove that if any of these problems can be solved in polynomial time, i.e., in time roughly proportional to n^k for some fixed k, then all these problems can be solved in polynomial time; conversely, if one of these problems can be proved to require more than polynomial time, then they all do. Satisfiability, i.e., answering the question "is there an assignment making the Boolean expression true?," is the most famous NP-complete problem. The tautology problem is at least as hard as, or harder than, satisfiability. so do not be frustrated by not finding a faster algorithm. (You would become very famous among computer scientists if you found one!) For more information, read *Foundations of Computer Science*, by Alfred V. Aho and Jeffrey D. Ullman, ISBN 0-7176-8233-2, p. 649.

7.2 Program debugging

The `gdb` debugger allows you to run your program in stop-motion form, i.e., to step through it a line at a time, examining variables as you go. This section attempts to present just enough information about `gdb` to get you started; for more information, see J. Oldham's short introduction[12], or the complete on-line manual[13].

To use `gdb`, proceed as follows.

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. Start `gdb` by typing

   ```
   gdb foo
   ```

   (Replace `foo` with the name of your executable, e.g., `a.out`.)

3. Set up to step through your program by typing the following `gdb` commands:

   ```
   break main
   run
   ```

[10] http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/generate-tautology.pl

[11] http://www.cs.trinity.edu/~bmassing/CS1321_2001spring/Homeworks/HW07/Problems/timer.h

[12] http://www.cs.trinity.edu/~bmassing/Misc/jdo-1321/lectures/gdb/

[13] http://www.gnu.org/manual/gdb-4.17/gdb.html

If your program needs command-line arguments, include them in the `run` command, e.g.,

```
run anArgument anotherArgument
```

4. Use the following commands to step through your program and examine variables:

   - `n` or `next` to execute the next line of the program, including all function calls.
   - `s` or `step` to execute the next line of the program, "stepping into" any called functions. That is, if the next two lines are

     ```
     x = foo(10);
     cout << x << endl;
     ```

     `n` will take you to the line outputting the value of `x`, while `s` will take you to the first line of function `foo`.
   - `l` or `list` to list surrounding lines of source code.
   - `p` or `print` to display the value of a variable. For example, if you have the following declarations:

     ```
     int x;
     double y[10];
     pair<char,char> z;
     ```

     then the following commands should all work:

     ```
     p x
     p y[5]
     p z.first
     ```
   - `h` or `help`, optionally followed by the name of a `gdb` command, to get help.

   Just pressing return repeats the most recent command again.

5. Exit `gdb` by typing `q` or `quit`.

`gdb` also runs very nicely under `emacs` and `xemacs`; the main editor window is split into two windows, one for `gdb` commands and output and the other showing source code (with an arrow indicating the next line to execute). To try this out, start `emacs` or `xemacs` and type `M-x gdb`. (The `M-x` is "meta-x", probably either Alt-x or ESC-x on your keyboard.) You will be prompted for the name of the program; type in the name of your executable (e.g., `a.out` or `foo`).

You might also want to try `xxgdb`, which provides a graphical interface for `gdb`. Start it up by typing `xxgdb foo`, where `foo` is the name of your executable.