the Stack

1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class

MCS 360 Lecture 13
Introduction to Data Structures
Jan Verschelde, 12 February 2020
The Stack

1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class
A stack is a LIFO (Last In First Out) sequence: we can only get (or pop) the element on top.

Example: a pile of books, stacked on top of each other.

Despite this restriction, many applications:
- parsing expressions,
- store information about function calls.

In the STL, a stack is an adapter: either a vector or a list implements a stack.
UML class diagram

```
stack

- data

+ empty()
+ top()
+ pop()
+ push()
```
abstract <typename T> stack;
/* a stack is a sequence of elements, stored as Last In First Out (LIFO) */

abstract bool empty ( stack s );
p postcodeition: empty(s)
  == true if s is empty,
  == false if s is not empty;

abstract T top ( stack s );
pre condition: not empty(s);
p postcodeition: top(s) is the top element of s;
abstract void push ( stack s, T e );
postcondition: push(s,e); top(s) == e;

abstract void pop ( stack s );
precondition: not empty(s);
postcondition: top(s) is removed from s;
the Stack

1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class
using the STL stack – counting down

Our first program with an STL stack does the following:

1. push given numbers to the stack, until zero is entered;
2. pop stored numbers from the stack, until empty.

$ use_stl_stack
give an integer (0 to stop) : 1
give an integer (0 to stop) : 2
give an integer (0 to stop) : 3
give an integer (0 to stop) : 0
popped 3
popped 2
popped 1
$

A stack is a natural data structure to reverse the order in any sequence.
Exercise 1: (recall exercise 2 of L-1)

Write a C++ program which prompts the user for a positive integer number.
The program writes the input number and prints the bits in the binary decomposition of the number, in the correct order, printing the most significant bit first.

A session with the program could go as follows:

Give a number : 360
The bits in 360 : 1 0 1 1 0 1 0 0 0 0.
using the STL stack

#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> s;
    int e;

    do
    {
        cout << "give an integer (0 to stop) : ";
        cin >> e;
        if(e <= 0) break;
        s.push(e);
    }
    while(true);
}
while(!s.empty())
{
    int e = s.top();
    s.pop();
    cout << "popped " << e << endl;
}

Members of stack<T> class:

- T top() const: returns the top of the stack,
- void pop(): removes the top element.
The Stack

1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class
Given an expression \((w*(x+y) / z - (p/(9 -8)))\), test if every closing bracket \()\) matches an opener \(\(\). Simple counting algorithm:

- +1 if encounter opener \(\(, and
- -1 if encounter closer \).  

Expression is balanced if final count equals zero.

Harder if different type of brackets, braces, and parentheses can be used, e.g.: \((w*[x+y] / z - [p/{9 -8}])\)
running the program

$ match_brackets
give an expression : (w*[x+y] / z - [p/{9 - 8}])
checking "(w*[x+y] / z - [p/{9 - 8}])" ...
pushed ( 
pushed [
] matches [ at 7
popped [
pushed [
pushed {
} matches { at 24
popped {
} matches [ at 25
popped [
) matches ( at 26
popped ( 
parenthesis in "(w*[x+y] / z - [p/{9 - 8}])") \ 
are balanced
$

Introduction to Data Structures (MCS 360)
the Stack

1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class
an algorithm which uses a stack

A stack stores all opening brackets:
- we push every opening bracket,
- for every matching closing bracket, we pop.

Outline of the algorithm:

For every character \( c \) in a given expression:
- if \( c \) is ‘(’, ‘{’, or ‘[’ then
  - push \( c \) to a stack
- else if \( c \) is ‘)’, ‘}’, or ‘]’ then
  - if top of stack matches \( c \) then
    - pop the stack
  - else
    - break out of loop: parenthesis unbalanced.

Careful: stack may be empty.
include <iostream>
#include <string>
#include <stack>

using namespace std;

int main()
{
    string expression;

    cout << "give an expression : ";
    getline(cin,expression,'
');

    stack<char> brackets; // stack of brackets
    const string opening_brackets = "("[ ";
    const string closing_brackets = ")"]";

    Positions of corresponding opening/closing brackets match.
For any string $s$ and character $c$:

- $s$.find($c$) either
  - returns `string::npos` if $c$ does not occur in $s$,
  - or
  - returns the first index $k$ for which $s[k] == c$.

We use `find with` $c == expression[i]$ on $s == opening_brackets$
or $s == closing_brackets$. 
pushing and popping brackets

bool balanced = true;

for(int i=0; i<expression.size(); i++)
    if(opening_brackets.find(expression[i])
        != string::npos) // push opening bracket
        brackets.push(expression[i]);
    else if(closing_brackets.find(expression[i])
        != string::npos)
    { // Does top of brackets match closing bracket?
        int k = closing_brackets.find(expression[i]);
        if(brackets.empty())
        {
            cout << "no matching opening bracket for "
                 << expression[i] << " at " << i << endl;
            balanced = false; break;
        }
    }
Does top of bracket match closing bracket?

As we have

- `closing_brackets.find(expression[i]) != string::npos`  
- `int k = closing_brackets.find(expression[i]);`  
- `!brackets.empty()`

we can examining the top of the stack of brackets:

```c++
char c = brackets.top();
if(c != opening_brackets[k]) // match?
    { // if not at same position, then no match
        cout << "unbalanced parenthesis : "
        << c << " != " << expression[i]
        << " at " << i << endl;
        balanced = false; break;
    }
```
we can pop matching opening bracket

else
{
    cout << expression[i] << " matches " << c << " at " << i << endl;
    brackets.pop();
    cout << "popped " << c << endl;
}
} // end of handling closing bracket

cout << "parenthesis in "" << expression << "\"";
if(!balanced)
    cout << " are not balanced" << endl;
else if(brackets.empty())
    cout << " are balanced" << endl;
else
    cout << " are not balanced, missing closing bracket" << endl;
Exercise 2: balancing brackets in C++ program

Write a program that prompts the user for the name of a file which contains C++ source code.

The program verifies whether every closing bracket in the code matches the corresponding opening bracket: every {, (, and [ is closed respectively by }, ), and ].

In your program, write a short message for every push and pop that happens during the execution of the program.
the Stack

1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class
adapting STL vectors

A stack is said to be an *adapter* class: we adapt a sequential container, get a stack implementation providing another interface to vector or list.

A dictionary between STL stack and vector, for any item \( t \) of type \( T \):

<table>
<thead>
<tr>
<th>stack( &lt;T&gt; \ s )</th>
<th>vector( &lt;T&gt; \ v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\s.push(t) \</td>
<td>\v.push_back(t) \</td>
</tr>
<tr>
<td>if(!s.empty())</td>
<td>if(!v.empty())</td>
</tr>
<tr>
<td>\t = s.top() \</td>
<td>\t = v[v.size()-1]</td>
</tr>
<tr>
<td>\t = s.top() \</td>
<td>\t = v.back()</td>
</tr>
<tr>
<td>\s.pop() \</td>
<td>\v.pop_back()</td>
</tr>
</tbody>
</table>
STL vector as stack

#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> s;
    int e;

    do
    {
        cout << "give an integer (0 to stop) : ";
        cin >> e;
        if(e <= 0) break;
        s.push_back(e);
    }
    while(true);
while(!s.empty())
{
    int e = s[s.size()-1];
    cout << "popped " << e
    << " = " << s.back() << endl;
    s.pop_back();
}

A stack is a natural data structure to reverse the order in any sequence.

Application: check if word is a palindrome.
A palindrome can be read forward and backward, some examples: dad, testset, racecar.
1. The Stack Abstract Data Type
   - LIFO, UML class diagram, stack ADT
   - using the STL stack

2. An Application: Test Expressions
   - are parentheses balanced?
   - an algorithm which uses a stack

3. Stack Implementations
   - adapting the STL vector class
   - adapting the STL list class
adapting STL lists

As an alternative to adapting the STL vector class, we can implement a stack adapting the STL list class. A dictionary between STL stack and list, for any item $t$ of type $T$:

<table>
<thead>
<tr>
<th>stack$\langle T \rangle$ $s$</th>
<th>List$\langle T \rangle$ $L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$.push($t$)</td>
<td>$L$.push_back($t$)</td>
</tr>
<tr>
<td>$s$.pop()</td>
<td>$L$.pop_back()</td>
</tr>
<tr>
<td>if(!$s$.empty())</td>
<td>if(!$L$.empty())</td>
</tr>
<tr>
<td>$t$ = $s$.top()</td>
<td>$t$ = $L$.back()</td>
</tr>
</tbody>
</table>

---

Introduction to Data Structures (MCS 360)
STL list as stack

```cpp
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> s;
    int e;

    do
    {
        cout << "give an integer (0 to stop) : ";
        cin >> e;
        if(e <= 0) break;
        s.push_back(e);
    }
    while(true);
}```
top and pop

```cpp
while(!s.empty())
{
    int e = s.back();
    s.pop_back();
    cout << "popped " << e << endl;
}
```

Observe that the uniform naming of methods in STL leads to another type of generic programming: in the description of the algorithm, we may declare `s` as `vector<T> s`, or as `list<T> s`. 
Summary + Additional Exercises

Started Chapter 5 on *Stacks*. Although simpler than vectors or lists, some algorithms reduce to mere loops with a stack.

**Additional Exercises:**

3. Write code using a stack to test if a string, given by the user, is a palindrome.

4. The phrase "murder for a jar of red rum" is a palindrome. Adjust the code of the first exercise to ignore spaces.

5. Adjust the test for balanced parentheses to match single (left ‘ and right ’) and double quotes ".

6. Use a stack to compute the value of a number given as string in some given basis ($< 10$). For example: "537" in octal (base 8) evaluates to $7 + 3 \times 8 + 5 \times 8^2$. 

---

*Introduction to Data Structures (MCS 360)*

the Stack

L-13  12 February 2020  31/31