Stacks

Learning Objectives

- Follow and explain stack based algorithms using the usual computer science terminology of push, pop and top
- Implement a stack class of your own using either an array or a linked list data structure

Stacks and the STL stack

❖ Definition

- A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)

❖ LIFO

- A stack is a Last-In/First-Out data structure. Entries are taken out of the stack in the reverse order of their insertion

The Standard Library Stack Class

- The C++ Standard Template Library (STL) has a stack class
- Stack is specified as a template class
- The most important member functions are:
  - **Push:** to add an entry at the top of the stack
  - **Pop:** to remove the top entry
  - **Top:** to get the item at the top of the stack without removing it
- There are no functions that allow a program to access entries other than the top entry
- **Stack underflow:** If a program attempts to pop an item off an empty stack
  - To help you avoid a stack underflow, the class provides a member function to test whether a stack is empty
- **Stack overflow**: If a program attempts to push an item onto a full stack

The Standard Library Stack Class (cont.)

```
template < class T, class Container = deque<T> >
class stack;
```

- **stacks** are implemented as *containers adaptors*
  - **Containers adaptors** are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements
  - The standard container classes **vector**, **deque** and **list** fulfill these requirements
- How to use
  - #include <stack>
  - stack<int> s1;

Programming Example: Balanced Parentheses

```
bool is_balanced(const string& expression){
  const char LEFT_PARENTHESIS = '(';
  const char RIGHT_PARENTHESIS = ')';
  stack<char> store;  // Stack to store the left parentheses
  string::size_type i; // An index into the string
  char next;           // The next character from the string
  bool failed = false; // True if a needed parenthesis is not found

  for (i = 0; !failed  &&  (i < expression.length( )); ++i)
   {
     next = expression[i];
     if (next == LEFT_PARENTHESIS)
        store.push(next);
     else if ((next == RIGHT_PARENTHESIS) && (!store.empty()))
        store.pop( ); // Pops the corresponding left parenthesis.
     else if ((next == RIGHT_PARENTHESIS) && (store.empty( )))
        failed = true;
   }
   return (store.empty( ) && !failed);
}
```

IMPLEMENTATIONS OF THE STACK CLASS

Array Implementation of a Stack

- Our stack template class definition uses two private member variables:
  - A partially-filled array, called `data`, that can hold up to `CAPACITY` items
  - A single member variable, `used`, that indicates how much of the partially-filled array is currently being used
    - ✓ `data[0]` is at “the bottom” of the stack
    - ✓ `data[used-1]` is at “the top” of the stack
    - ✓ If the value of `used` is zero, this will indicate an empty stack
- Invariant of the `Stack` Class
  - The number of items in the stack is stored in the member variable `used`
  - The items in the stack are stored in a partially filled array called `data`, with the bottom of the stack at `data[0]`, the next entry at `data[1]`, and so on to the top of the stack at `data[used-1]`

Linked-List Implementation of a Stack

- A stack as a dynamic structure
- Size can grow and shrink during execution
- **The head of the linked list serves as the top of the stack**
- Invariant of the Stack Class (Linked-List Version):
  - The items in the stack are stored in a linked list, with the top of the stack stored at the head node, down to the bottom of the stack at the tail node
  - The member variable `top_ptr` is the head pointer of the linked list of items