Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Stacks and Queues

A stack

- A special kind of list in which elements are only inserted and removed from 1 end. *(imagine maintaining a pile of plates. New plates are added at the top of the pile. When a plate is required, it is removed from the top of the pile)*

  LIFO – Last In First Out

- The terminology commonly used for a stack is that elements are '*pushed*' on and '*popped*' off the stack.
  - The public methods of the stack are `push()` and `pop()`
- In fact the STL has just 5 methods for a stack template class:
  - **`push(val)`** pushes the val onto the top of the stack
  - **`pop()`** removes the val on the top of the stack (without returning it)
  - **`top()`** returns the val on the top of the stack (without removing it)
  - **`size()`** returns the no of elements on the stack
  - **`empty()`** returns true if the stack is empty, false otherwise

Implementing a stack

- We could implement it from scratch – for example using a linked list with just the methods we want
- If we already have a List class written, we can implement the Stack class re-using the List class through ***composition***.
  - The stack would then have a list as a *private attribute*.
  - **`push()`** will call the `insertAtFront()` method of the list
  - **`pop()`** will call the `removeFromFront()` method of the list

  - On our lab sheet we called the list methods `insert()` and `deleteMostRecent()`
  - In the STL they are `push_back` and `pop_back`

3. If we already have a vector class, we can implement the Stack class through composition.
   - The stack would then have a Vector as a *private attribute*.
   - **`push()`** will call the `push_back()` method of the vector
   - **`pop()`** will call the `pop_back()` method of the vector

push_back and pop_back are methods of the STL vector template class.

   – *There is another method to re-use the List or Vector class to implement a stack using 'private inheritance'. We will re-visit after covering inheritance next sem.*

The Stack: Using composition to re-use a general List class

```
class Stack  {
{
public:
   void push(double val)
   { s.insertAtFront(val); }

   bool pop (int &val) const
   {return s.removeFromFront(val); }

   bool isEmpty const;
   { return s.isEmpty(); }

   void printStack() const
   { s.print() }

private:
   List s;
};
```

I have in-lined the methods to show on one overhead - should really be implemented outside the class definition in the usual way.

The Stack: exercise in developing a stack class

- The third alternative, of course, is to implement the stack class ‘from scratch’. It would need to implement the 2 methods `push()` and `pop()`, and to facilitate this it should, as in other examples, maintain as a private attribute a pointer to the top node in the stack. (`head`)

- Exercise: implement a stack of *character* type by each of these methods:
  - Composition, using an STL list (`list<char>`)to hold the data
  - Composition, using an STL vector `(vector<char>)` to hold the data
  - from scratch (assuming no other class is available to re-use)

Should we use a vector or a list to implement a stack?

- In the STL, the stack template class and queue template class are implemented on many different 'containers'
- When we ask for one, we can specify what container we want it to use.
- So what would we be likely to choose for a stack?
- Advantages of a vector for a stack?
  - No pointers to add overhead to the size of the list?
  - Indexed access to the top element is very efficient within the vector class
    - The 'top' item is the one at the end of the vector
  - Pop simply reduces the size of the vector by one
    - very very fast
  - Push always adds to the end of the vector
    - also fast *as long as present capacity is not exceeded*
    - *There is overhead if internally the vector must increase capacity.*
- Advantages of a linked list?
  - No wasted space as in a vector (capacity usually will exceed the size)
    - But we do have the overhead of storing all the pointer links
  - Pop and push can both be from the head of the list, so very fast – just some pointers to manipulate.

A Queue

- a special kind of list in which elements are always:
  - added at one end (the 'tail' of the queue) this is the 'en-que' operation
  - removed from the other end (the 'head' of the queue) this is the 'de-que' operation.

FIFO – First In First Out

- The STL queue template class has the methods
  - **`push()`** to en-que an element
  - **`pop()`** to de-que an element (without returning it)
  - **`front()`** to return the element at the front of the queue (without de-queing it)
  - **`back()`** to return the element at the back of the queue (it wouldn't be a queue if we could remove this one!)

Exercise: implement a queue of *double* type by each of these methods:

- Composition, using an STL list (`list<double>`)to hold the data
- Composition, using an STL vector (`vector<double>`) to hold the data
- from scratch (assuming no other class is available to re-use)

Should we use a vector or a list to implement a queue?

- In the STL, the stack template class and queue template class are implemented on many different 'containers'
- When we ask for one, we can specify what container we want it to use.
- So what would we be likely to choose for a queue?
- Advantages of a vector for a queue?
  - No pointers to add overhead to the size of the list?
  - Push can still add to the end of the vector
    - Fast as long as present capacity is not exceeded
  - But pop will have to remove the top item in the vector.
    - Normally, this means moving everything else in the vector up to fill the space
    - There are clever ways to implement a vector to get around this
  - Indexed access to the first and last element in the queue is very efficient
- Advantages of a linked list?
  - No wasted space as in a vector with capacity exceeding size used.
    - But we do have the overhead of storing all the pointer links
  - push can both be from the head of the list, so very fast – just some pointers to manipulate
  - But pop will have to traverse the whole list to find the last item
    - Only efficient if the list is implement as a doubly-liked list

The **`deque`** is a specialist type of vector in the STL that can grow and shrink at both ends