template<class T>
class stack
{
    public:
    virtual ~stack() {}
    virtual bool empty() const = 0;
    virtual int size() const = 0;
    virtual T& top() = 0;
    virtual void pop() = 0;
    virtual void push(const T& theElement) = 0;
};
Derive From A Linear List Class

- arrayList
- chain
**Derive From ArrayList**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Stack top is either left end or right end of linear list
- `empty()` => `arrayList::empty()`
  - $O(1)$ time
- `size()` => `arrayList::size()`
  - $O(1)$ time
- `top()` => `get(0)` or `get(size() - 1)`
  - $O(1)$ time
Derive From array List

- when top is left end of linear list
  - \texttt{push}(theElement) => \texttt{insert}(0, theElement)
  - \textbf{O(size)} time
  - \texttt{pop()} => \texttt{erase}(0)
  - \textbf{O(size)} time
Derive From `arrayList`

| a b c d e | 0 1 2 3 4 5 6 |

- when top is right end of linear list
  - `push(theElement) => insert(size(), theElement)`
  - $\mathcal{O}(1)$ time
  - `pop() => erase(size()-1)`
  - $\mathcal{O}(1)$ time

- use right end of list as top of stack
Derive From Chain

- stack top is either left end or right end of linear list
- `empty() => chain::empty()`
  - $O(1)$ time
- `size() => chain::size()`
  - $O(1)$ time
Derive From Chain

- when top is left end of linear list
  - top() => get(0)
  - O(1) time
  - push(theElement) => insert(0, theElement)
  - O(1) time
  - pop() => erase(0)
  - O(1) time
Derive From Chain

- when top is right end of linear list
  - \( \text{top()} \rightarrow \text{get(size()) - 1} \)
  - \( O(\text{size}) \) time
  - \( \text{push(theElement)} \rightarrow \text{insert(size(), theElement)} \)
  - \( O(\text{size}) \) time
  - \( \text{pop()} \rightarrow \text{erase(size()-1)} \)
  - \( O(\text{size}) \) time

- use left end of list as top of stack
template<class T>
class derivedArrayStack :
    private arrayList<T>,
    public stack<T>
{
    public:
        // code for stack methods comes here
};
Constructor

derivedArrayStack(int initialCapacity = 10)
    : arrayList<T> (initialCapacity) {}
empty() And size()

bool empty() const
{ return arrayList<T>::empty(); }

int size() const
{ return arrayList<T>::size(); }
T& top()
{
    if (arrayList<T>::empty())
        throw stackEmpty();
    return get(arrayList<T>::size() - 1);
}
```cpp
void push(const T& theElement)
{
    insert(arrayList<T>::size(), theElement);
}
```
void pop()
{
    if (arrayList<T>::empty())
        throw stackEmpty();
    erase(arrayList<T>::size() - 1);
}
Evaluation

• Merits of deriving from `arrayList`
  ▪ Code for derived class is quite simple and easy to develop.
  ▪ Code is expected to require little debugging.
  ▪ Code for other stack implementations such as a linked implementation are easily obtained.
  • Just replace `private arrayList<T> with private chain<T>`
  • For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.
Demerits

- Unnecessary work is done by the code.
  - `top()` verifies that the stack is not empty before `get` is invoked. The index check done by `get` is, therefore, not needed.
  - `insert(size(), theElement)` does an index check and a `copy_backward`. Neither is needed.
  - `pop()` verifies that the stack is not empty before `erase` is invoked. `erase` does an index check and a `copy`. Neither is needed.
  - So the derived code runs slower than necessary.
Evaluation

- Code developed from scratch will run faster but will take more time (cost) to develop.
- Tradeoff between software development cost and performance.
- Tradeoff between time to market and performance.
- Could develop easy code first and later refine it to improve performance.
A Faster pop()

if (arrayList<T>::empty())
    throw stackEmpty();
erase(arrayList<T>::size() - 1);

vs.

try {
erase(arrayList<T>::size() - 1);
catch (illegalIndex
    {throw stackEmpty();}}
Code From Scratch

• Use a 1D array stack whose data type is T.
  ▪ same as using array element in ArrayList

• Use an int variable stackTop.
  ▪ Stack elements are in stack[0:stackTop].
  ▪ Top element is in stack[stackTop].
  ▪ Bottom element is in stack[0].
  ▪ Stack is empty iff stackTop = -1.
  ▪ Number of elements in stack is stackTop + 1.
template class <T>

class arrayStack : public stack<T>
{
    public:
        // public methods come here
    private:
        int stackTop; // current top of stack
        int arrayLength; // stack capacity
        T *stack; // element array
};
Constructor

template<class T>
arrayStack<T>::arrayStack(int initialCapacity)
{// Constructor.
    if (initialCapacity < 1)
        {// code to throw an exception comes here
        }
    arrayLength = initialCapacity;
    stack = new T[arrayLength];
    stackTop = -1;
}
template<class T>
void arrayStack<T>::push(const T& theElement)
{// Add theElement to stack.
    if (stackTop == arrayLength - 1)
        //{ // code to double capacity comes here
        
    // add at stack top
    stack[++stackTop] = theElement;
}
void pop()
{
    if (stackTop == -1)
        throw stackEmpty();
    stack[stackTop--].~T();  // destructor for T
}
Linked Stack From Scratch

- See text.
## Performance

50,000,000 pop, push, and peek operations

<table>
<thead>
<tr>
<th>Class</th>
<th>initial capacity</th>
<th>10</th>
<th>50,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayStack</td>
<td></td>
<td>2.7s</td>
<td>1.5s</td>
</tr>
<tr>
<td>derivedArrayStack</td>
<td></td>
<td>7.5s</td>
<td>6.3s</td>
</tr>
<tr>
<td>STL</td>
<td></td>
<td>5.6s</td>
<td>-</td>
</tr>
<tr>
<td>derivedLinkedStack</td>
<td></td>
<td>41.0s</td>
<td>41.0s</td>
</tr>
<tr>
<td>linkedStack</td>
<td></td>
<td>40.5s</td>
<td>40.5s</td>
</tr>
</tbody>
</table>