Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure.
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

Example: ADT modeling a simple stock trading system
- The data stored are buy/sell orders
- The operations supported are:
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object o): inserts element o
  - pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - top(): returns a reference to the last inserted element without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations **pop** and **top** cannot be performed if the stack is empty
- Attempting the execution of **pop** or **top** on an empty stack throws an EmptyStackException

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
The C++ run-time system keeps track of the chain of active functions with a stack. When a function is called, the run-time system pushes on the stack a frame containing:
- Local variables and return value
- Program counter, keeping track of the statement being executed

When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack.

```cpp
main() {
    int i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);
}

bar(int m) {
    ...
}
```

### Array-based Stack

A simple way of implementing the Stack ADT uses an array. We add elements from left to right, and a variable keeps track of the index of the top element.

**Algorithm size()**
```cpp
return t + 1
```

**Algorithm pop()**
```cpp
if isEmpty() then
    throw EmptyStackException
else
    t ← t - 1
    return S[t+1]
```

**Algorithm push(o)**
```cpp
if t = S.length - 1 then
    throw FullStackException
else
    t ← t + 1
    S[t] ← o
return S
```

The array storing the stack elements may become full. A push operation will then throw a FullStackException. Limitation of the array-based implementation is not intrinsic to the Stack ADT.

**Algorithm spans1(X, n)**
```cpp
Input array X of n integers
Output array S of spans of X
S ← new array of n integers
for i ← 0 to n - 1 do
    s ← 1
    while s ≤ i ∧ X[i] ≤ X[i - s] do
        s ← s + 1
    S[i] ← s
return S
```

Spans have applications to financial analysis:
- E.g., stock at 52-week high.

**Quadratic Algorithm**

Algorithm runs in $O(n^2)$ time.
Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back".
- We scan the array from left to right.
  - Let $i$ be the current index.
  - We pop indices from the stack until we find index $j$ such that $X[j] < X[i]$.
  - We set $S[i] ← i - j$.
  - We push $x$ onto the stack.

Linear Algorithm

- Each index of the array is pushed into the stack exactly one time.
- Each index is popped from the stack at most once.
- The statements in the while-loop are executed at most $n$ times.
- Algorithm $spans2$ runs in $O(n)$ time.

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.
- How large should the new array be?
  - Incremental strategy: increase the size by a constant $c$.
  - Doubling strategy: double the size.

Algorithm $push(o)$

```
if t = S.length - 1 then
    A ← new array of size ...
    for i ← 0 to t do
        A[i] ← S[i]
    t ← t + 1
    S[t] ← o
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ push operations.
- We assume that we start with an empty stack represented by an array of size 1.
- We call amortized time of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$.

Incremental Strategy Analysis

- We replace the array $k = n/c$ times.
- The total time $T(n)$ of a series of $n$ push operations is proportional to
  $$n + c + 2c + 3c + 4c + \ldots + kc = \frac{n + c(k + 1)}{2}.$$
- Since $c$ is a constant, $T(n)$ is $O(n^2)$, i.e., $O(n^2)$.
- The amortized time of a push operation is $O(n)$.

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times.
- The total time $T(n)$ of a series of $n$ push operations is proportional to
  $$n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 2n - 1.$$
- $T(n)$ is $O(n)$.
- The amortized time of a push operation is $O(1)$.
Stack Interface in C++

- Interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Most similar STL construct is vector

```cpp
template<typename Object>
class Stack {
public:
    int size();
    bool isEmpty();
    Object& top();
    void push(Object o);
    Object pop() throw(EmptyStackException);
};
```

Array-based Stack in C++

```cpp
template<typename Object>
class ArrayStack {
private:
    int capacity; // stack capacity
    Object *S; // stack array
    int top; // top of stack

public:
    ArrayStack(int c) {
        capacity = c;
        S = new Object[capacity];
        t = -1;
    }
    bool isEmpty() {
        return (t < 0);
    }
    Object pop() throw(EmptyStackException) {
        if(isEmpty())
            throw EmptyStackException
        return S[t--];
    }
    // … (other functions omitted)
};
```