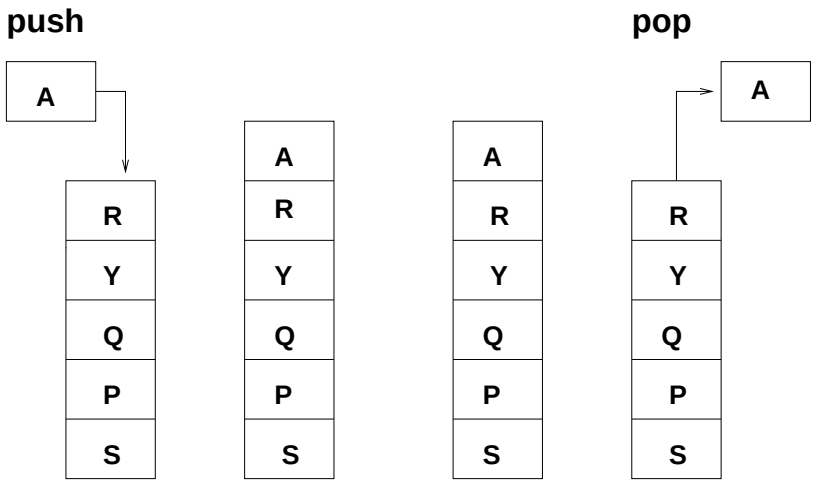
Stack Data Structure

1. A stack is a linear list in which all additions and deletions are restricted to one end, called **top**.
2. If you inserted objects into a stack and then you removed them, the order of the objects is reversed. Numbers inserted as 1, 2, 3, 4, 5 are removed as 5, 4, 3, 2, 1.
3. This reversing attribute is the reason why stacks are called *LIFO* (last-in, first-out) data structure.
   It behaves very much like the common stack of plates or stack of newspapers. The last object added to the stack is placed on the top and is easily accessible, while objects that have been in the stack for a while are more difficult to access.
4. Thus the stack is appropriate if we expect to access *only* the top object, all other objects are inaccessible.

Stack ADT

The three natural operations on a stack are `push` (insert), `pop` (delete), and `top` (view).

1. The function `push(object)` adds an object to the top of the stack.
2. The function `void pop()` removes an object from the top of the stack.
3. The function `top` reads the value on the top of the stack (it does not remove it).
4. If useful, we can combine `top` and `pop` and get the function `object topAndPop()`
   which returns and removes the most recent object from the stack.



Informal Stack Interface

The C++ interface corresponding to our stack ADT. It provides to a user useful information about all the public functions of the class `Stack`.

```
class Stack {
public:
   // return the top object of a non-empty stack
   int top() throw(StackEmptyException);
   // push object onto the stack
   // throwing this exception is not always necessary
   void push(const int elem) throw(StackFullException);
   // returns first object of  a non-empty stack
   int pop() throw(StackEmptyException);
   // checks whether no elements are stored?
   bool isEmpty() const;
};
```

The operations `pop()` and `top()` cannot be performed if stack is empty. The attempt of removing or getting the top object on empty stack throws exception. An exception is also thrown if there is no space to add a new object to a stack.

Array Implementation of Stack

A stack can be implemented using only two data members: `array` and an integer `topOfStack`, the latter is the array index of the top element of the stack. When `topOfStack` is -1, the stack is empty. To push, we increment `topOfStack` and place the new element at the top of the array. Pop is implemented by decrementing `topOfStack`. The exception class `RuntimeException` provides a function `getMessage` The exceptions `StackFullException` and `StackEmptyException` are subclasses of `RuntimeException`. The class is in the file `ArrayStack.h`.

```
class ArrayStack {
 private: // member data
   int  capacity;      // actual length of stack array
   int* array;         // the stack array (a pointer)
   int  topOfStack;    //index of the top of the stack
```

Array Implementation Stack (cont.)

```
public:
  // constructor given max capacity
  ArrayStack() : capacity(0), array(0), topOfStack(-1) {}
  ArrayStack(int cap) : capacity(cap),
      array(new int[cap]), topOfStack(-1) {}
   ArrayStack(const ArrayStack& st); // copy constructor
  // assignment operator
  ArrayStack& operator=(const ArrayStack& st);
  ~ArrayStack() {  delete [] array; }  // destructor
  bool isEmpty() const { return (topOfStack < 0); }
  // return the top object of the stack
  int top() throw(StackEmptyException);
  // push object onto the stack
  void push(const int elem)  throw(StackFullException);
  int pop() throw(StackEmptyException); // pop the stack
  // number of elements in the stack (auxiliary function)
  int size() const {  return (topOfStack + 1); }
}; // end of class ArrayStack
```

The Stack Implementation (cont.)

```
// return the top of the stack
int ArrayStack::top() throw(StackEmptyException)
{   if (isEmpty())
       throw StackEmptyException("Access to empty stack");
    return array[topOfStack];
}
// push object onto the stack
void ArrayStack::push(const int elem)throw(StackFullException)
{   if (size() == capacity)
       throw StackFullException("Stack overflow");
    array[++topOfStack] = elem;
}
```

The Implementation of Copy Constructor

```
// pop the stack
int ArrayStack::pop() throw(StackEmptyException)
{
   if (isEmpty())
      throw StackEmptyException("Access to empty stack");
   return array[topOfStack--];
}
// copy constructor
ArrayStack::ArrayStack(const ArrayStack& st):
   capacity(st.capacity),
   array(new int[capacity]),
   topOfStack(st.topOfStack)
{
   for (int i = 0; i <= topOfStack; i++) // copy contents
      array[i] = st.array[i];
}
```

The Implementation of Assignment Operator

```
// assignment operator
ArrayStack& ArrayStack::
operator=(const ArrayStack& st)
{
   if (this != &st) { // avoid self copy (x = x)
      delete [] array; // delete old contents
      capacity = st.capacity;
      topOfStack = st.topOfStack;
      array = new int[capacity];
      // copy contents
      for (int i = 0; i <= topOfStack; i++)
         array[i] = st.array[i];
   }
   return *this;
}
```

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 pop or top on an empty stack throws a StackEmpty exception

Templated Stack

```
template <typename E> class ArrayStack
{
   private: E* S; // array holding the stack
   int cap; // capacity
   int t; // index of top element
   public: // constructor given capacity
   ArrayStack( int c): S(new E[c]), cap(c), t(-1) { }
   void pop()
   {
      if (empty()) throw StackEmpty("Pop from empty stack");
      t--;
   }
   void push(const E& e)
   {
       if (size() == cap) throw StackFull("Push to full stack");
       S[++t] = e;
   } //... (other methods of Stack interface)
}
```

Example

The current top of the stack is denoted by the star (*)

```
ArrayStack<int> A;                    // A = [ ], size = 0
A.push(7);                            // A = [7*], size = 1
A.push(13);                           // A = [7, 13*], size = 2
cout << A.top() << endl;              // A = [7*] outputs: 13
A.pop();                              // outputs 7, size = 0
ArrayStack<string> B(10);             // B = [ ], size = 0
B.push("apple");          // B = [apple*], size = 1
B.push("green");          // B = [apple, green*], size = 2
cout << B.top()
     << endl; B.pop();// B = [apple*], outputs: green
```

Performance and Limitations

If n is the number of elements in the stack, then

- each stack operation runs in time $O(1)$
- the space used is $O(n)$

However, the size of the stack is fixed at runtime and trying to push a new element into a full stack throws an exception. We can avoid this situation by using an expandable array in the implementation of the stack.
Expandable dynamic arrays were discussed previously. In order to use an expandable dynamic array implementation of the stack we need to add one new function and change `push`.

The Expandable Arrays

- In an insert operation (without an index), we always insert at the end
- When the array is full, we 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

Doubling Strategy for The Expandable Arrays

```
//a private function double size of an array
void ArrayStack::doubleSize( )
{  capacity *= 2;
   int newArray[] = new int[capacity];
   for (int i = 0; i <= topOfStack; i++)
      newArray[i] = array[i];
   delete [] array; // remove old contents
   array = newArray; // switch pointers
}
// push an object onto the stack (new implementation)
void ArrayStack::push(const int& elem)
{  if (size() == capacity)
      doubleSize();
   array[++topOfStack] = elem;
}
```

Amortized Analysis of the Push Operations

The worst case analysis is not very useful when applied to the sequence of push operations based on expandable dynamic array. *Amortized analysis* can be used to show that the average cost of a push operation is still $O(1)$.

Assume that the initial capacity of the dynamic array is 1. If the array is full we double its capacity. So we have the following sequence of capacities: 1, 2, 4, 8, 16, ... We focus on the average cost of the push operation. Let denote by c_i the cost of the i-th push operation. If the array is not full, then $c_i = 1$, since only one operation is performed. If the array is full, an expansion occurs, and then $c_i = i$, that is, we need to include the cost of additional $i - 1$ copy operations (from the smaller array to the larger one) plus one push operation.

To insert n objects, we replace the array $k = log_2 n$ times.

Amortized Analysis of the Push Operations

The ith push operation which causes an array expansion occurs only when $i-1$ is an exact power of 2. We know that

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

The total cost of n push operations is therefore

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j < n + 2n = 3n,$$

since there are at most n operations that cost 1 and the cost of the copy operations form a geometric series. The amortized cost of a single push operation is therefore $3n/n = 3$, that is, it is $O(1)$.
Note that the worst case analysis gives you $O(n^2)$ which is an overestimate, because it does not take into account the fact that the array expansion happens only from time to time (not for every push operation).
Notice that if we increment the size of the array by a constant (and not by doubling), the amortized cost of the push operation is $O(n)$.

Incremental Strategy Analysis

If there is no space for a new object increase the size by a constant c

To insert n objects, we replace the array $k = n/c$ times

The total time $T(n)$ of a series of n insert operations is proportional to

$n + c + 2c + 3c + 4c + \ldots + kc = n + c(1 + 2 + 3 + \ldots + k) = n + ck(k + 1)/2$

Since c is a constant, $T(n)$ is $O(n + k^2)$, i. e., $O(n^2)$.

The amortized time of an insert operation is $O(n)$.

Stack Applications

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of function calls in the C++ run-time system
- Auxiliary data structure for algorithms
- Component of other data structures

Parentheses Matching

Example

PROBLEM: Check an expression to see if the parentheses match correctly.
Each "(", "{", or "[" must be paired with a matching ")", "}", or "]", respectively.
correct: ()(())([()])
correct: ((()(()))([()])
incorrect:)(())([()])
incorrect: ([])
incorrect: (

Parentheses Matching - Algorithm (cont.)

Consider the following three kinds of parentheses: `( )`, `{ }`, or `[ ]`. Any symbol other than one of these is ignored.

TESTING EXAMPLES: `{[x+y*(z+7)]*(a+b)}` (each of the left parentheses has a corresponding right parenthisis), and `((x+y*{z+7}*[a+b])` (no matching for the most outward pair of parentheses).

ALGORITHM: Scan characters of the expression from left to right. Every time a left parenthesis occurs, it is pushed onto the stack. Every time a right parenthesis occurs, a matching left parenthesis is popped off of the stack. But if the matching left parenthesis is not on top of the stack or the stack is empty, then the string is unbalanced (return `false`). Otherwise it is balanced (return `true`).

C++ Implementation

```
bool checkOpeningParen(ArrayStack &s, const char oparen)
{  bool failed = false;
   if (s.isEmpty()) failed = true;
   else { // check an opening parenthesis
      char top = static_cast<char>(s.pop());
      if (top != oparen) failed = true;
   }
   return failed;
}
```

C++ Implementation (cont)

```
bool isBalanced(const string &expr)
{  ArrayStack s; // new stack object
   bool failed = false;
   for (int i = 0; !failed && (i < expr.length()); i++) {
      char c = expr.at(i);
      switch (c) {
      case '(':  case '{':  case '[': s.push(c); break;
      case ')':
         failed = checkOpeningParen(s, '('); break;
      case '}':
         failed = checkOpeningParen(s, '{'); break;
      case ']':
         failed = checkOpeningParen(s, '['); break;
      } // end case
   } // end for loop
   return (s.isEmpty() && ! failed);
}
```

C++ Implementation (cont.)

```
int main() {
   string expr1("{[x+y*(z+7)]*(a+b)}");
   string expr2("((x+y*{z+7}*[a+b])");
   cout << "Is expression 1 balanced: "
        << (isBalanced(expr1) ? "true" : "false")<<endl;
   cout << "Is expression 2 balanced: "
        << (isBalanced(expr2) ? "true" : "false")<<endl;
   return 0;
}
```