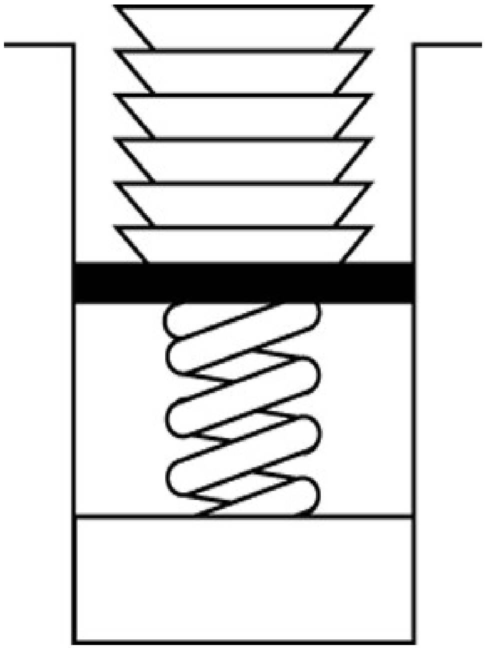
Stacks

The Stack ADT

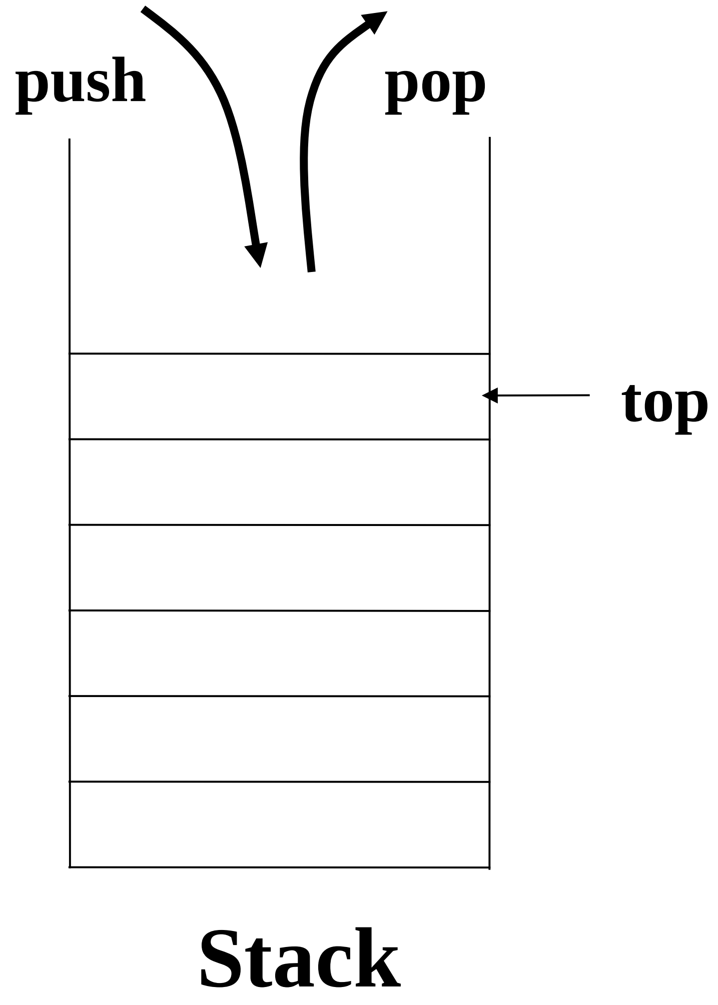
- The Stack ADT stores arbitrary objects.
- Insertions and deletions follow the *last-in first-out* (LIFO) scheme.
    - The last item placed on the stack will be the first item removed.
      (similar to a stack of dishes)



Stack of Dishes

ADT Stack Operations

- Create an empty stack
- Destroy a stack
- Determine whether a stack is empty
- Add a new item -- **push**
- Remove the item that was added most recently -- **pop**
- Retrieve the item that was added most recently



ADT Stack Operations(cont.)

- **Stack**()
  - creates a an empty stack
- **~Stack**()
  - destroys a stack
- **isEmpty**():boolean
  - determines whether a stack is empty or not
- **push**(in newItem:StackItemType)
  - Adds newItem to the top of a stack
- **pop**() throw StackException
- **topAndPop**(out stackTop:StackItemType)
  - Removes the top of a stack (ie. removes the item that was added most recently
- **getTop**(out stackTop:StackItemType)
  - Retrieves the top of stack into stackTop

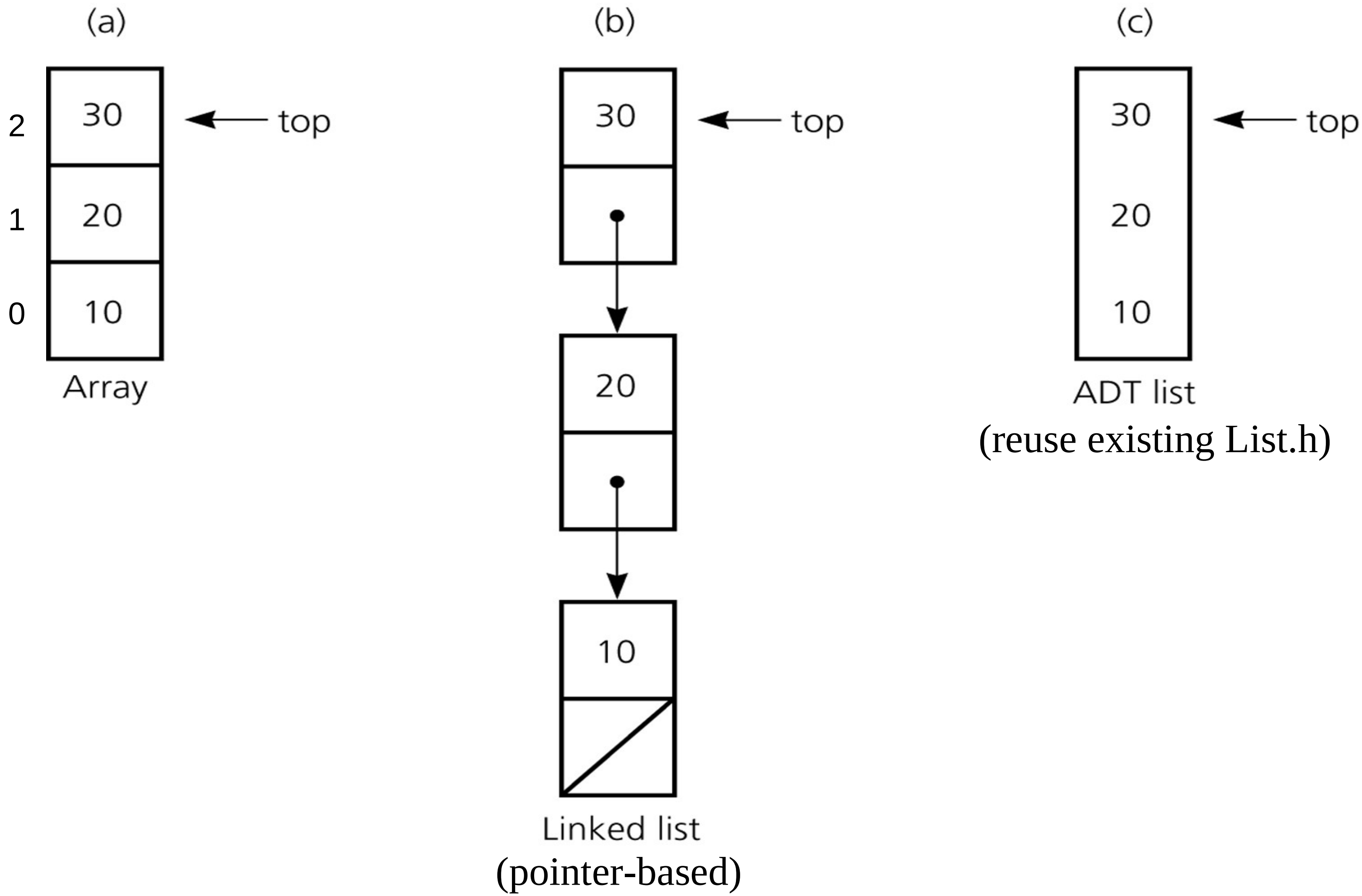
Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list
  - The ADT list (linked list of the previous lecture)

- All three implementations use a `StackException` class to handle possible exceptions

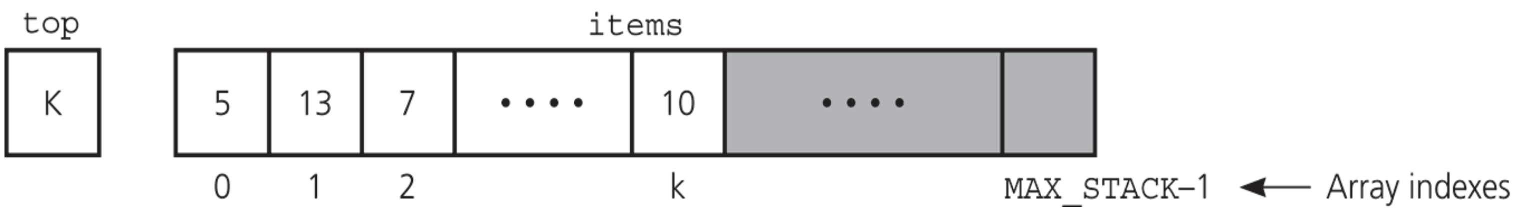
```
class StackException {
public:
    StackException(const string& err) : error(err) {}
    string error;
};
```

Implementations of the ADT Stack (cont.)



An Array-Based Implementation of the ADT Stack

- Private data fields
  - An array of `items` of type `StackItemType`
  - The index `top`
- Compiler-generated destructor, copy constructor, and assignment operator



An Array-Based Implementation –Header File

```
#include "StackException.h"
const int MAX_STACK = maximum-size-of-stack;
template <class T>
class Stack {
public:
   Stack();  // default constructor; copy constructor and destructor are supplied by the
     compiler

   // stack operations:
   bool isEmpty() const;                    // Determines whether a stack is empty.
   void push(const T& newItem);             // Adds an item to the top of a stack.
   void pop();                      // Removes the top of a stack.
   void topAndPop(T& stackTop);
   void getTop(T& stackTop) const;   // Retrieves top of stack.

private:
   T items[MAX_STACK];  // array of stack items
   int top;                      // index to top of stack
};
```

An Array-Based Implementation

```
template <class T>
Stack<T>::Stack(): top(-1) {}      // default constructor


template <class T>
bool Stack<T>::isEmpty() const {
  return top < 0;
}
```

An Array-Based Implementation

```
template <class T>
void Stack<T>::push(const T& newItem) {

  if (top >= MAX_STACK-1)
    throw StackException("StackException: stack full on push");
  else
    items[++top] = newItem;
 }
```

An Array-Based Implementation – pop

```
template <class T>
void Stack<T>::pop() {

   if (isEmpty())
      throw StackException("StackException: stack empty on pop");
   else
      --top;     // stack is not empty; pop top
}
```

An Array-Based Implementation – pop

```
template <class T>
void Stack<T>::topAndPop(T& stackTop) {

   if (isEmpty())
      throw StackException("StackException: stack empty on pop");
   else // stack is not empty; retrieve top
      stackTop = items[top--];
 }
```

An Array-Based Implementation – getTop

```
template <class T>
void Stack<T>::getTop(T& stackTop) const  {
   if (isEmpty())
      throw StackException("StackException: stack empty on getTop");
   else
       stackTop = items[top];
}
```

An Array-Based Implementation

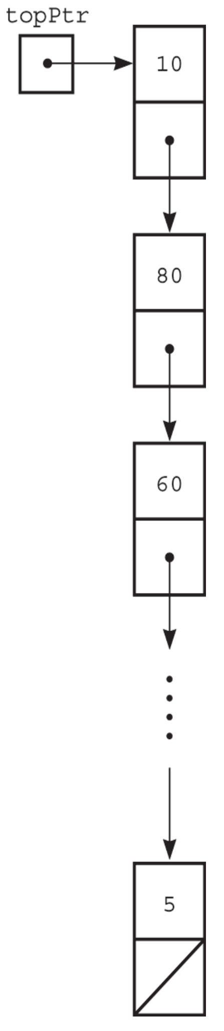
- Disadvantages of the array based implementation is similar the disadvantages of arrays
  - It forces all stack objects to have MAX_STACK elements

A Pointer-Based Implementation of the ADT Stack

- A pointer-based implementation
  - Required when the stack needs to grow and shrink dynamically
  - Very similar to linked lists

- `top` is a reference to the head of a linked list of items

- A copy constructor, assignment operator, and destructor must be supplied



A Pointer-Based Implementation – Header File

```
template <class Object>
class StackNode
{
   public:
      StackNode(const Object& e = Object(), StackNode* n = NULL)
         : element(e), next(n) {}

      Object item;
      StackNode* next;
};
```

A Pointer-Based Implementation – Header File

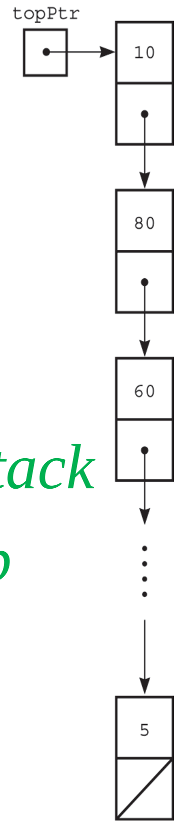
```
#include "StackException.h"
template <class T>
class Stack{
public:
   Stack();                                          // default constructor
   Stack(const Stack& rhs);                          // copy constructor
   ~Stack();                                         // destructor
   Stack& operator=(const Stack& rhs);               // assignment operator
   bool isEmpty() const;
   void push(const T& newItem);
   void pop();
   void topAndPop(T& stackTop);
   void getTop(T& stackTop) const;
private:
   StackNode<T> *topPtr;                    // pointer to the first node in the stack
};
```

A Pointer-Based Implementation – constructor and isEmpty

```
template <class T>
Stack<T>::Stack() : topPtr(NULL) {}   // default constructor

template <class T>
bool Stack<T>::isEmpty() const
{
   return topPtr == NULL;
}
```

A Pointer-Based Implementation – push

```
template <class T>
void Stack<T>::push(const T& newItem) {
   // create a new node
   StackNode *newPtr = new StackNode;

   newPtr->item = newItem; // insert the data


   newPtr->next = topPtr; // link this node to the stack
   topPtr = newPtr;                // update the stack top
}
```



A Pointer-Based Implementation – pop

```
template <class T>
void Stack<T>::pop() {
  if (isEmpty())
    throw StackException("StackException: stack empty on pop");
  else {
    StackNode<T> *tmp = topPtr;
    topPtr = topPtr->next; // update the stack top
    delete tmp;
  }
}
```

A Pointer-Based Implementation – topAndPop

```
template <class T>
void Stack<T>::topAndPop(T& stackTop) {
  if (isEmpty())
    throw StackException("StackException: stack empty on
   topAndPop");
  else {
    stackTop = topPtr->item;
    StackNode<T> *tmp = topPtr;
    topPtr = topPtr->next; // update the stack top
    delete tmp;
  }
}
```

A Pointer-Based Implementation – getTop

```
template <class T>
void Stack<T>::getTop(T& stackTop) const  {
   if (isEmpty())
      throw StackException("StackException: stack empty on getTop");
   else
      stackTop = topPtr->item;
}
```

A Pointer-Based Implementation – destructor

```
template <class T>
Stack<T>::~Stack() {
  // pop until stack is empty
  while (!isEmpty())
     pop();
}
```

A Pointer-Based Implementation – assignment

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack& rhs) {
    if (this != &rhs) {
        if (!rhs.topPtr)
            topPtr = NULL;
        else {
            topPtr = new StackNode<T>;
            topPtr->item = rhs.topPtr->item;
            StackNode<T>* q = rhs.topPtr->next;
            StackNode<T>* p = topPtr;
            while (q) {
                p->next = new StackNode<T>;
                p->next->item = q->item;
                p = p->next;
                q = q->next;
            }
            p->next = NULL;
        }
    }
    return *this;
}
```

A Pointer-Based Implementation – copy constructor

```
template <class T>
Stack<T>::Stack(const Stack& rhs) {
    *this = rhs; // reuse assignment operator
}
```

Testing the Stack Class

```
int main() {
    Stack<int> s;
    for (int i = 0; i < 10; i++)
        s.push(i);

    Stack<int> s2 = s; // test copy constructor (also tests assignment)

    std::cout << "Printing s:" << std::endl;
    while (!s.isEmpty()) {
        int value;
        s.topAndPop(value);
        std::cout << value << std::endl;
    }
```

Testing the Stack Class

```
    std::cout << "Printing s2:" << std::endl;
    while (!s2.isEmpty()) {
        int value;
        s2.topAndPop(value);
        std::cout << value << std::endl;
    }

    return 0;
}
```

An Implementation That Uses the ADT List

```
#include "StackException.h"
#include "List.h"

template <class T>
class Stack{
public:

   bool isEmpty() const;
   void push(const T& newItem);
   void pop();
   void topAndPop(T& stackTop);
   void getTop(T& stackTop) const;

private:
    List<T> list;
}
```

An Implementation That Uses the ADT List

- No need to implement constructor, copy constructor, destructor, and assignment operator
  - The `list`'s functions will be called when needed

- **isEmpty():** return list.isEmpty()
- **push(x):** list.insert(x, list.zeroth())
- **pop():** list.remove(list.first()->element)
- **topAndPop(&x)** and **getTop(&x)** are similar

Comparing Implementations

- **Array based:**
  - Fixed size (cannot grow and shrink dynamically)
- **Using a single pointer:**
  - May need to perform realloc calls when the currently allocated size is exceeded
  - But push and pop operations can be very fast
- **Using a customized linked-list:**
  - The size can match perfectly to the contained data
  - Push and pop can be a bit slower than above
- **Using the previously defined linked-list:**
  - Reuses existing implementation
  - Reduces the coding effort but may be a bit less efficient

A Simple Stack Application -- Undo sequence in a text editor

Problem: abcd← ←fg ← ➔ abf

// Reads an input line. Enter a character or a backspace character (←)

readAndCorrect(out aStack:Stack) {

?????????????????????????????

}

A Simple Stack Application -- Undo sequence in a text editor

Problem: abcd← ←fg ← ➔ abf

```
// Reads an input line. Enter a character or a backspace character (←)
readAndCorrect(out aStack:Stack) {
   aStack.createStack()
   read newChar
   while (newChar is not end-of-line symbol) {
      if (newChar is not backspace character)
         aStack.push(newChar)
      else if (!aStack.isEmpty())
         aStack.pop()
      read newChar
   }
}
```

A Simple Stack Application -- Display Backward

- Display the input line in reversed order by writing the contents of stack aStack.

```
displayBackward(in aStack:Stack) {

?????????


}
```

A Simple Stack Application -- Display Backward

- Display the input line in reversed order by writing the contents of stack aStack.

```
displayBackward(in aStack:Stack) {
    while (!aStack.isEmpty())) {
        aStack.pop(newChar)
        print newChar
    }
}
```

Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
- An example of balanced braces

  `abc{defg{ijk}{l{mn}}op}qr`

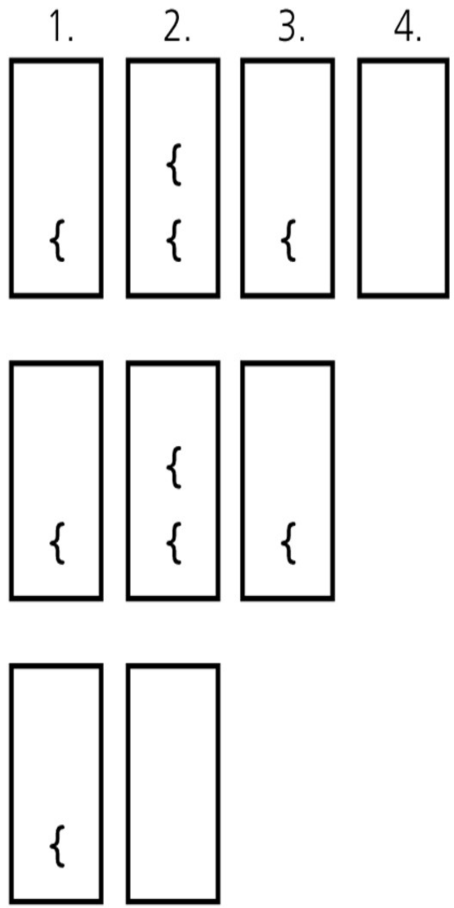
- An example of unbalanced braces

  `abc{def}}{ghij{kl}m`

- Requirements for balanced braces
  - Each time we encounter a “}”, it matches an already encountered “{”
  - When we reach the end of the string, we have matched each “{”

Checking for Balanced Braces -- Traces

Input string — Stack as algorithm executes



`{a{b}c}`

1. push "{"
2. push "{"
3. pop
4. pop

Stack empty ⟹ balanced

`{a{bc}`

1. push "{"
2. push "{"
3. pop

Stack not empty ⟹ not balanced

`{ab}c}`

1. push "{"
2. pop

Stack empty when last "}" encountered ⟹ not balanced

Checking for Balanced Braces -- Algorithm

```
aStack.createStack();        balancedSoFar = true;       i=0;
while (balancedSoFar and i < length of aString) {
    ch = character at position i in aString;     i++;
    if (ch is ‘{‘)                              // push an open brace
        aStack.push(‘{‘);
    else if (ch is ‘}’)                         // close brace
        if (!aStack.isEmpty())
            aStack.pop();                       // pop a matching open brace
        else                                    // no matching open brace
            balancedSoFar = false;
    // ignore all characters other than braces
}
if (balancedSoFar and aStack.isEmpty())
    aString has balanced braces
else
    aString does not have balanced braces
```

Recognizing Strings in a Language

- L = {w$w' : w is a (possible empty) string of characters other than $,
  w' = reverse(w) }
  - abccba, aa, $, abc$abc, ab, a are in or out the language L?

Recognizing Strings in a Language

- L = {w$w' : w is a (possible empty) string of characters other than $,
  w' = reverse(w) }
  - abccba, aa, $ are in the language L
  - abcabc, ab, a$ are not in the language L

- Problem: Deciding whether a given string in the language L or not.

- A solution using a stack
  - ??
  - ??

Recognizing Strings in a Language

- L = {w$w' : w is a (possible empty) string of characters other than $,

  w' = reverse(w) }
  - abccba, aa, $ are in the language L
  - abcabc, ab, a$ are not in the language L

- Problem: Deciding whether a given string in the language L or not.

- A solution using a stack
  - Traverse the first half of the string, pushing each char onto a stack
  - Once you reach the $, for each character in the second half of the string, match a popped character off the stack

Recognizing Strings in a Language -- Algorithm

```
aStack.createStack();    i=0;   ch = character at position i in aString;
while (ch is not '$') {  // push the characters before $ (w) onto the stack
    aStack.push(ch);   i++;   ch = character at position i in aString;
}
i++;  inLanguage = true;          // skip $; assume string in language
while (inLanguage and i <length of aString)     // match the reverse of
    if (aStack.isEmpty())  inLanguage = false;  // first part shorter than second part
    else {
        aStack.pop(stackTop);    ch = character at position i in aString;
        if (stackTop equals to ch)  i++;           // characters match
        else inLanguage = false;               // characters do not match
    }
if (inLanguage and aStack.isEmpty())
    aString is in language
else
    aString is not in language
```

Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation
- To evaluate an infix expression *//**in**fix: operator **in** b/w operands*
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression *//**post**fix: operator **after** operands; similarly we have **pre**fix: operator **before** operands*

Infix Expression	Postfix Expression	Prefix Expression
5 + 2 * 3		
5 * 2 + 3		
5 * (2 + 3) - 4		

Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation
- To evaluate an infix expression *//**in**fix: operator **in** b/w operands*
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression *//**post**fix: operator **after** operands; similarly we have **pre**fix: operator **before** operands*

Infix Expression	Postfix Expression	Prefix Expression
5 + 2 * 3	5 2 3 * +	+ 5 * 2 3
5 * 2 + 3	5 2 * 3 +	+ * 5 2 3
5 * (2 + 3) - 4	5 2 3 + * 4 -	- * 5 + 2 3 4

Application: Algebraic Expressions

- Infix notation is easy to read for humans, whereas pre-/postfix notation is easier to parse for a machine. The big advantage in pre-/postfix notation is that there never arise any questions like operator precedence
- Or, to put it in more general terms: it is possible to restore the original (parse) tree from a pre-/postfix expression without any additional knowledge, but the same isn't true for infix expressions

- For example, consider the infix expression 1 # 2 $ 3. Now, we don't know what those operators mean, so there are two possible corresponding postfix expressions: 1 2 # 3 $ and 1 2 3 $ #. Without knowing the rules governing the use of these operators, the infix expression is essentially worthless.

Evaluating Postfix Expressions

- When an operand is entered, the calculator
  - Pushes it onto a stack

- When an operator is entered, the calculator
  - Applies it to the top two operands of the stack
  - Pops the operands from the stack
  - Pushes the result of the operation on the stack

Evaluating Postfix Expressions: 2 3 4 + *

Key entered	Calculator action		After stack operation: Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

Converting Infix Expressions to Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

Converting Infix Expressions to Postfix Expressions

ch	Stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

a - (b + c * d)/ e ➔ a b c d * + e / -

Converting Infix Expr. to Postfix Expr. -- Algorithm

```
for (each character ch in the infix expression) {
  switch (ch) {
    case operand:      // append operand to end of postfixExpr
       postfixExpr=postfixExpr+ch;   break;
    case ‘(‘:          // save ‘(‘ on stack
       aStack.push(ch);   break;
    case ‘)’:          // pop stack until matching ‘(‘, and remove ‘(‘
       while (top of stack is not ‘(‘) {
          postfixExpr=postfixExpr+(top of stack);   aStack.pop();
       }
       aStack.pop();   break;
```

Converting Infix Expr. to Postfix Expr. -- Algorithm

```
  case operator:
      aStack.push();   break;            // save new operator
} } // end of switch and for

// append the operators in the stack to postfixExpr
while (!isStack.isEmpty()) {
    postfixExpr=postfixExpr+(top of stack);
     aStack(pop);
}
```

The Relationship Between Stacks and Recursion

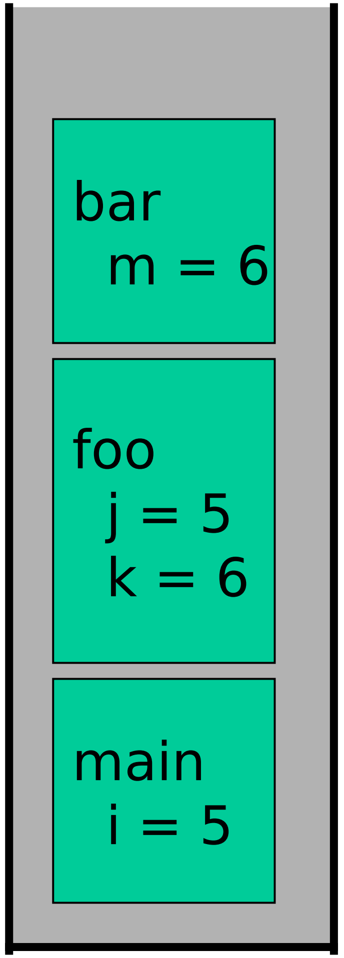
- A strong relationship exists between recursion and stacks
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack

  That's why we can get **stack overflow** error if a function makes makes too many recursive calls
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

C++ Run-time Stack

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

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

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

bar(int m) {
  …
  }
```



Run-time Stack

Example: Factorial function

```
int fact(int n)
{
  if (n ==0)
   return (1);
  else
      return (n * fact(n-1));
}
```

Tracing the call `fact (3)`

			N = 0 if (N==0) true **return (1)**
		N = 1 if (N==0) false return (1***fact(0)**)	N = 1 if (N==0) false return (1***fact(0)**)
	N = 2 if (N==0) false return (2***fact(1)**)	N = 2 if (N==0) false return (2***fact(1)**)	N = 2 if (N==0) false return (2***fact(1)**)
N = 3 if (N==0) false return (3***fact(2)**)	N = 3 if (N==0) false return (3***fact(2)**)	N = 3 if (N==0) false return (3***fact(2)**)	N = 3 if (N==0) false return (3***fact(2)**)
After original call	After 1st call	After 2nd call	After 3rd call

Tracing the call `fact (3)`

N = 1
if (N==0) false
return (1* **1**)

N = 2
if (N==0) false
return (2***fact(1)**)

N = 3
if (N==0) false
return (3***fact(2)**)

After return from 3rd call

N = 2
if (N==0) false
return (2* **1**)

N = 3
if (N==0) false
return (3***fact(2)**)

After return from 2nd call

N = 3
if (N==0) false
return (3* **2**)

After return from 1st call

return **6**

Example: Reversing a string

```
void printReverse(const char* str)
{

    ????????



}
```

Example: Reversing a string

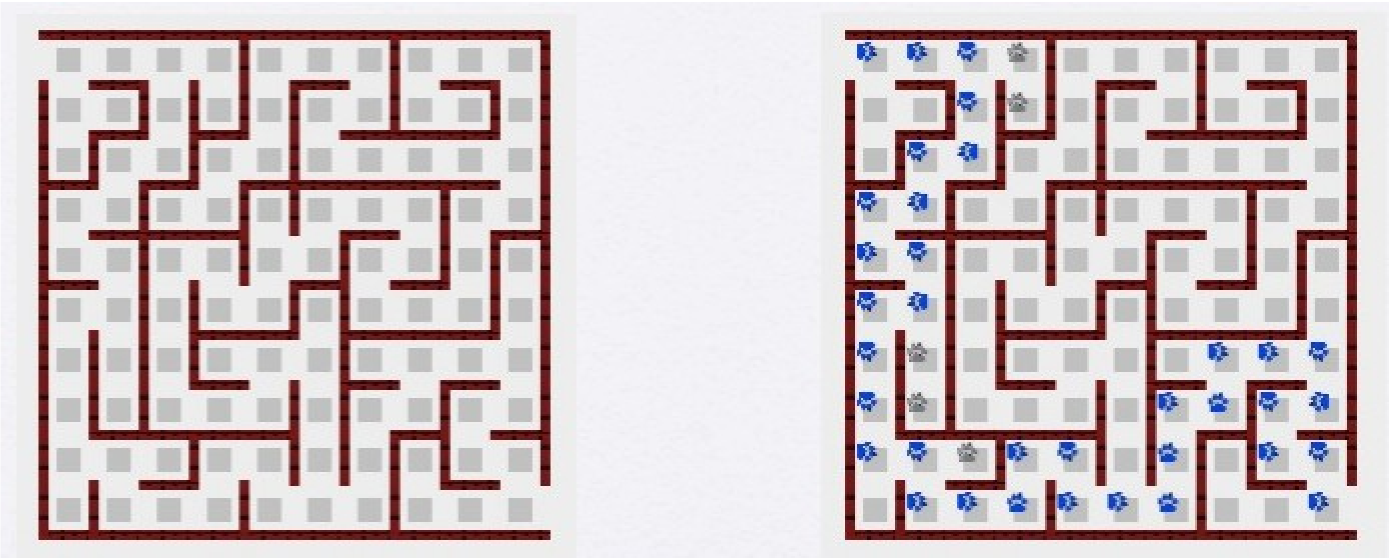
```
void printReverse(const char* str)
{
    if (*str) {
        printReverse(str + 1)
        cout << *str << endl;
    }
}
```

Example: Reversing a string

```
void printReverseStack(const char* str)
{
    Stack<char> s;
    for (int i = 0; str[i] != ’\0’; ++i)
        s.push(str[i]);

    while(!s.isEmpty()) {
        char c;
        s.topAndPop(c);
        cout << c;
    }
}
```

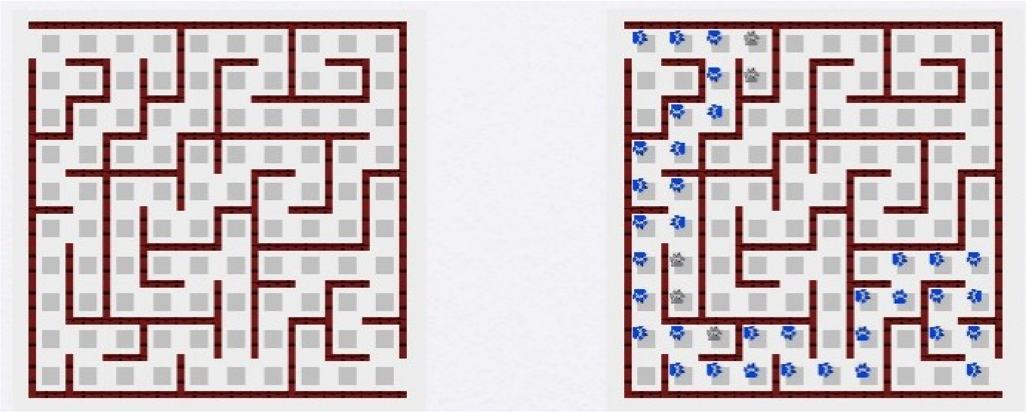
Example: Maze Solving

- Find a path on the maze represented by 2D array

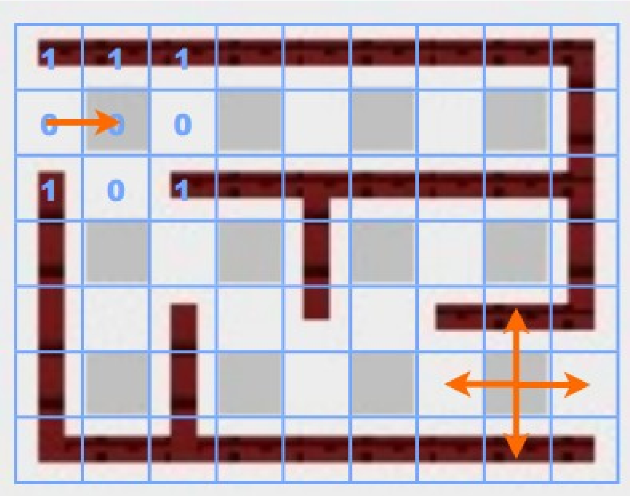


Example: Maze Solving

- Find a path on the maze represented by 2D array

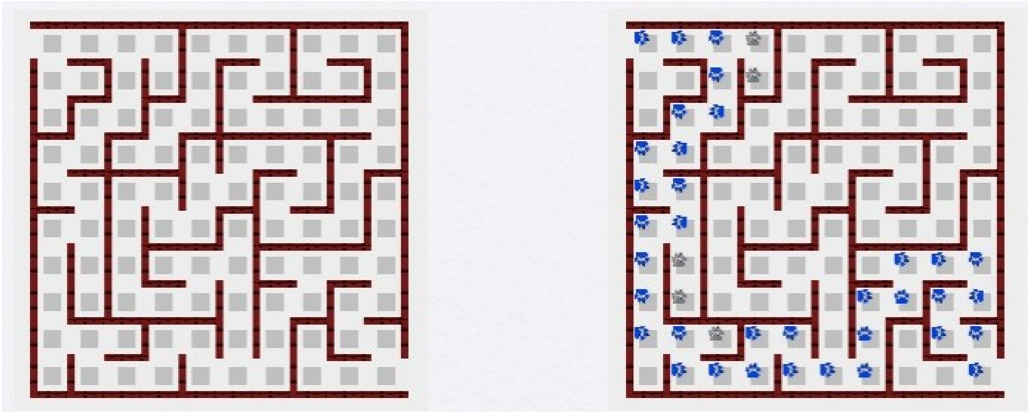


- Push the traced points into a stack

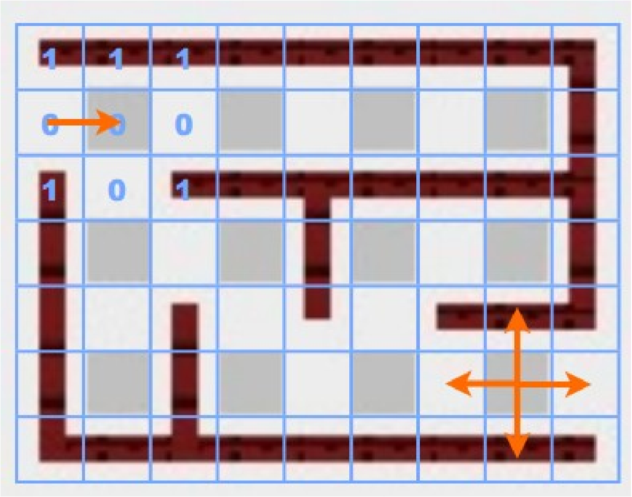


Example: Maze Solving

- Find a path on the maze represented by 2D array



- Move forward if possible. If not, pop until a new direction move is possible.



Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?

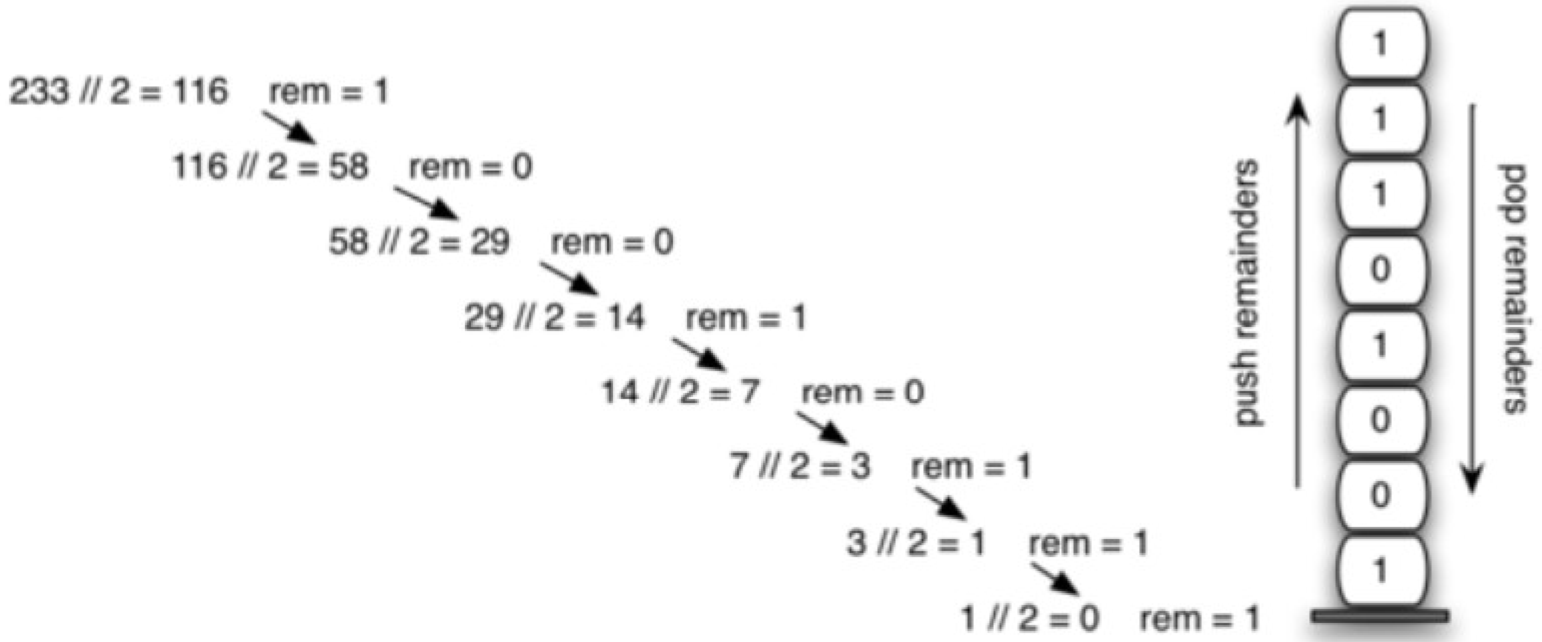
Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?

233 // 2 = 116 rem = 1
116 // 2 = 58 rem = 0
58 // 2 = 29 rem = 0
29 // 2 = 14 rem = 1
14 // 2 = 7 rem = 0
7 // 2 = 3 rem = 1
3 // 2 = 1 rem = 1
1 // 2 = 0 rem = 1

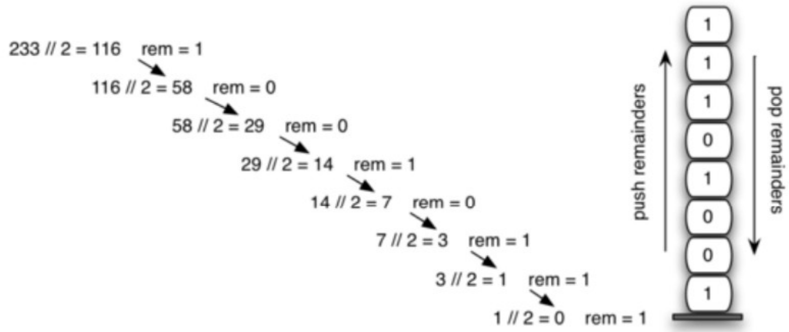
Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?



Example: Integer to Binary

- Binary to integer is easy; how about the reverse?
- What is 233 in binary?




```
while dec_number > 0:
   rem = dec_number % 2
   rem_stack.push(rem)
   dec_number = dec_number / 2

bin_string = ""
while not rem_stack.is_empty():
   bin_string = bin_string + str(rem_stack.pop())

return bin_string
```