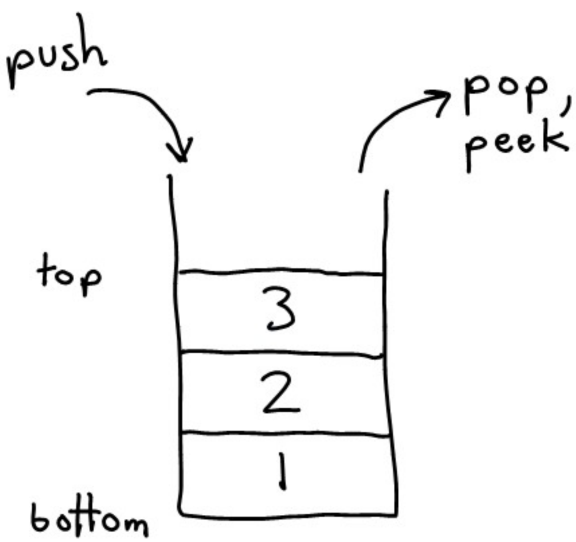
Stack Abstract Data Type

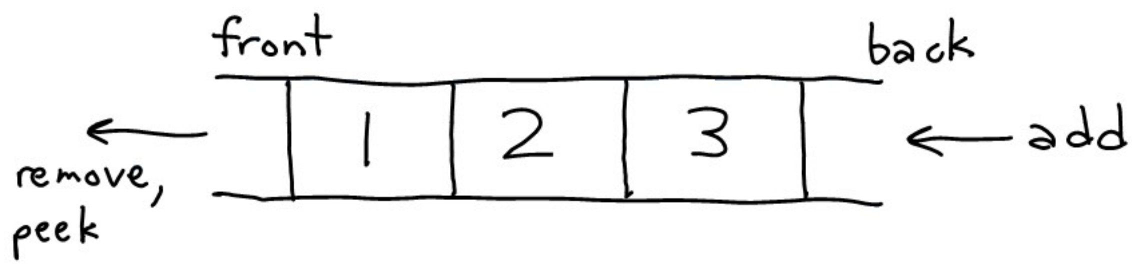
Stacks and queues

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.

- We will examine two specialty collections:
  - **stack**: Retrieves elements in the reverse of the order they were added.
  - **queue**: Retrieves elements in the same order they were added.



stack



queue

Abstract data types (ADTs)

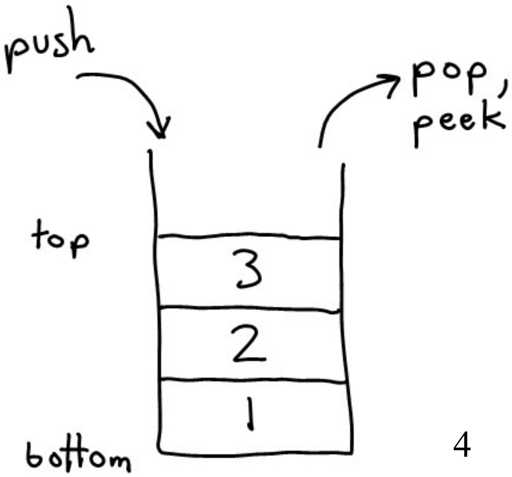
- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it

- Even if we don't know exactly how a stack or queue is implemented,
  - We just need to understand the idea of the collection and what operations it can perform.

Stack

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").

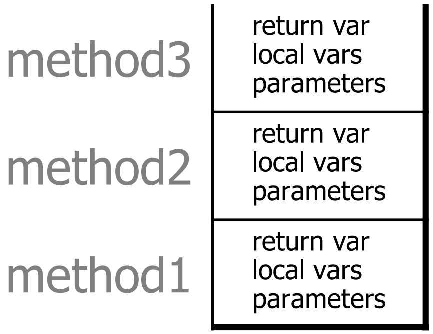
- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



Stacks

Stacks in computer science

- Programming languages and compilers:
  - Method/function calls are placed onto a stack *(call=push, return=pop)*
  - compilers use stacks to evaluate expressions



- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } and other operators match
  - convert "infix" expressions to "postfix" or "prefix"

- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

Class `Stack`

`Stack<`**`E`**`>()`	constructs a new stack with elements of type **E**
`push(`**`value`**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);         // bottom [42, -3, 17] top
cout << s.pop();  // 17
```

Stack limitations

- Remember: You cannot loop over a stack in the usual way.

```
Stack<int> s ;
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Instead, you must pull contents out of the stack to view them.
  - common idiom: Removing each element until the stack is empty.

```
while (!s.isEmpty()) {
    do something with s.pop();
}
```

Exercise

- Consider an input file of exam scores in reverse ABC order:

```
Yeilding       Janet      87
White          Steven     84
Todd           Kim        52
Tashev         Sylvia     95
...
```

- Write code to print the exam scores in ABC order using a stack.

Execise solution

```
ifstream file;
Stack<string> names;  // stack of strings

file.open("data.txt");
while (file.good())
{
    getline(file, line);
    names.push(line);
}
file.close();

while(!names.isEmpty()){
    cout << names.pop() << endl;
}
```

What happened to my stack?

- Suppose we're asked to write a method `max` that accepts a Stack of integers and returns the largest integer in the stack.
  - The following solution is seemingly correct:

```
// Precondition: s.size() > 0
int max(Stack<int> & s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = max(maxValue, next);
    }
    return maxValue;
}
```

  - The algorithm is correct, but what is wrong with the code?

What happened to my stack?

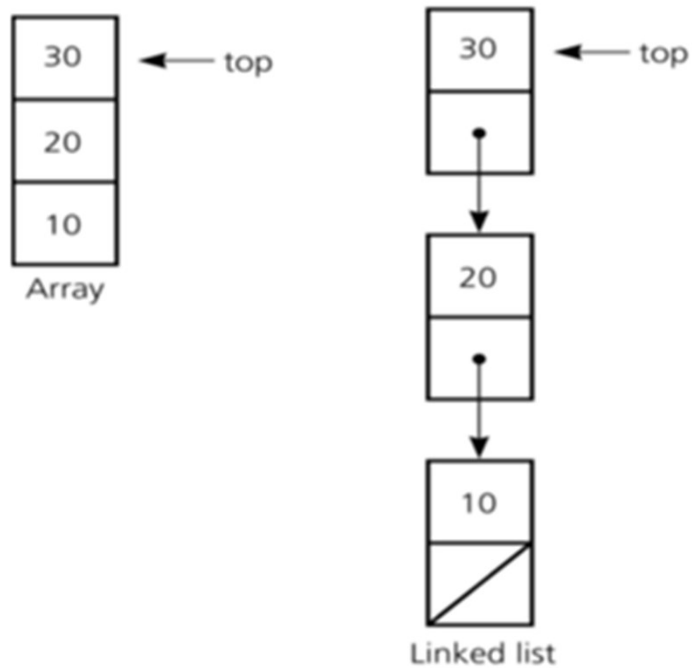
- The code destroys the stack in figuring out its answer.
  - To fix this, you must save and restore the stack's contents:

```
int max(Stack<int> & s) {
    Stack<int> backup;
    int maxValue = s.pop();
    backup.push(maxValue);
    while (!s.isEmpty()) {
        int next = s.pop();
        backup.push(next);
        maxValue = max(maxValue, next);
    }
    while (!backup.isEmpty()) {
        s.push(backup.pop());
    }
    return maxValue;
}
```

Implementation of Stack Abstract Data Type

The stack abstract data type can be implemented using either

- An array, or
- A linked list



Fixed size versus dynamic size:

- An array-based implementation prevents the push operation from adding an item to the stack if the stack's size limit has been reached
- A pointer-based implementation does not put a limit on the size of the stack

Array-Based Implementation of Stack

– An array of `items`

– The index `top`

StackException

- We will use a `StackException` class to handle possible exceptions

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

Array-Based Implementation of Stack

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

   bool isEmpty() const;  // Determines if stack is empty.
   void push(const T& newItem); // Adds an item to the top of
                                // a stack.
   T  pop();        // Removes and returns the top of a stack.
   T  peek() 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(){// default constructor
   top = -1;
}


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

An Array-Based Implementation – pop

```
template <class T>
T Stack<T>::pop() {
  if (isEmpty())
     throw StackException("StackException: stack empty
  on pop");
  else // stack is not empty; return top
     return(items[top--]);
 }
```

An Array-Based Implementation - push

```
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 – peek

```
template <class T>
T Stack<T>::peek() const  {
    if (isEmpty())
        throw StackException("StackException: stack empty
   on peek");
    else
         return(items[top]);
}
```

An Array-Based Implementation

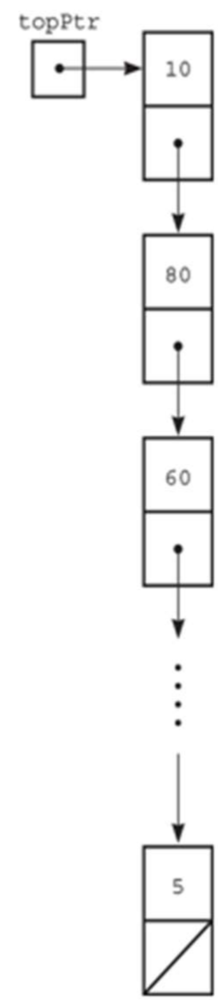
- Disadvantages of the array based implementation : fixed size
  - it forces all stack objects to have MAX_STACK elements
- We can fix this limitation by using a dynamic array instead of an array

```
template <class T>
class Stack {
public:
   Stack(int size) : items(new T [size]) { };
   ... // other parts not shown
private:
   T* items;     // pointer to the stack elements
   int top;               // index to top of stack
};
```

"Need to implement copy constructor, destructor and assignment operator in this case"

Implementing Stack as a Linked List

- `top` is a pointer to the front of a linked list of items
- A copy constructor, assignment operator, and destructor must be supplied



Stack as linked nodes

```
template <class T>
class StackNode
{
  public:
    StackNode(const T& e = T(), StackNode* n = nullptr){
      item = e;
      next = n;
    }
    T item;
    StackNode* next;
};
```

A Pointer-Based Implementation

```
#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);
   T  pop();
   T  peek() 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(){      // default constructor
  topPtr=nullptr;
}

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

A Pointer-Based Implementation – push

```
template <class T>
void Stack<T>::push(const T& newItem) {
   // create a new node
   StackNode<T> *newPtr = new StackNode<T>;
   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>
T Stack<T>::pop() {
    if (isEmpty())
        throw StackException("StackException: stack empty
   on  pop");
    else {
        T stackTop = topPtr->item;
        StackNode<T> *tmp = topPtr;
        topPtr = topPtr->next; // update the stack top
        delete tmp;
        return stackTop;
    }
}
```

A Pointer-Based Implementation – peek

```
template <class T>
T  Stack<T>::peek() const  {
   if (isEmpty())
      throw StackException("StackException: stack empty
   on peek");
   else
       return(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 <T>& rhs) {
   if (this != &rhs) {
      while (!isEmpty()) pop();
      if (!rhs.topPtr)  topPtr = nullptr;
      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 = nullptr;
        }
    }
    return *this;
}
```

A Pointer-Based Implementation – copy constructor

```
template <class T>
Stack<T>::Stack(const Stack<T>& rhs) {
    topPtr = new StackNode<T> ;
    *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)

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

Testing the Stack Class

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

    return 0;
}
```

An Implementation That Uses the ADT List

```
#include "StackException.h"
#include "LinkedList.h"

template <class T>
class Stack{
public:

   bool isEmpty() const;
   void push(const T& newItem);
   T& pop();
   T& peek() const;

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

An Implementation That Uses the ADT List

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

- **isEmpty():** return list.isEmpty()
- **push(x):** list.add(0, x)
- **pop():** list.remove()
- **peek():** list.get(0)

Comparing Implementations

- **Array based:**
  - Fixed size (cannot grow and shrink dynamically)
- **Using a dynamic array:**
  - 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, but still O(1)
- **Using the LinkedList class:**
  - Reuses existing implementation
  - Reduces the coding effort but may be a bit less efficient

Stack exercises – Q1

```
s.push("A"); s.push("B"); s.pop();
s.push("C"); s.push("D"); s.peek();
s.pop();     s.push("E"); s.push("F"):
s.pop();     cout << s.peek());
```

Suppose I create a new stack s with **capacity 3**, and run the instructions above. What is the output?

a) E
b) F
c) Nothing, there is StackException: stack full on push
d) Nothing, there is StackException: stack empty on pop

Stack exercises – Q2

```
s.push("A"); s.push("B"); s.pop();
s.push("C"); s.push("D"); s.peek();
s.pop();     s.push("E"); s.push("F"):
s.pop();     cout << s.peek());
```

Suppose I create a new stack s with **unbounded capacity**, and run the instructions above. What is the output?

a) E
b) F
c) Nothing, there is StackException: stack full on push
d) Nothing, there is StackException: stack empty on pop

Stack applications- Delimeter Matching

- You want to write a program to check the parentheses in a math expression are balanced:

  $$(w * (x + y) / z – (p / (r – q)))$$

- It may have several different types of delimiters: **braces{}, brackets[], parentheses()**

- Each opening (left) delimiter must be matched by a closing (right) delimiter.
- A delimiter that opens the last must be closed by a matching delimiter first. For example, [a * (b + c] + d) is wrong!

Stack applications- Delimeter Matching

- Read characters one-by-one from the expression.
- Whenever you see a **left** (opening) delimiter, **push** it to stack.
- Whenever you see a **right** (closing) delimiter, **pop** from stack and check **match** (i.e. same type?)
- If they don't match, report mismatch error.

- What happens if the stack is **empty** when you try to match a closing delimiter?
- What happens if the the stack is **non-empty** after you reach to the end of the expression?

Stack Exercise –Q3

- What happens when using the delimiter matching algorithm to parse the following expression (assume the stack is unbounded):

a { b [c] d } (e) f }

a) There is no error: the expression is valid
b) Stack is non-empty after it's done parsing
c) Delimeter mismatch error
d) Stack full exception
e) Stack empty exception

Stack application: Evaluation of Postfix expressions

- A *postfix expression* is a mathematical expression but with the operators written after the operands rather than before.

  `1 + 1` becomes `1 1 +`

  `1 + 2 * 3 + 4` becomes `1 2 3 * + 4 +`

  – supported by many kinds of fancy calculators
  – never need to use parentheses
  – never need to use an = character to evaluate on a calculator

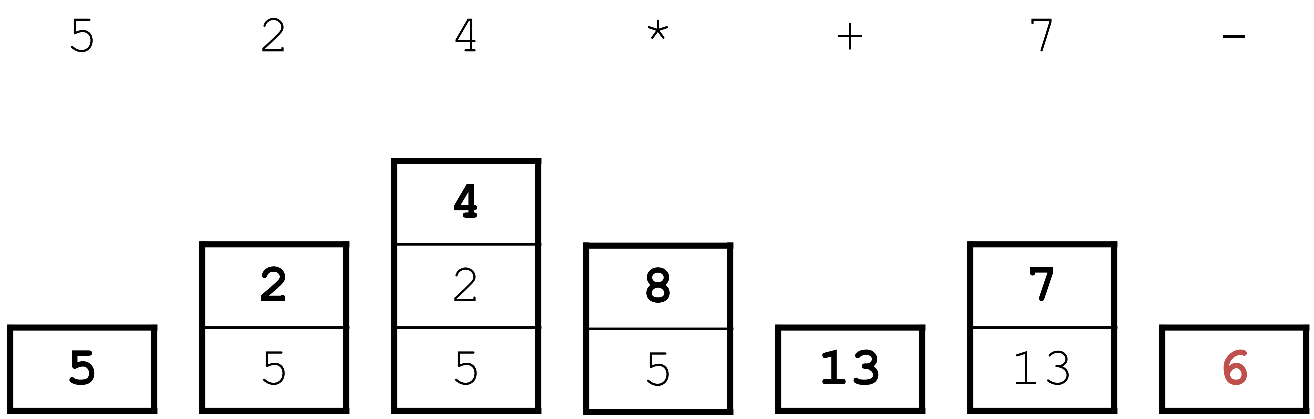
- Write a method `postfixEvaluate` that accepts a postfix expression string, evaluates it, and returns the result.
  – All operands are integers; legal operators are `+` , `-`, `*`, and `/`

  `postFixEvaluate("5 2 4 * + 7 -")` returns `6`

Postfix algorithm

- The algorithm: Use a **stack**
  - When you see an operand, push it onto the stack.
  - When you see an operator:
    - pop the last two operands off of the stack.
    - apply the operator to them.
    - push the result onto the stack.
  - When you're done, the one remaining stack element is the result.

`"5 2 4 * + 7 -"`



Exercise solution

```
// Evaluates the given prefix expression and returns its result.
// Precondition: string represents a legal postfix expression

int postfixEvaluate(string expression) {
    Stack<int> s ;
    istringstream line(expression);
    string token;
    while (line >> token) {
        if (checkIfNumber(token)) {     // an operand (integer)
            s.push(stoi(token));
        } else {                        // an operator
            int operand2 = s.pop();
            int operand1 = s.pop();
            if (token== "+") {
                s.push(operand1 + operand2);
            } else if (token== "-") {
                s.push(operand1 - operand2);
            } else if (token=="*") {
                s.push(operand1 * operand2);
            } else {
                s.push(operand1 / operand2);
            }
        }
    }
    return s.pop();
}
```

Stack Application: Infix to Postfix

- 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

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

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

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:       // process stack operators of greater precedence
      while (!aStack.isEmpty() and top of stack is not '(' and
                        precedence(ch) <= precedence(top of stack) ) {
           postfixExpr=postfixExpr+(top of stack);
           aStack(pop);
       }
       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);
}
```

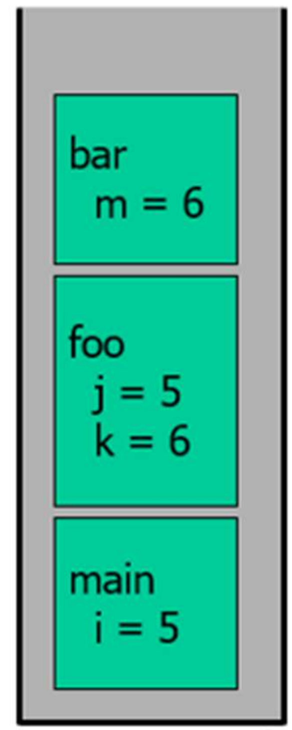
Relationship Between Stacks and Recursion

- There is a strong relationship 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
- Stacks can be used to implement a non-recursive version of a recursive algorithm

Run-time Stack

- The 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 an activation record containing local variables and return value
- When a function returns, its activation record 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

Tracing a Recursive Function

- Consider the recursive factorial function

```
int fact(int n)
{
  if (n ==0)
     // base case
      return (1);
  else
     // recursive step
     return (n * fact(n-1));
}
```

- What is the result of the following call?

```
fact(3);
```

A Sequence of Computations of Factorial Function

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

cout << fact (3);

6

fact(3):

return (3 * fact(2));

3 * 2

fact(2):

return (2 * fact(1));

2 * 1

fact(1):

return (1 * fact(0));

1 * 1

fact(0):

return 1

Example 2: Palindrome

- Write a recursive function `isPalindrome` accepts a `String` and returns `true` if it reads the same forwards as backwards.
  - `isPalindrome("madam")` → `true`
  - `isPalindrome("racecar")` → `true`
  - `isPalindrome("step on no pets")` → `true`
  - `isPalindrome("able was I ere I saw elba")` → `true`
  - `isPalindrome("Java")` → `false`
  - `isPalindrome("rotater")` → `false`
  - `isPalindrome("byebye")` → `false`
  - `isPalindrome("notion")` → `false`

isPalindrome

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
bool isPalindrome(string s) {
    if (s.length() < 2) {
        return true;   // base case
    } else {
        char first = s[0];
        char last  = s[s.length() - 1];
        if (first != last) {
            return false;
        }              // recursive case
        string middle = s.substr (1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

Trace of `isPalindrome(0,3,str)`

`is_palindrome("abba");`

```
bool isPalindrome(string s)
if (s.length() < 2)
  return true;
else {
  char first = s[0];
  char last  = s[s.length() - 1];
  if (first != last) return false;
  string middle=s.substr(1,s.length()-2);
  return isPalindrome(middle);
}
```

true

s = “abba”
s.length <2: false
first = ‘a’; last = ‘a’
first!=last: false
middle =“bb”
return(true
<u>is_palindrome(middle)</u>)

true

s = “bb”
s.length <2: false
first = ‘b’; last = ‘b’
first!=last: false
middle =“”
return(true
<u>is_palindrome(middle)</u>)

s= “”
s.length <2: true
return true

true

Comparison of Iteration and Recursion

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a function is avoided in iterative version.

- However a recursive solution can be sometimes the most natural and logical way of solving a problem.

- Conflict: machine efficiency versus programmer efficiency