CSC 222: Computer Programming II

Spring 2004

Stacks and recursion

- stack ADT
  - push, pop, top, empty, size
- vector-based implementation, <stack> library
- application: parenthesis/delimiter matching
- run-time stack

Lists & stacks

list ADT

DATA: sequence of items
OPERATIONS: add item, look up item, delete item, check if empty, get size, …

e.g., array, vector, DeckOfCards, CaveMaze, List, SortedList

stack ADT

- a stack is a special kind of (simplified) list
- can only add/delete/look at one end (commonly referred to as the top)

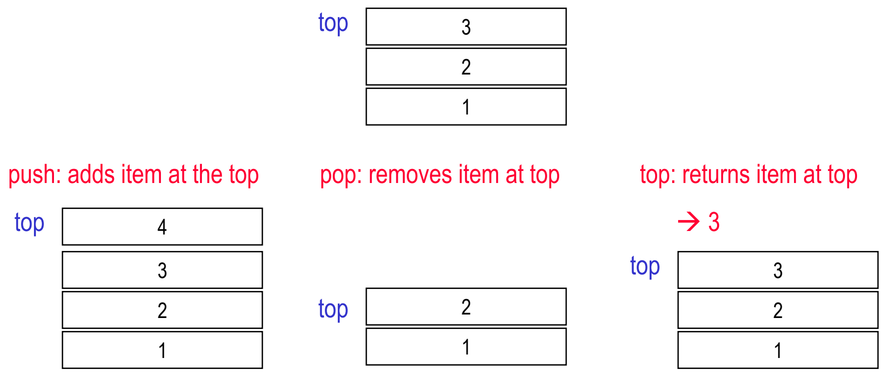
DATA: sequence of items
OPERATIONS: push on top, peek at top, pop off top, check if empty, get size

these are the ONLY operations allowed on a stack
— stacks are useful because they are simple, easy to understand
— each operation is O(1)

Stack examples

- PEZ dispenser
- pile of cards
- cars in a driveway
- function activation records (later)

a stack is also known as
- push-down list
- last-in-first-out (LIFO) list



Stack exercise

- start with empty stack
- PUSH 1
- PUSH 2
- PUSH 3
- TOP
- PUSH 4
- POP
- POP
- TOP
- PUSH 5

stack implementation

recall that the vector class provides member functions for all of these

- push_back
- pop_back
- back
- size

we could simply use a vector whenever we want stack behavior

better yet, define a stack class in terms of vector

```
template <class Type>
class stack
{
  public:
    stack() { /* does nothing */ }

    void push(const Type & item)
    {
          elements.push_back(item);
    }

    void pop()
    {
          elements.pop_back();
    }

    Type top() & const
    {
          return elements.back();
    }

    bool empty() const
    {
          return (elements.size() == 0);
    }

    int size() const
    {
          return elements.size();
    }

  private:
    vector<Type> elements;
};
```


In-class exercise

what does this code do?

create a C++ project

- copy stack.h and pushy.cpp and add to the project
- test it

now what?

```
stack<int> numStack;

int num;
while (cin >> num) {
    numStack.push(num);
}

while ( !numStack.empty() ) {
    cout << numStack.top() << endl;
    numStack.pop();
}
```

```
stack<int> numStack1, numStack2;

int num;
while (cin >> num) {
    numStack1.push(num);
}

while ( !numStack1.empty() ) {
    numStack2.push(numStack1.top());
    numStack1.pop();
}

while ( !numStack2.empty() ) {
    cout << numStack2.top() << endl;
    numStack2.pop();
}
```

<stack> class

since a stack is a common data structure, a predefined C++ library exists

```
#include <stack>
```

the standard `stack` class has all the same member functions as our implementation

```
void push(const TYPE & item);      // adds item to top of stack
void pop();                        // removes item at top of stack
TYPE & top() const;                // returns item at top of stack
bool empty() const;                // returns true if stack is empty
int size() const;                  // returns size of stack
```

replace "stack.h" with <stack> in your program and verify it works

Stack application

consider mathematical expressions such as the following

- a compiler must verify such expressions are of the correct form

(A * (B + C))   ((A * (B + C)) + (D * E))

how do you make sure that parentheses match?

common first answer:

- count number of left and right parentheses
- expression is OK if and only if # left = # right

(A * B) +)C(

more subtle but correct answer:

- traverse expression from left to right
- keep track of # of unmatched left parentheses
- if count never becomes negative and ends at 0, then OK

Parenthesis matching

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string expression;
    cout << "Enter the expression to check: ";
    getline(cin, expression);

    int openCount = 0;
    for (int i = 0; i < expression.length(); i++) {
        if (expression[i] == '(') {
            openCount++;
        }
        else if (expression[i] == ')') {
            openCount--;
            if (openCount < 0) {
                cout << "INVALID: unmatched ')'" << endl;
                exit(1);
            }
        }
    }

    if (openCount == 0) {
        cout << "VALID expression" << endl;
    }
    else {
        cout << "INVALID: unmatched '('" << endl;
    }

    return 0;
}
```

`openCount` keeps track of unmatched left parens

as the code traverses the string, the counter is

- incremented on '('
- decremented on ')'

`openCount` must stay non-negative and end at 0

Delimiter matching

now, let's generalize to multiple types of delimiters

(A * [B + C])   {(A * [B + C]) + [D * E]}

does a single counter work?

how about separate counters for each type of delimiter?

recursive solution:

- traverse the expression from left to right
- if you find a left delimiter,
  - recursively traverse until find the matching delimiter

stack-based solution:

- start with an empty stack of characters
- traverse the expression from left to right
  - if next character is a left delimiter, push onto the stack
  - if next character is a right delimiter, must match the top of the stack

Delimiter matching

```
int main()
{
    string expression;
    cout << "Enter the expression to check: ";
    getline(cin, expression);

    stack<char> delimiters;

    for (int i = 0; i < expression.length(); i++) {
        if (IsLeftDelimiter(expression[i])) {
            delimiters.push(expression[i]);
        }
        else if (IsRightDelimiter(expression[i])) {
            if (!delimiters.empty() &&
                delimiters.top() == MatchingDelimiter(expression[i])) {
                 delimiters.pop();
            }
            else {
                  cout << "INVALID: unmatched " << expression[i] << endl;
                  exit(1);
            }
        }
    }

    if (delimiters.empty()) {
          cout << "VALID expression" << endl;
    }
    else {
          cout << "INVALID: unmatched" << delimters.top() << endl;
    }

    return 0;
}
```

here, defined abstract functions for categorizing delimiters

note natural correspondence to the simpler version

Delimiter matching (cont.)

```
const string LEFT = "([{";
const string RIGHT = ")]}";

. . .

bool IsLeftDelimiter(char ch)
{
    return (LEFT.find(ch) != string::npos);
}

bool IsRightDelimiter(char ch)
{
    return (RIGHT.find(ch) != string::npos);
}

char MatchingDelimiter(char ch)
{
    if (IsLeftDelimiter(ch)) {
          int index = LEFT.find(ch);
          return RIGHT[index];
    }
    else if (IsRightDelimiter(ch)) {
          int index = RIGHT.find(ch);
          return LEFT[index];
    }
    else {
          return '?';
    }
}
```

to avoid global modifications whenever a new pair of delimiters is added:

- use global strings to store delimiters of each type (keep data in parallel)
- to see if a char is a delimiter, search these strings using find
- to get the matching delimiter, find the char then access the char in the other parallel string

Reverse Polish

evaluating Reverse Polish (postfix) expressions

- note: if entering expressions into a calculator in postfix, don't need parens
- this format was used by early HP calculators (& some models still allow the option)

```
1 2 +           → 1 + 2
1 2 + 3 *       → (1 + 2) * 3
1 2 3 * +       → 1 + (2 * 3)
```

to evaluate a Reverse Polish expression:

- start with an empty stack that can store numbers
- traverse the expression from left to right
- if next char is an operand (number or variable), push on the stack
- if next char is an operator (+, -, *, /, …),
  1. pop 2 operands off the top of the stack
  2. apply the operator to the operands
  3. push the result back onto the stack
- when done, the value of the expression is on top of the stack

Run-time stack

when a function is called in C++ (or any language):

- suspend the current execution sequence
- allocate space for parameters, locals, return value, …
- transfer control to the new function

when the function terminates:

- deallocate parameters, locals, …
- transfer control back to the calling point (& possibly return a value)

note: functions are LIFO entities

- `main` is called first, terminates last
- if main calls `Foo` and `Foo` calls `Bar`, then
  `Bar` terminates before `Foo` which terminates before `main`

→ a stack is a natural data structure for storing information about function calls and the state of the execution

Run-time stack (cont.)

an activation record stores info (parameters, locals, …) for each invocation of a function

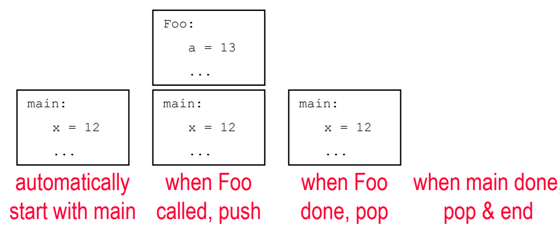
- when the function is called, an activation record is pushed onto the stack
- when the function terminates, its activation record is popped
- note that the currently executing function is always at the top of the stack

```
void Foo(int a)
{
    a++;
    cout << "Foo " << a << endl;
}

int main()
{
    int x = 12;

    Foo(x);
    cout << "main " << x << endl;

    return 0;
}
```



Another example

```
void Bar(int z)
{
    z--;
    cout << "Bar " << z << end;'
}

void Foo(int a)
{
    a++;
    cout << "Foo " << a << endl;
    Bar(a + 10);
}

int main()
{
    int x = 12;

    Foo(x);
    cout << "main1 " << x << endl;

    Bar(x);
    cout << "main2 " << x << endl;

    return 0;
}
```

run time stack behavior?

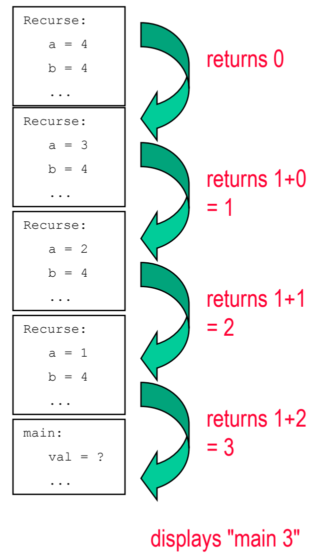
Recursive example

```
void Recurse(int a, int b)
{
    if (a >= b) {
        return 0;
    }
    else {
        return 1 + Recurse(a+1, b);
    }
}

int main()
{
    int val = Recurse 1, 4);

    cout << "main " << val << endl;

    return 0;
}
```



recursive functions are treated just like any other functions

- when a recursive call is made, an activation record for the new instance is pushed on the stack
- when terminates (i.e., BASE CASE), pop off activation record & return value

Programming language implementation

note: function calls are not the only predictable (LIFO) type of memory

- blocks behave like unnamed functions, each is its own environment

```
for (int i = 0; i < 10; i++) {
    int sum = 0;
    if (i % 3 == 0) {
        int x = i*i*i;
        sum = sum + x;
    }
}
```

even within a function or block, variables can be treated as LIFO

```
int x;
. . .
int y;
```

for most programming languages, predictable (LIFO) memory is allocated/deallocated/accessed on a run-time stack

In-class exercise

copy the following files from the ~davereed/csc222/Code directory:

mergeDemo.cpp Sorts.h Die.h Die.cpp

build a project and test merge sort on various list sizes

to visualize the run-time stack:

- modify the recursive MergeSort function so that it prints a message at the beginning and end (specifying the range being sorted)

  PUSH merge 0 9
  .
  .
  .
  POP merge 0 9

- add a GLOBAL! counter to Sorts.h and keep track of the number of recursive calls (& display along with PUSH message)