

Chapter 10

Stacks & Queues

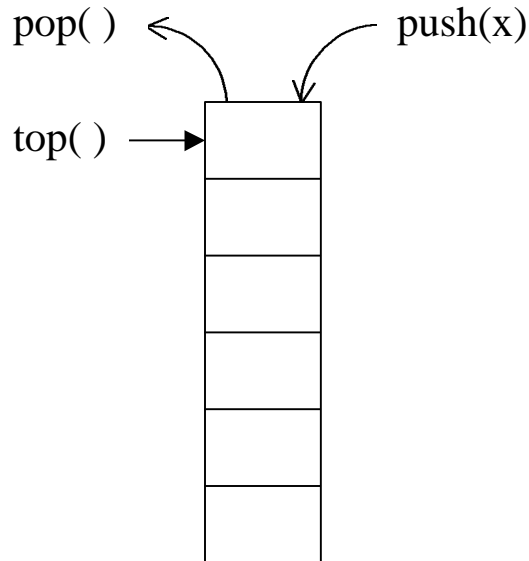
Data Structures in C++: Using the STL

Timothy Budd

- Stack data abstraction
- Adaptors
- Reverse Polish Notation
- Converting infix to postfix
- Queues in simulation
- Ring Buffers
- BFS and DFS

Stacks

- LIFO – Last in, First out



- Run-time stack used on every function call
 - Activation Record (Stack Frame)
 - Recursion requires a stack
- stack is an Adaptor class
 - Template argument is a container
 - Any container defining:

value_type	back()
push_back()	empty()
pop_back()	size()

can be a stack!

The Stack Adaptor

```
template <class Container>
class stack
{
public:
    typedef Container::value_type value_type;

    // operations
    bool    empty() { return c.empty; }
    int     size()  { return c.size(); }
    value_type& top(){ return c.back(); }
    void    push(value_type& x)
            { c.push_back(x); }
    void    pop()   { c.pop_back(); }

protected:
    Container c;
};

// Use of the stack adaptor:

stack< vector<int> >      stack_one;
stack< list<double> >   stack_two;
stack< deque<string> >  stack_three;

// The above Containers support all 6 required
// member functions on the previous page.
```

Which Container is Appropriate?

- `string` use is obvious!
- `vectors` and `deques` have capacity \geq size
 - efficient if container doesn't grow beyond its original size
 - inefficient if `realloc()`'s must be done
- `lists` are time and space efficient if size grows and shrinks during lifetime of list
 - objects, once allocated, never move
 - inserting/deleting is efficient at head and tail of list
 - inserting/deleting in middle requires a search loop to find the item in question
 - cost of this flexibility is the space required, at each node, to hold pointers to the previous and next node

Stack Application: RPN Calculator

- RPN (Reverse Polish Notation)

- post-fix rather than in-fix:

- $a+b*c-d$ becomes $a\ b\ c\ *\ +\ d\ -$

- $(a+b)*(c-d)$ is $a\ b\ +\ c\ d\ -\ *$

- No parentheses are needed in post-fix

- HP calculators vs. TI calculators

- Problem:

- Evaluate any RPN expression

- Solution:

- Push operands on stack

- When operator is seen:

- pop two top operands off stack

- perform: operand1 op operand2

- push result on top of stack

```

//      test of rpn calculator routine
//
//      Described in Chapter 10 of
//      Data Structures in C++ using the STL
//      Published by Addison-Wesley, 1997
//      Written by Tim Budd, budd@cs.orst.edu
//      Oregon State University
//

# include <list.h>
# include <stack.h>

//
//      class calculatorEngine
//      simulate the behavior of a simple integer calculator
//

class calculatorEngine
{
public:
    enum    binaryOperator {plus, minus, multiply, divide};

    int     currentMemory    () { return data.top (); }
    void    pushOperand      (int value) { data.push (value); }
    void    doOperator       (binaryOperator theOp);

protected:
    stack < list<int> > data;
};

void calculatorEngine::doOperator(binaryOperator theOp)
    // perform a binary operation on stack values
{
    int right = data.top();
    data.pop();
    int left = data.top();
    data.pop();
    int result;
    switch(theOp)    // do the operation
    {
        case plus:
            result = left + right;
            break;
        case minus:
            result = left - right;
            break;
        case multiply:
            result = left * right;
            break;
        case divide:
            result = left / right;
            break;
    }

    // push the result back on the stack
    data.push(result);
}

```

```

void calculator()
{
    int intval;
    calculatorEngine calc;
    char c;

    while (cin >> c)
    {
        switch(c)
        {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                cin.putback(c);
                cin >> intval;
                calc.pushOperand(intval);
                break;

            case '+':
                calc.doOperator(calculatorEngine::plus);
                break;

            case '-':
                calc.doOperator(calculatorEngine::minus);
                break;

            case '*':
                calc.doOperator(calculatorEngine::multiply);
                break;

            case '/':
                calc.doOperator(calculatorEngine::divide);
                break;

            case 'p':
                cout << calc.currentMemory() << '\n';
                break;

            case 'q':
                return; // quit calculator
        }
    }
}

void main () { calculator(); }

```

Convert In-fix to Post-fix

Uses the following stack rules:

- Operands are passed thru. Only operators are stacked.
- Given typical precedence of operators:
 - + , - level 1 (i.e. lower)
 - * , / level 2 (i.e. higher)
- Only stack higher precedence operators on top of lower ones. (exception: '('s are always stacked and have prec. level 0.)
- Pop off ops until a lower precedence op is revealed or stack is empty. (exception: When seeing ')' pop off all ops until a '(' is encountered. Consume both parens.)

```

//
// test of infix to Postfix routine
//
// Described in Chapter 10 of
// Data Structures in C++ using the STL
// Published by Addison-Wesley, 1997
// Written by Tim Budd, budd@cs.orst.edu
// Oregon State University
//

# include <string>
# include <list.h>
# include <stack.h>

// operators listed in precedence order
enum operators { leftparen, plusOp, minusOp, multiplyOp, divideOp };

string opString (operators theOp)
// return a textual representation of an operator
{
    switch (theOp)
    {
        case plusOp:   return " + ";
        case minusOp:  return " - ";
        case multiplyOp: return " * ";
        case divideOp: return " / ";
    }
}

void processOp
(operators theOp, stack<list<operators> > & opStack,
 string & result)
{
    // pop stack while operators have higher precedence
    while ((! opStack.empty()) && (theOp < opStack.top()))
    {
        result += opString(opStack.top());
        opStack.pop();
    }
    // then push current operator
    opStack.push(theOp);
}

string infixToPostfix(string infixStr)
{
    stack < list<operators> > opStack;
    string result("");
    int i = 0;

    while (infixStr[i] != '\0')
    {
        if (isdigit(infixStr[i])) // process constants
        {
            while (isdigit(infixStr[i]))
                result += infixStr[i++];
            result += " "; // add separator
        }
        else
            switch(infixStr[i++]) // process other characters
            {
                case '(':
                    opStack.push(leftparen);
                    break;
            }
    }
}

```

```

        case ')':
            while (opStack.top() != leftparen)
            {
                result += opString(opStack.top());
                opStack.pop();
            }
            opStack.pop();    // pop off left paren
            break;
        case '+': processOp(plusOp, opStack, result);
            break;
        case '-': processOp(minusOp, opStack, result);
            break;
            case '*': processOp(multiplyOp, opStack,
                result);
                break;
            case '/': processOp(divideOp, opStack, result);
                break;
    } // switch
} // while
while (! opStack.empty())    // empty the stack on end of input
{
    result += opString(opStack.top());
    opStack.pop();
}

return result;    // return result string
}

```

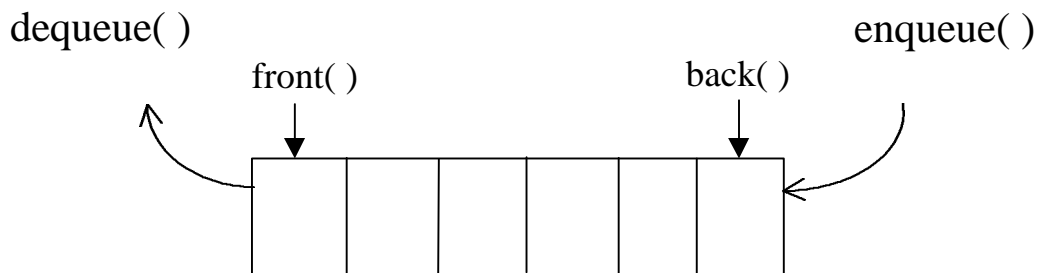
```

void main ()
{
    string input = "5 * (27 + 3 * 7) + 22";
    cout << infixToPostfix(input) << "\n";
}

```

Queues

- FIFO – First in, First out



- Heavily used in simulations
 - Queueing Models (Networks)
 - Operating System Services
- `queue` is an Adaptor class
 - Template argument is a container
 - Any container defining:

<code>front()</code>	<code>back()</code>
<code>push_back()</code>	<code>empty()</code>
<code>pop_front()</code>	<code>size()</code>

can be a queue!

The Queue Adaptor

```
template <class Container>
class queue
{
public:
    typedef Container::value_type value_type;

    // operations
    bool      empty() { return c.empty; }
    int       size()  { return c.size(); }
    value_type& front() { return c.front(); }
    value_type& back() { return c.back(); }
    void      enqueue(value_type& x)
              { c.push_back(x); }
    void      dequeue() { c.pop_front(); }

protected:
    Container c;
};

// Use of the stack adaptor:

queue< list<double> >      queue_one;
queue< deque<string> >    queue_two;

// Notice that vectors cannot be used!
// Vectors don't support pop_front();
```

```

//    bank teller simulation routine
//
//    Described in Chapter 10 of
//    Data Structures in C++ using the STL
//    Published by Addison-Wesley, 1997
//    Written by Tim Budd, budd@cs.orst.edu
//    Oregon State University
//

# include <list.h>
# include <vector.h>
# include <stack.h>
# include <string>

//
//    class Customer
//        a single customer waiting in the bank teller line

class randomInteger
{
public:
    unsigned int operator () (unsigned int);
};

unsigned int randomInteger::operator () (unsigned int max)
{
    // rand return random integer
    // convert to unsigned to make positive
    // take remainder to put in range
    unsigned int rval = rand();
    return rval % max;
}

randomInteger randomizer;

class Customer
{
public:
    // constructors
    Customer (int at) : arrivalTime(at),
                     processtime(2 + randomizer(6)) {}
    Customer () : arrivalTime(0), processtime(0) { }

    // operations
    bool done    () { return --processtime < 0; }
    int  arrival () { return arrivalTime; }

    operator < (Customer & c)    // order by arrival time
    { return arrivalTime < c.arrivalTime; }

    operator == (Customer & c)  // no two customers are alike
    { return false; }

protected:
    unsigned int arrivalTime;
    unsigned int processtime;
};

```

```

//
// class Teller
// a teller servicing customers in a bank teller line

class Teller
{
public:
    Teller() { free = true; }

    bool isFree() // see if teller is free to work
    {
        if (free) return true;
        if (customer.done())
            free = true;
        return free;
    }

    void addCustomer(Customer & c) // start servicing customer
    {
        customer = c;
        free = false;
    }

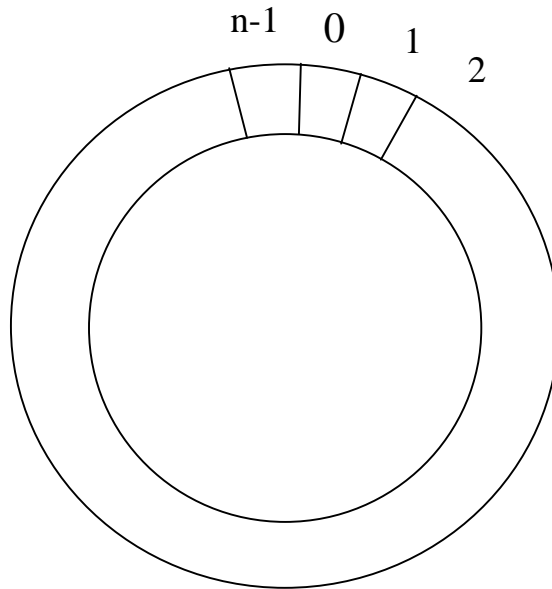
protected:
    bool free;
    Customer customer;
};

void main()
{
    int numberOfTellers = 5;
    int numberOfMinutes = 60;
    double totalWait = 0;
    int numberOfCustomers = 0;
    vector < Teller > teller(numberOfTellers);
    queue < list< Customer > > line;

    for (int time = 0; time < numberOfMinutes; time++)
    {
        if (randomizer(10) < 9)
        {
            Customer newCustomer(time);
            line.push(newCustomer);
        }
        for (int i = 0; i < numberOfTellers; i++)
        {
            if (teller[i].isFree() & ! line.empty())
            {
                Customer frontCustomer = line.front();
                numberOfCustomers++;
                totalWait += (time - frontCustomer.arrival());
                teller[i].addCustomer(frontCustomer);
                line.pop();
            }
        }
    }
    cout << "average wait:" << (totalWait / numberOfCustomers)
    << endl;
}

```

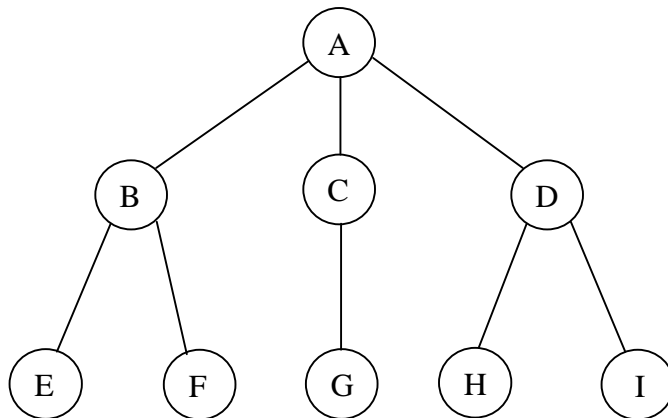
Ring Buffers (Circular Queues)



- vectors & arrays work well
 - use modulo (%) division
 - $v[i \% n]$ handles $i < 0$ and $i > n$
- Obviously, never move items in the queue, just move the `front` and `rear` indices.
- Can also use linked lists (textbook)

Search

Given a tree, search every node:



Traversal can be done in two ways:

Breadth First Search

Traverses L to R across a row before descending to the next level.

Depth First Search

Traverses L to R going down to the next level before going across to the next sibling.

Breadth First Search

Uses a Queue to store nodes:

1. Enqueue the root node on the queue.
2. Dequeue the first node n from the queue.
3. Enqueue the children of n into the queue in left-to-right order.
4. Repeat steps 2 and 3 until goal is found or queue is empty.

Depth First Search

Uses a Stack to store nodes:

1. Push root node of tree on the stack.
2. Pop the top node off the stack. Call this node n .
3. Push children of node n onto the stack in right-to-left order (so they are popped off in left-to-right order).
4. Repeat steps 2 and 3 until stack is empty or goal is reached.