Class Objective/Overview

• Understand *Stacks*

• Understand *C++ STL stack Container*

• Familiarize with *Stack Applications*

• Familiarize with *The miniStack Class*

• Understand *Queues*

• Understand *C++ STL queue Container*

• Familiarize with *The miniQueue Class*
Stack ADT
Stacks

- Like vectors and lists, stack is also an ordered collection, but the construction and access rules are purposely limited.

- Stacks organize the data in a Last-In, First-Out (LIFO) manner. We can think of a stack as a sequence of elements where we always add to and remove from the same end.
  - Data is pushed (inserted) onto a stack or popped (erased) off of a stack.
  - operations occur at one end, called the top of the stack
  - We can only examine data on the top of the stack.
Stacks

- Stacks can be easily implemented using array/vector-like data structures or via linked lists. You can see both possibilities below.

- Actually, implementation of a stack is pretty trivial using either the std::vector<T> or std::list<T> types as an underlying implementation.
  
  - To push onto the stack, do a push_back onto the underlying vector or list
  
  - To pop from the stack, do a pop_back on the underlying vector or list
  
  - To access the top of the stack, get the back() of the vector or list.
STL Stack Container in C++
std::stack - Operations

stack(); Create an empty stack. This is the default constructor.

bool empty() const;
Check whether the stack is empty. Return true if it is empty and false otherwise.

void pop();
Remove the item from the top of the stack.
Precondition: The stack is not empty.
Postcondition: Either the stack is empty or the stack has a new topmost item from a previous push.

void push(const T &item);
Insert the argument item at the top of the stack.
Postcondition: The stack has a new item at the top.
std::stack - Operations

int size() const;
Return the number of items on the stack.

T& top() const;
Return a reference to the value of the item at the top of the stack.
Precondition: The stack is not empty.

const T& top() const;
Constant version of top().
Using Stacks

• Simply, stack object is instantiated with specifying the type of stack elements:
  • stack <string> strStk

• Also, when instantiating the stack, we could specify both:
  • The type of elements to go on the stack:
    stack <string, vector<string> >
  • The sequence to be used as the implementing data structure:
    stack<string, vector<string> >

• By the way, be careful with things like stack<string, vector<string> >. The blank between the two “ > ” is important. Without that, C++ assumes that you are writing the `»` operator (as in cin » n)
Stack Applications
Stack Applications

- Stacks are among the most common data structures and crop up in a variety of applications.
- They are especially useful in parsing and translation of computer languages, mathematics, and other formal notations.
  - Parentheses matching
  - Postfix translation and evaluation
  - Compilers
- One of common application of stacks is in converting recursive algorithms to iterative.
Using Stacks for Hex Conversion

Conversion of a decimal number to Hex string:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'F'</td>
<td>15</td>
</tr>
<tr>
<td>'F'</td>
<td>10</td>
</tr>
<tr>
<td>'A'</td>
<td>10</td>
</tr>
<tr>
<td>'F'</td>
<td>15</td>
</tr>
</tbody>
</table>

431 % 16 = 15
431 / 16 = 26
26 % 16 = 10
26 / 16 = 1
1 % 16 = 1
1 / 16 = 0

Push Digit Characters

Stack <char>

numStr = "";
string hex="0123456789ABCDEF"

while (num != 0){
    push( hex[num%16] ) ;
    num = num / 16;
}

while (!stk.empty()){
    numStr = numStr + stk.top () ;
    stk.pop();
}

//numStr is the Hex string of num
cout << numStr << endl;
Using Stacks in Function Calls

- We call the collection of information that represents a function call in progress an activation of the function.

- Computer systems use a runtime stack (called the activation stack) to keep track of the return addresses, actual parameters, and local variables associated with function calls.

- Each function call results in pushing an activation record containing that information onto the stack.

- Returning from function is accomplished by getting and saving the return address out of the top record on the stack, popping the stack once, and jumping to the saved address.
Using Stacks in Function Calls

• Activation stack is very helpful in executing recursive functions.

• Computers really don’t do recursion. Instead, a recursive algorithm really gets translated as a \textit{series of stack pushes} followed by a \textit{jump back} to the beginning of the recursive function.

• Using a stack makes it easy to “backtrack” to the proper location when a function call returns.

• \textit{Recursive functions} require large activation stack memory size. Systems with limited activation stack size (e.g., Windows 3.1 system) recursive functions may overflow the system.

• A similar problem can occur when programming for \textit{embedded systems}. Such systems generally have very limited memory overall, including very limited activation stacks.
Using Stacks in Function Calls

Example: Calculation of 3!

```c
int fact(int n)
{
    if (n<=1) return 1;
    else
    return n * fact(n-1);
}
void main()
{
    int value;
    value = fact(3);
}
```

Stack contents during each recursive call

ARI = activation record instance
Example (cont’d):
Calculation of 3!

```c
int fact(int n)
{
    if (n<=1) return 1;
    else
        return n * fact(n-1);
}
void main()
{
    int value;
    value = fact(3);
}
```

Stack contents during each `return`

ARI = activation record instance
Using Stacks in Expression Evaluation

• **Postfix** (also known as Reverse Polish Notation or RPN) is a parentheses-free notation for mathematical expressions:

  • Operators in postfix appear after their operands. Examples of postfix expression versus the conventional **infix** notation:
    • 1 2 + instead of 1+2
    • 1 2 3 * + instead of 1+2*3
    • 1 2 + 3 * instead of (1+2)*3

• **Infix** notation
  • A binary operator appears between its operands (e.g., “1+2”).
  • More complex than postfix, because it requires the use of operator precedence and parentheses (e.g., “(1+2)*3”).
  • In addition, some operators are left-associative, and a few are right-associative (e.g., “-3”).
Using Stacks in Expression Evaluation

• Postfix is easy to evaluate using a single stack to hold operands:

  • When you see a constant, push it onto the stack
  • When you see an operator that needs k operands,
    • pop k numbers from the stack
    • apply the operator to them
    • push the result back onto the stack
  • At the end a single value remains on the stack. This is the value of the expression.
Using Stacks in Expression Evaluation

Postfix Evaluation Example: expression $2 \ 3 \ +$

<table>
<thead>
<tr>
<th>Scan of Expression and Action</th>
<th>Current operandStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify 2 as an operand. Push integer 2 on the stack.</td>
<td>2</td>
</tr>
<tr>
<td>2. Identify 3 as an operand. Push integer 3 on the stack.</td>
<td>3 2</td>
</tr>
<tr>
<td>3. Identify + as an operator Begin the process of evaluating +.</td>
<td>3 2</td>
</tr>
<tr>
<td>4. getOperands() pops stack twice and assigns 3 to right and 2 to left.</td>
<td>operandStack empty</td>
</tr>
<tr>
<td>5. compute() evaluates left + right and returns the value 5. Return value is pushed on the stack.</td>
<td>5</td>
</tr>
</tbody>
</table>
Using Stacks in Expression Evaluation

• Conversion to postfix is generally performed via a stack. For example, you can use this pseudo code to convert a "normal" infix algebraic expression without parentheses into postfix.

• Try this with an expression like "1 + 2 * 3 + 4".
  • The output will be: 
    "1 2 3 * + 4 +"

• This algorithm can be modified, without too much trouble, to work with parentheses as well.

```
//Expression with no parentheses
S = an empty stack ;
while (more input available ) {
    read the next token , T;
    if T is a variable name or a number
        print it ;
    else { // T is an operator
        while (S is not empty and T has lower precedence than top operator on S) {
            print top operator on S ;
            pop S ;
        }
        push T onto S
    }
}
while (S is not empty) {
    print top operator on S ;
    pop S ;
}
```
infix2Postfix Class

#ifndef INFIX_TO_POSTFIX
#define INFIX_TO_POSTFIX

#include <iostream>
#include <string>
#include <stack>
#include <ctype.h>
#include "d_expsym.h" // expressionSymbol class
#include "d_except.h" // for expressionError exception

using namespace std;

// labels designating the parentheses characters
const char lParen = '(', rParen = ')';

class infix2Postfix
{
    public:
        infix2Postfix();
            // default constructor. infix expression is NULL string
        infix2Postfix(const string& infixExp);
            // initialize the infix expression
}
infix2Postfix Class

void setInfixExp(const string& infixExp);
   // change the infix expression

string postfix();
   // return a string that contains the equivalent postfix
   // expression. the function throws expressionError if an
   // error occurs during conversion

private:

string infixExpression;
   // the infix expression to convert

string postfixExpression;
   // built to contain the postfix equivalent of infixExpression

stack<expressionSymbol> operatorStack;
   // stack of expressionSymbol objects

void outputHigherOrEqual(const expressionSymbol& op);
   // the expressionSymbol object op holds the current
   // symbol. pop the stack and output as long as the symbol
   // on the top of the stack has a precedence >= that of
   // the current operator

bool isOperator(char ch) const;
   // is ch one of '+','-','*','/','%','^'
### infix2Postfix Class

```cpp
void infix2Postfix::outputHigherOrEqual(const expressionSymbol& op) {
    expressionSymbol op2;
    while(!operatorStack.empty() && (op2 = operatorStack.top()) >= op) {
        operatorStack.pop();
        postfixExpression += op2.getOp();
        postfixExpression += ' ';
    }
}

bool infix2Postfix::isOperator(char ch) const {
    return ch == '+' || ch == '-' || ch == '*' || ch == '%' || ch == '/' || ch == '^';
}

infix2Postfix::infix2Postfix() {}

infix2Postfix::infix2Postfix(const string& infixExp):
    infixExpression(infixExp) {}
```
infix2Postfix Class

```cpp
void infix2Postfix::setInfixExp(const string& infixExp)
{
    // assign a new infix expression
    infixExpression = infixExp;
    // make postfixExpression the NULL string
    postfixExpression = "";
}

string infix2Postfix::postfix()
{
    expressionSymbol op;
    // maintain rank for error checking
    int rank = 0, i;
    char ch;

    // process until end of the expression
    for (i=0; i < infixExpression.length(); i++)
    {
        ch = infixExpression[i];

        // ******** process an operand ********
        // an operand is a single digit non-negative integer
        if (isdigit(ch))
        {
            // add operand to postfix expression
            postfixExpression += ch;
        }
    }
}
```
infix2Postfix Class

// a blank
postfixExpression += ch;
postfixExpression += ' ';
// rank of an operand is 1, accumulated rank
// must be 1
rank++; 
if (rank > 1)
    throw expressionError("infix2Postfix:
Operator expected");
}
// ********** process an operator or '(' **********
else if (isOperator(ch) || ch == '(')
{
    // rank of an operator is -1. rank of '(' is 0.
    // accumulated rank should be 0
    if (ch != '(')
        rank--; 
    if (rank < 0)
        throw expressionError("infix2Postfix:
Operand expected");
else
    {
        // rank of an operator is -1. rank of '(' is 0.
        // accumulated rank should be 0
        if (ch != '(')
            rank--;

        if (rank < 0)
            throw expressionError("infix2Postfix:
Operand expected");
else
infix2Postfix Class

// output the operators on the stack with higher // or equal precedence. push the current operator // on the stack
op = expressionSymbol(ch);
outputHigherOrEqual(op);
operatorStack.push(op);

} else if (ch == rParen)
{
    // build an expressionSymbol object holding ')', which // has precedence lower than the stack precedence // of any operator except '. pop the operator stack // and output operators from the subexpression until // '(' surfaces or the stack is empty. if the stack is // empty, a '(' is missing; otherwise, pop off '.
    op = expressionSymbol(ch);
    outputHigherOrEqual(op);
    if(operatorStack.empty())
       throw expressionError("infix2Postfix: Missing '('");
    else
       operatorStack.pop(); // get rid of '('}
```cpp
// ********** make sure ch is whitespace **********
else if (!isspace(ch))
    throw expressionError("infix2Postfix: Invalid input");

// finish processing
if (rank != 1)
    throw expressionError("infix2Postfix: Operand expected");
else{
    // flush operator stack and complete expression evaluation.
    // if find left parenthesis, a right parenthesis is missing.
    while (!operatorStack.empty())
    {
        op = operatorStack.top();
        operatorStack.pop();
        if (op.getOp() == lParen)
            throw expressionError("infix2Postfix: Missing ')");
        else{
            postfixExpression += op.getOp();
            postfixExpression += ' ';
        }
    }
}

return postfixExpression;
```

#ifndef POSTFIX_EVALUATION
#define POSTFIX_EVALUATION

#include <iostream>
#include <string>
#include <stack>
#include <ctype.h>

#include "d_except.h" // for expressionError exception

using namespace std;

class postfixEval
{
    public:
        postfixEval();
            // default constructor. postfix expression is NULL string

        string getPostfixExp() const;
            // return the postfix expression

        void setPostfixExp(const string& postfixExp);
            // change the postfix expression
}
postfixEval Class

```cpp
int evaluate();
    // evaluate the postfix expression and return
    // its value. the function throws expressionError
    // if an error occurs during evaluation

private:

    string postfixExpression;
    // the postfix expression to evaluate

    stack<int> operandStack;
    // stack of operands

    void getOperands(int& left, int& right);
    // pop left and right operands from stack.
    // Precondition: the stack must have at least two entries.
    // if the stack is empty prior to a pop() operation, the
    // function throws the exception expressionError

    int compute(int left, int right, char op) const;
    // compute "left op right". if right is 0 and op
    // is '/' or '%', the function throws expressionError

    bool isOperator(char ch) const;
    // is ch one of '+','-','*','/','%','^'

};
```
postfixEval Class

```cpp
void postfixEval::getOperands(int& left, int& right)
{
    // can we pop the right operand?
    if (operandStack.empty())
        throw expressionError("postfixEval: Too many operators");

    // pop right operand
    right = operandStack.top();
    operandStack.pop();

    // can we pop the left operand?
    if (operandStack.empty())
        throw expressionError("postfixEval: Too many operators");

    // pop left operand
    left = operandStack.top();
    operandStack.pop();
}

int postfixEval::compute(int left, int right, char op) const
{
    int value;
```
postfixEval Class

// evaluate "left op right"
switch(op)
{
  case '+':    value = left + right;
               break;
  case '-':    value = left - right;
               break;
  case '*':    value = left * right;
               break;
  case '%':    if (right == 0)
               throw expressionError("postfixEval: divide by 0");
               value = left % right;
               break;
  case '/':    if (right == 0)
               throw expressionError("postfixEval: divide by 0");
               value = left / right;
               break;
  case '^':    // make sure we are not computing 0^0
                if (left == 0 && right == 0)
                throw expressionError("postfixEval: 0^0 undefined");
}
postfixEval Class

// general case. compute value = 1*left*...*left.
// if right == 0, skip the loop and left^0 is 1
while (right > 0)
{
    value *= left;
    right--;
}
break;

return value;
}

bool postfixEval::isOperator(char ch) const
{
    return ch == '+' || ch == '-' || ch == '*' ||
    ch == '%' || ch == '/' || ch == '^';
}

// default constructor
postfixEval::postfixEval()
{}
string postfixEval::getPostfixExp() const
{
    return postfixExpression;
}

void postfixEval::setPostfixExp(const string& postfixExp)
{
    postfixExpression = postfixExp;
}

int postfixEval::evaluate()
{
    // expValue contains the evaluated expression
    int left, right, expValue;
    char ch;
    int i;

    // process characters until the end of the string is reached
    // or an error occurs
    for (i=0; i < postfixExpression.length(); i++)
    {
        // get the current character
        ch = postfixExpression[i];

postfixEval Class

// look for an operand, which is a single digit
// non-negative integer
if (isdigit(ch))
    // value of operand goes on the stack
    operandStack.push(ch - '0');
// look for an operator
else if (isOperator(ch))
{
    // pop the stack twice and get the
    // left and right operands
    getOperands(left, right);
    // evaluate "left op right" and push on stack
    operandStack.push(compute(left, right, ch));
}
// any other character must be whitespace.
// whitespace includes blank, tab, and newline
else if (!isspace(ch))
    throw expressionError("postfixEval: Improper char");
}

// the expression value is on the top of the stack.
// pop it off
expValue = operandStack.top();
operandStack.pop();
postfixEval Class

// if data remains on the stack, there are too
// many operands
if (!operandStack.empty())
    throw expressionError("postfixEval: Too many operands");

return expValue;
}
#endif // POSTFIX_EVALUATION
Program Example

// the program inputs an infix expression until the user enters an empty
// string, it uses the class infix2Postfix to convert the infix expression
// to postfix, handling errors that may occur by catching the corresponding
// expressionError exception. if there is no error, the postfix string is
// correctly formatted. use the class postfixEval to evaluate the postfix
// expression and output the result. this is the value of the original
// infix expression

#include <iostream>
#include <string>

// if Microsoft VC++, compensate for a getline() bug in <string>
#ifdef _MSC_VER
#include "d_util.h"
#endif _MSC_VER

#include "d_inftop.h"      // infix2Postfix class
#include "d_rpn.h"         // postfixEval class

using namespace std;
int main()
{
    // use iexp for infix to postfix conversion
    infix2Postfix iexp;
    // infix expression input and postfix expression output
    string infixExp, postfixExp;
    // use pexp to evaluate postfix expressions
    postfixEval pexp;

    // input and evaluate infix expressions until the user enters an empty string
    // get the first expression
    cout << "Enter an infix expression: ";
    getline(cin, infixExp);

    while (infixExp != "")
    {
        // an exception may occur. enclose the conversion to postfix
        // and the output of the expression value in a try block
        try
        {
            // convert to postfix
            iexp.setInfixExp(infixExp);
            postfixExp = iexp.postfix();

            // evaluate postfix expression
            pexp = postfixExp;
            cout << "Postfix expression: " << pexp << endl;

            // get next expression
            cout << "Enter an infix expression: ";
            getline(cin, infixExp);
        }  // end try
        // handle exception
        catch (exception e)
        {
            cout << "Error: " << e.what() << endl;
        }
    }  // end while
}  // end main
Program Example

// output the postfix expression
cout << "The postfix form is " << postfixExp << endl;

// use pexp to evaluate the postfix expression
pexp.setPostfixExp(postfixExp);

cout << "Value of the expression = " << pexp.evaluate() << endl << endl;
{
// catch an exception and output the error
    catch (const expressionError & ee)
        {
            cout << ee.what() << endl << endl;
        }
// input another expression
    cout << "Enter an infix expression: ";
    getline(cin, infixExp);
}

    return 0;
Program Example

/*
   Run:

Enter an infix expression: 3 \(^2\) \(^2\) (1+2)
The postfix form is 3 2 1 2 + ^ ^
Value of the expression = 6561

Enter an infix expression: 3 * (4 - 2 \(^5\)) + 6
The postfix form is 3 4 2 5 ^ - * 6 +
Value of the expression = -78

Enter an infix expression: (7 + 8*7
infix2Postfix: Missing ')'

Enter an infix expression: (9 + 7) 4
infix2Postfix: Operator expected

Enter an infix expression: 2*4*8/
infix2Postfix: Operand expected

Enter an infix expression:
*/
miniStack Class
miniStack Class

• Stack class could be efficiently implemented using Vector class.
• Associate the concept of “back” in the vector with “top” in the stack.
• An empty stack corresponds to a vector with size 0.
• Pushing an item onto the queue corresponds to adding an element at the back of the list with push_back() and increasing its size with 1.
• Popping an item from the queue corresponds to using pop_front() to remove the first element (front) of the list and decreasing its size by 1.
#ifndef VECTOR_BASED_STACK_CLASS
#define VECTOR_BASED_STACK_CLASS

#include <vector>  // vector class used by object composition
#include "d_except.h"  // for underflowError exception

using namespace std;

template <typename T>
class miniStack
{
    public:
        miniStack();  // constructor. create an empty stack
        void push(const T& item);  // push (insert) item onto the stack.
        // Postcondition: the stack has a new topmost element and
        // the stack size increases by 1
### miniStack Class

```plaintext
void pop();  
// remove the item from the top of the stack.  
// Precondition: the stack is not empty.  
// if the stack is empty, the function throws  
// the underflowError exception

T& top();  
// return a reference to the element on the top  
// of the stack.  
// Precondition: the stack is not empty.  
// if the stack is empty, the function throws  
// the underflowError exception

const T& top() const;  
// constant version of top()

bool empty() const;  
// determine whether the stack is empty

int size() const;  
// return the number of elements in the stack
```
miniStack Class

```cpp
private:
    vector<T> stackVector;
    // a vector object maintains the stack items and size

// the constructor has nothing to do. the default
// constructor for the vector class initializes
// stackVector to be empty
template <typename T>
miniStack<T>::miniStack()
{
}

// push item on the stack by inserting it at
// the rear of the vector
template <typename T>
void miniStack<T>::push(const T& item)
{
    stackVector.push_back(item);
}
```
miniStack Class

// pop the stack by removing the item at the rear of the vector
template <typename T>
void miniStack<T>::pop()
{
    // check for an empty stack
    if (empty())
        throw underflowError("miniStack pop(): stack empty");

    // pop the stack
    stackVector.pop_back();
}

// the top of the stack is at the rear of the vector
template <typename T>
T& miniStack<T>::top()
{
    // check for an empty stack
    if (empty())
        throw underflowError("miniStack top(): stack empty");

    // return the element at the rear of the vector
    return stackVector.back();
}
miniStack Class

// constant version of top()
template <typename T>
const T& miniStack<T>::top() const
{
    // check for an empty stack
    if (empty())
        throw underflowError("miniStack top(): stack empty");
    // return the element at the rear of the vector
    return stackVector.back();
}

template <typename T>
bool miniStack<T>::empty() const
{
    return stackVector.size() == 0;
}

template <typename T>
int miniStack<T>::size() const
{
    return stackVector.size();
}
#endif // VECTOR_BASED_STACK_CLASS
Queue ADT
Queues

• Like vectors and lists, queue is also an ordered collection, but the construction and access rules are purposely limited.

• A Queue is a FIFO (First in First Out) data structure. Elements are inserted in the Rear of the queue and are removed from the Front.

• Only the front (least recently inserted) and back (most recently inserted) element may be accessed at any given time.
  
  • Actually, many authors would limit access to only the front element.

• In general, queues are used when things must be processed "in order", but can “pile up” before we get to them.
STL Queue Container in C++
std::queue - Operations

queue();  
Create an empty queue. This is the default constructor.

bool empty() const;
Check whether the queue is empty. Return true if it is empty and false otherwise.

T& front();
Return a reference to the value of the item at the front of the queue.
   Precondition: The queue is not empty.

const T& front();
Constant version of front().
std::queue - Operations

T& back();
Return a reference to the value of the item at the back of the queue.
    Precondition: The queue is not empty.

const T& back();
Constant version of back().

int size() const;
Return the number of elements in the queue.
std::queue - Operations

**void pop();**

Remove the item from the front of the queue.

- **Precondition:** The queue is not empty.
- **Postcondition:** The element at the front of the queue is the element that was added immediately after the element just popped or the queue is empty.

**void push (const T &item);**

Insert the argument item at the back of the queue.

- **Postcondition:** The queue has a new item at the back.
Queue Instantiation

• Similar to stack, queue object is instantiated with specifying the type of queue elements:
  • queue <string> strQ;

• Also, when instantiating the queue, we could specify both:
  • The type of elements to go on the queue:
    queue <string, list<string>> >
  • The sequence to be used as the implementing data structure:
    queue<string, list<string>> >

```cpp
queue<string, list<string>> q;
q.push( "abc" );
q.push( "de" );
assert (q.front () == "abc" );
assert (q.back () == "de" );
q.pop ();
assert (q.size () == 1 );
assert (q.front () == "de" );
q.pop ();
assert (q.empty () );
```
Implementing Queues using Arrays

• A list-based implementation is pretty straightforward. Adding and removing elements from the queue are $O(1)$ operations.

• An array-based implementations of queues is far more difficult than it was for stacks, because we must allow the structure to grow at one end and shrink from the other.

• What do we do when we bump into the end of the array and still want to add something to the queue?
  • There may be lots of unused space at the front, so we should be able to squeeze the new data in somewhere!
  • Now, we could just copy all the remaining data back to the beginning of the array. But doing this would make adding data to the end of the queue an $O(N)$ operation where $N$ is the number of elements still in the queue).
  • For the list we managed this in $O(1)$ time, so we might ask if we can’t do that well with arrays.
Implementing Queues using Arrays

• The solution to this problem is to simply let start and stop “wrap around” to the beginning of the array. For example, when adding to the queue, instead of:

  \[
  \text{stop}++; \\
  \]

we use:

  \[
  \text{stop = (stop +1) \% size;} \\
  \]

where size is the size of the array being used to hold the queue.

```cpp
template <class T>
void queue<T>::push (const T& x)
{
    assert (theSize < ArraySize);
    stop = (stop + 1) \% ArraySize ;
    array[stop] = x;
    theSize++;
}

template <class T>
inline void queue<T>::pop()
{
    assert (theSize > 0);
    start = (start + 1) \% ArraySize ;
    theSize--; 
}
```
miniQueue Class
miniQueue Class

• Queue class could be efficiently implemented using List class.

• Pushing an item onto the queue corresponds to adding an element at the back of the list with push_back() and increasing its size with 1.

• Popping an item from the queue corresponds to using pop_front() to remove the first element (front) of the list and decreasing its size by 1.
miniQueue Class

```c++
#ifndef QUEUE_CLASS
#define QUEUE_CLASS

#include <list> // list class used by object composition
#include "d_except.h"

using namespace std;

// implements a queue
template <typename T>
class miniQueue
{
    public:

        miniQueue();
        // constructor; create an empty queue

        void push(const T& item);
        // insert an element at the back of the queue.
        // Postcondition: the queue has one more element
```
miniQueue Class

void pop();
    // remove an element from the front of the queue.
    // Precondition: the queue is not empty. if the
    // queue is empty, the function throws the
    // underflowError exception
    // Postcondition: the queue has one less element

T& front();
    // return a reference to the front of the queue.
    // Precondition: the queue is not empty. if the
    // the queue is empty, the function throws the
    // underflowError exception

const T& front() const;
    // constant version of front()

int size() const;
    // return the queue size

bool empty() const;
    // is the queue empty?
miniQueue Class

private:
    list<T> qlist;
    // a list object maintains the queue items and size
};

// the constructor has nothing to do. the default
// constructor for the list class initializes
// qlist to be empty
template <typename T>
miniQueue<T>::miniQueue()
{
}

// insert item into the queue by inserting it at
// the back of the list
template <typename T>
void miniQueue<T>::push(const T& item)
{
    qlist.push_back(item);
}
miniQueue Class

// remove the item at the front of the queue by erasing the front of the list
template <typename T>
void miniQueue<T>::pop()
{
    // if queue is empty, throw underflowError
    if (qlist.size() == 0)
        throw underflowError("miniQueue pop(): empty queue");

    // erase the front
    qlist.pop_front();
}

template <typename T>
T& miniQueue<T>::front()
{
    // if queue is empty, throw underflowError
    if (qlist.size() == 0)
        throw underflowError("miniQueue front(): empty queue");

    return qlist.front();
}
miniQueue Class

template<typename T>
const T& miniQueue<T>::front() const
{
    // if queue is empty, throw underflowError
    if (qlist.size() == 0)
        throw underflowError("miniQueue front(): empty queue");

    return qlist.front();
}

template<typename T>
T& miniQueue<T>::back()
{
    // if queue is empty, throw underflowError
    if (qlist.size() == 0)
        throw underflowError("miniQueue back(): empty queue");

    return qlist.back();
}
miniQueue Class

template<typename T>
const T& miniQueue<T>::back() const
{
    // if queue is empty, throw underflowError
    if (qlist.size() == 0)
        throw underflowError("miniQueue back(): empty queue");
    return qlist.back();
}

// return the queue size
template<typename T>
template<typename T>
const int miniQueue<T>::size() const
{
    return qlist.size();
}

// is the queue empty?
template<typename T>
bool miniQueue<T>::empty() const
{
    return qlist.empty();
}

#endif // QUEUE_CLASS
Questions?
Assignment #3

• Due Sun Oct 6th, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #6, #9, & #7:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.23</td>
<td>322</td>
</tr>
<tr>
<td>6.25</td>
<td>322</td>
</tr>
<tr>
<td>9.18</td>
<td>497</td>
</tr>
<tr>
<td>9.22</td>
<td>497</td>
</tr>
<tr>
<td>7.17</td>
<td>378</td>
</tr>
<tr>
<td>7.18</td>
<td>378</td>
</tr>
</tbody>
</table>

• Programming Assignment
  • Ford & Topp, Chapter #6:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.37</td>
<td>325</td>
</tr>
</tbody>
</table>
Assignment #3

• Due Sun Oct 6th, 11:59pm

• Submission Format:

  • Written Assignment
    • Create single PDF file with name: cs361_assignment_3_<firstName>_<lastName>
    • Have a cover page with your name and your email
    • Submit through Blackboard.

  • Programming Assignment
    • Make sure your program compiles and executes using g++ on Dept’s Linux machines.
    • Create a “Readme.txt” file that list how to compile and execute your program. Include your name and your email.
    • Your main file should be named “main.cpp”.
    • You should have a Makefile.
    • Zip all files (.h, .c, Makefile, etc.) and name it: program
    • Submit through Blackboard.

• Final Submission Materials (2 Files):
  • One PDF for written assignment.
  • One ZIP file for the program