Eric Roberts
CS106B

Handout #35A
February 11, 2013

Solutions to Section Handout #5

Problem 1. Implementing the array-with-gap form of the two stack model

```
private:

/* Constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters  */
   int capacity;          /* Effective size of the array  */
   int nBefore;           /* Size of the before stack     */
   int nAfter;            /* Size of the after stack      */

/* Make it illegal to copy editor buffers */

   EditorBuffer(const EditorBuffer & value) { }
   const EditorBuffer & operator=(const EditorBuffer & rhs) { return *this; }

/* Private method prototypes */

   void pushBefore(char ch);
   void pushAfter(char ch);
   char popBefore();
   char popAfter();
   void expandCapacity();
```

```
/*
 * File: buffer.cpp (gap-buffer version)
 * -------------------------------------
 * This file implements the EditorBuffer class using the ends of a dynamic
 * array to represent two stacks.
 */

#include <iostream>
#include "buffer.h"
#include "error.h"
using namespace std;

/*
 * Implementation notes: Buffer constructor and destructor
 * -------------------------------------------------------
 * The constructor must set up the initial configuration of the empty
 * buffer.  The destructor frees the dynamic array.
 */

EditorBuffer::EditorBuffer() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   nBefore = 0;
   nAfter = 0;
}

EditorBuffer::~EditorBuffer() {
   delete[] array;
}
```

☞

```
/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * The four moveCursor methods use push and pop to transfer values
 * between the two stacks.
 */

void EditorBuffer::moveCursorForward() {
   if (nAfter != 0) {
      pushBefore(popAfter());
   }
}

void EditorBuffer::moveCursorBackward() {
   if (nBefore != 0) {
      pushAfter(popBefore());
   }
}

void EditorBuffer::moveCursorToStart() {
   while (nBefore != 0) {
      pushAfter(popBefore());
   }
}

void EditorBuffer::moveCursorToEnd() {
   while (nAfter != 0) {
      pushBefore(popAfter());
   }
}

/*
 * Implementation notes: character insertion and deletion
 * ------------------------------------------------------
 * Each of the functions that inserts or deletes characters can do so
 * with a single push or pop operation.
 */

void EditorBuffer::insertCharacter(char ch) {
   pushBefore(ch);
}

void EditorBuffer::deleteCharacter() {
   if (nAfter != 0) {
      popAfter();
   }
}

/*
 * Implementation notes: getText and getCursor
 * -------------------------------------------
 * The getText implementation uses a form of the string constructor that
 * takes a C-style string and a length.
 */

string EditorBuffer::getText() const {
   return string(array, nBefore) + string(array + capacity - nAfter, nAfter);
}

int EditorBuffer::getCursor() const {
   return nBefore;
}
```

```
/*
 * Implementation notes: pushBefore, pushAfter, popBefore, popAfter
 * ----------------------------------------------------------------
 * These methods simulate the stack operation at the appropriate end
 * of the array.  This level of decomposition is included to make the
 * stack metaphor more obvious.
 */

void EditorBuffer::pushBefore(char ch) {
   if (nBefore + nAfter == capacity) expandCapacity();
   array[nBefore++] = ch;
}

void EditorBuffer::pushAfter(char ch) {
   if (nBefore + nAfter == capacity) expandCapacity();
   nAfter++;
   array[capacity - nAfter] = ch;
}

char EditorBuffer::popBefore() {
   if (nBefore == 0) error("popBefore: Stack is empty");
   nBefore--;
   return array[nBefore];
}

char EditorBuffer::popAfter() {
   if (nAfter == 0) error("popAfter: Stack is empty");
   return array[capacity - nAfter--];
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the size of the array whenever the old one
 * runs out of space.  To do so, expandCapacity allocates a new array,
 * copies the old characters to the new array, and then frees the old array.
 */

void EditorBuffer::expandCapacity() {
   char *oldArray = array;
   int oldCapacity = capacity;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < nBefore; i++) {
      array[i] = oldArray[i];
   }
   for (int i = 0; i < nAfter; i++) {
      array[capacity - i - 1] = oldArray[oldCapacity - i - 1];
   }
   delete[] oldArray;
}
```

Problem 2: Doubly linked lists

```
private:

/*
 * Implementation notes
 * --------------------
 * In this representation, the buffer is coded as a doubly linked
 * list that is chained into a ring, with the dummy cell at both
 * the beginning and the end.
 */

/*
 * Type: Cell
 * ----------
 * This structure stores a single character along with links to the
 * previous and next cells in the ring.
 */

   struct Cell {
      char ch;
      Cell *prev;
      Cell *next;
   };

/* Data fields required for the linked-list representation */

   Cell *start;                 /* Pointer to the dummy cell      */
   Cell *cursor;                /* Pointer to cell before cursor  */

/* Make it illegal to copy editor buffers */

   EditorBuffer(const EditorBuffer & value) { }
   const EditorBuffer & operator=(const EditorBuffer & rhs) { return *this; }
```

```
/*
 * File: buffer.cpp (doubly linked version)
 * ----------------------------------------
 * This file implements the EditorBuffer class using a doubly linked
 * list to represent the buffer.
 */

#include <iostream>
#include "buffer.h"
using namespace std;

/*
 * Implementation notes: EditorBuffer constructor
 * ----------------------------------------------
 * This function initializes an empty editor buffer, represented
 * as a doubly linked list.  In this implementation, the ends of
 * the linked list are joined to form a ring, with the dummy cell
 * at both the beginning and the end.  This representation makes
 * it possible to implement the moveCursorToEnd method in constant
 * time, and reduces the number of special cases in the code.
 */

EditorBuffer::EditorBuffer() {
   start = cursor = new Cell;
   start->next = start;
   start->prev = start;
}
```

☞

```
/*
 * Implementation notes: EditorBuffer destructor
 * ---------------------------------------------
 * The destructor must delete every cell in the buffer.  Note
 * that the loop structure is not exactly the standard idiom for
 * processing every cell within a linked list, because it is not
 * legal to delete a cell and later look at its next field.
 */

EditorBuffer::~EditorBuffer() {
   Cell *cp = start->next;
   while (cp != start) {
      Cell *next = cp->next;
      delete cp;
      cp = next;
   }
   delete start;
}

/*
 * Implementation notes: cursor movement
 * -------------------------------------
 * In a doubly linked list, each of these operations runs in
 * constant time.
 */

void EditorBuffer::moveCursorForward() {
   if (cursor->next != start) {
      cursor = cursor->next;
   }
}

void EditorBuffer::moveCursorBackward() {
   if (cursor != start) {
      cursor = cursor->prev;
   }
}

void EditorBuffer::moveCursorToStart() {
   cursor = start;
}

void EditorBuffer::moveCursorToEnd() {
   cursor = start->prev;
}
```

```
/*
 * Implementation notes: insertCharacter, deleteCharacter
 * ------------------------------------------------------
 * This code is much like that used for the traditional linked
 * list except that more pointers need to be updated.
 */

void EditorBuffer::insertCharacter(char ch) {
   Cell *cp = new Cell;
   cp->ch = ch;
   cp->next = cursor->next;
   cp->prev = cursor;
   cursor->next->prev = cp;
   cursor->next = cp;
   cursor = cp;
}

void EditorBuffer::deleteCharacter() {
   if (cursor->next != start) {
      Cell *oldcell = cursor->next;
      cursor->next = oldcell->next;
      cursor->next->prev = cursor;
      delete oldcell;
   }
}

/*
 * Implementation notes: getText
 * -----------------------------
 * This method returns the string contained in the buffer by walking
 * through the linked list and concatenating each of the characters.
 */

string EditorBuffer::getText() const {
   string result = "";
   for (Cell *cp = start->next; cp != start; cp = cp->next) {
      result += cp->ch;
   }
   return result;
}

/*
 * Implementation notes: getCursor
 * -------------------------------
 * This method counts the number of times you need to advance a pointer
 * from the start before reaching the cursor position.
 */

int EditorBuffer::getCursor() const {
   int n = 0;
   for (Cell *cp = start; cp != cursor; cp = cp->next) {
      n++;
   }
   return n;
}
```