# Editor Buffers

Eric Roberts
CS 106B
February 4, 2015

## Why Look at Old Editors?

- In today’s class and again on Friday, I’m going to look at an ancient (at least in the sense of Internet time) editor technology, which is largely based on the TECO (Text Editor and COrrector) I used to prepare my doctoral thesis.
- Some of you will no doubt greet this idea with skepticism. Why should you study something so old that doesn’t meet even the most requirements we would insist on in an editor?
- We’ve kept that editor in CS 106B for two decades because:
  - It’s absolutely the best example of how using different data representations can have a profound effect on performance.
  - No modern editor is simple enough to understand in its entirety.
  - TECO is historically important as the first extensible editor. It was the first platform for EMACS, which is still in use today.
  - The strategies we describe for representing the editor persist in modern editor classes, such as Java’s **`JEditPane`**.

## WYSIWYG *vs.* Command Line

- Most editors today follow the WYSIWYG principle, which is an acronym for “what you see is what you get” that keeps the screen updated so that it shows the current document.
- TECO was a command-based editor. To edit a document, you enter commands that consist of a letter, possibly along with some additional data. Rather than showing the contents of the editor all the time, command-line editors showed the contents only when the user asked for them.
- Most histories of computer science date the first WYSIWYG editor to Xerox PARC in 1974. I saw one running almost two years earlier, which was written by a couple of graduate students at Harvard on their PDP-1 computer. That editor was called Splat! and seems largely lost to history.

# The PDP-1 Computer

- A great deal of early graphics development was done on the Digital Equipment Corporation’s PDP-1 computer, which was released in 1959. Most PDP-1s came with a Type 30 display, which was a 1024x1024 pixel display. Splat! ran on this display, but so did Spacewar!

## The Editor Commands

- Our minimal version of TECO has the following commands:

| | |
|---|---|
| **`I`***text* | Inserts the characters following the **`I`** into the buffer. |
| **`J`** | Moves the current point (the ***cursor***) to the beginning. |
| **`Z`** | Moves the cursor to the end of the buffer. |
| **`F`** | Moves the cursor forward one character. |
| **`B`** | Moves the cursor backward one character. |
| **`D`** | Deletes the character after the cursor. |
| **`Q`** | Exits from the editor. |

# Methods in the `EditorBuffer` Class

**`buffer.moveCursorForward()`**
Moves the cursor forward one character (does nothing if it’s at the end).

**`buffer.moveCursorBackward()`**
Moves the cursor backward one character (does nothing if it’s at the beginning).

**`buffer.moveCursorToStart()`**
Moves the cursor to the beginning of the buffer.

**`buffer.moveCursorToEnd()`**
Moves the cursor to the end of the buffer.

**`buffer.insertCharacter(ch)`**
Inserts the character **`ch`** at the cursor position and advances the cursor past it.

**`buffer.deleteCharacter()`**
Deletes the character after the cursor, if any.

**`buffer.getText()`**
Returns the contents of the buffer as a string.

**`buffer.getCursor()`**
Returns the position of the cursor.

## The buffer.h Interface

```
/*
 * File: buffer.h
 * --------------
 * This file defines the interface for an editor buffer class.
 */

#ifndef _buffer_h
#define _buffer_h

class EditorBuffer {

public:

/*
 * Constructor: EditorBuffer
 * -------------------------
 * Creates an empty editor buffer.
 */

   EditorBuffer();

/*
 * Destructor: ~EditorBuffer
 * -------------------------
 * Frees any storage associated with this buffer.
 */

   ~EditorBuffer();
```

## The **buffer.h** Interface

```
/*
 * Methods: moveCursorForward, moveCursorBackward
 * Usage: buffer.moveCursorForward();
 *        buffer.moveCursorBackward();
 * -----------------------------------
 * These functions move the cursor forward or backward one
 * character, respectively.  If you call moveCursorForward
 * at the end of the buffer or moveCursorBackward at the
 * beginning, the function call has no effect.
 */

   void moveCursorForward();
   void moveCursorBackward();

/*
 * Methods: moveCursorToStart, moveCursorToEnd
 * Usage: buffer.moveCursorToStart();
 *        buffer.moveCursorToEnd();
 * --------------------------------
 * These functions move the cursor to the start or the end of this
 * buffer, respectively.
 */

   void moveCursorToStart();
   void moveCursorToEnd();
```

## The **buffer.h** Interface

```
/*
 * Method: insertCharacter
 * Usage: buffer.insertCharacter(ch);
 * ----------------------------------
 * This function inserts the single character ch into this
 * buffer at the current cursor position.  The cursor is
 * positioned after the inserted character, which allows
 * for consecutive insertions.
 */

   void insertCharacter(char ch);

/*
 * Method: deleteCharacter
 * Usage: buffer.deleteCharacter();
 * --------------------------------
 * This function deletes the character immediately after
 * the cursor.  If the cursor is at the end of the buffer,
 * this function has no effect.
 */

   void deleteCharacter();
```

# The **`buffer.h`** Interface

```
/*
 * Method: getText
 * Usage: string str = buffer.getText();
 * -------------------------------------
 * Returns the contents of the buffer as a string.
 */

   std::string getText() const;

/*
 * Method: getCursor
 * Usage: int cursor = buffer.getCursor();
 * ---------------------------------------
 * Returns the index of the cursor.
 */

   int getCursor() const;
```

*Private section goes here.*

```
#endif
```

# Where Do We Go From Here?

- Our goal for this week is to implement the **`EditorBuffer`** class in three different ways and to compare the algorithmic efficiency of the various options. Those representations are:

    1. A simple *array model*.
    2. A *two-stack model* that uses a pair of character stacks.
    3. A *linked-list model* that uses pointers to indicate the order.

- For each model, we’ll calculate the complexity of each of the six fundamental methods in the **`EditorBuffer`** class. Some operations will be more efficient with one model, others will be more efficient with a different underlying representation.

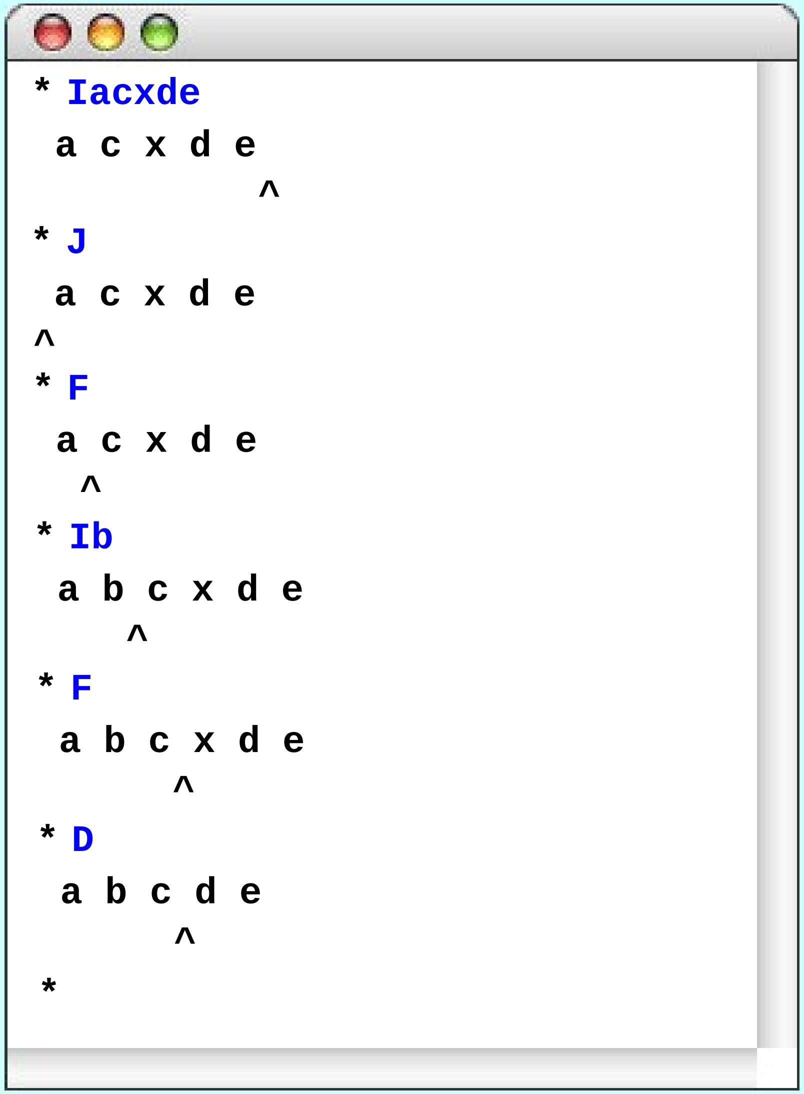
## The Array Model

- Conceptually, the simplest strategy for representing the editor buffer is to use an array for the individual characters.
- To ensure that the buffer can contain an arbitrary amount of text, it is important to allocate the array storage dynamically and to expand the array whenever the buffer runs out of space.
- The array used to hold the characters will contain elements that are allocated but not yet in use, which makes it necessary to distinguish the ***capacity*** of the array from its ***effective size***.
- In addition to the size and capacity information, the data structure for the editor buffer must contain an additional integer variable that indicates the current position of the cursor. This variable can take on values ranging from 0 up to and including the length of the buffer.

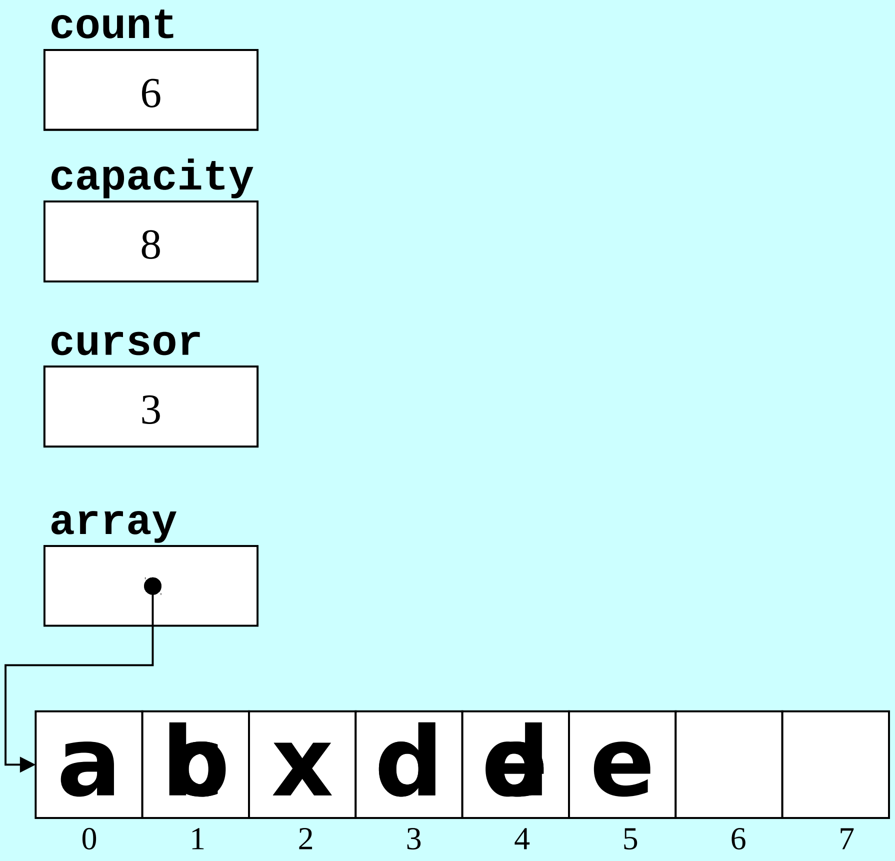
# Private Data for Array-Based Buffer

```
private:

/*
 * Implementation notes: Buffer data structure
 * -------------------------------------------
 * In the array-based implementation of the buffer, the characters in the
 * buffer are stored in a dynamic array.  In addition to the array, the
 * structure keeps track of the capacity of the buffer, the length of the
 * buffer, and the cursor position.  The cursor position is the index of
 * the character that follows where the cursor would appear on the screen.
 */

/* Constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters     */
   int capacity;          /* Allocated size of that array    */
   int length;            /* Number of character in buffer   */
   int cursor;            /* Index of character after cursor */

/* Private method prototype */

   void expandCapacity();
```

# Array Editor Simulation



```
* Iacxde
 a c x d e
          ^
* J
 a c x d e
^
* F
 a c x d e
  ^
* Ib
 a b c x d e
    ^
* F
 a b c x d e
      ^
* D
 a b c d e
      ^
*
```



## Array-Based Buffer Implementation

```
/*
 * File: buffer.cpp (array version)
 * --------------------------------
 * This file implements the EditorBuffer class using an array representation.
 */

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

/*
 * Implementation notes: Constructor and destructor
 * ------------------------------------------------
 * The constructor initializes the private fields.  The destructor
 * frees the heap-allocated memory, which is the dynamic array.
 */

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

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

## Array-Based Buffer Implementation

```
/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * The four moveCursor methods simply adjust the value of the
 * cursor instance variable.
 */

void EditorBuffer::moveCursorForward() {
   if (cursor < length) cursor++;
}

void EditorBuffer::moveCursorBackward() {
   if (cursor > 0) cursor--;
}

void EditorBuffer::moveCursorToStart() {
   cursor = 0;
}

void EditorBuffer::moveCursorToEnd() {
   cursor = length;
}
```

# Array-Based Buffer Implementation

```
/*
 * Implementation notes: insertCharacter and deleteCharacter
 * ---------------------------------------------------------
 * Each of the functions that inserts or deletes characters must shift
 * all subsequent characters in the array, either to make room for new
 * insertions or to close up space left by deletions.
 */

void EditorBuffer::insertCharacter(char ch) {
   if (length == capacity) expandCapacity();
   for (int i = length; i > cursor; i--) {
      array[i] = array[i - 1];
   }
   array[cursor] = ch;
   length++;
   cursor++;
}

void EditorBuffer::deleteCharacter() {
   if (cursor < length) {
      for (int i = cursor+1; i < length; i++) {
         array[i - 1] = array[i];
      }
      length--;
   }
}
```

# Array-Based Buffer Implementation

```
/* Simple getter methods: getText, getCursor */

string EditorBuffer::getText() const {
   return string(array, length);
}

int EditorBuffer::getCursor() const {
   return cursor;
}

/*
 * 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;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < length; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

# The Two-Stack Model

- In the two-stack implementation of the **`EditorBuffer`** class, the characters in the buffer are stored in one of two stacks. The characters before the cursor are stored in a stack called **`before`** and the characters after the cursor are stored in a stack called **`after`**. Characters in each stack are stored so that the ones close to the cursor are near the top of the stack.

- For example, given the buffer contents

```
a b c|d e
```

  the characters would be stored like this:

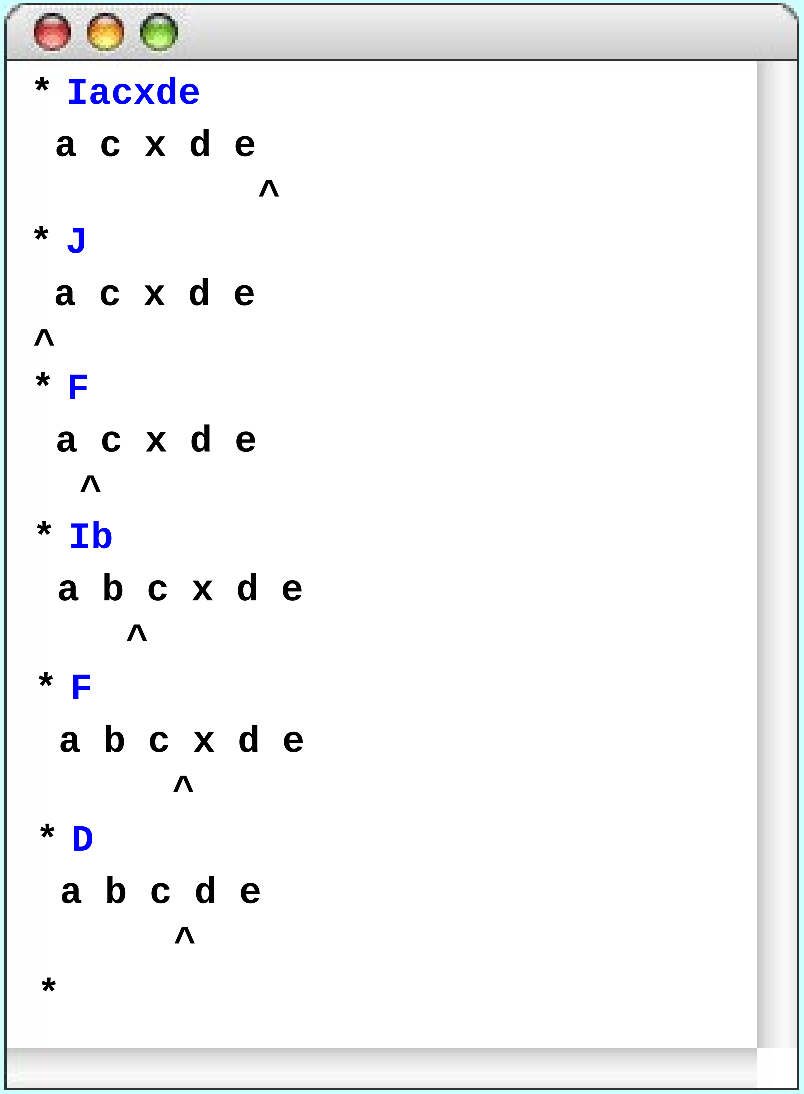
```
  c
  b           d
  a           e
------      ------
before      after
```

- The code for the two-stack model is in the reader, but we’ll write it together in class.

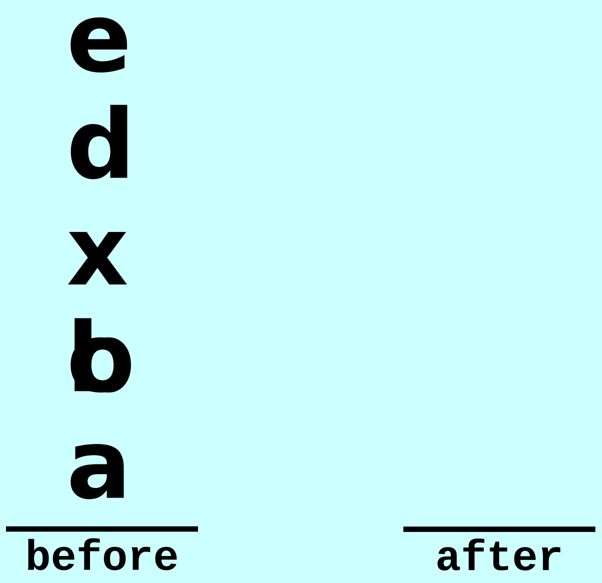
# Private Data for Stack-Based Buffer

```
private:

/*
 * Implementation notes: Buffer data structure
 * ------------------------------------------
 * In the stack-based buffer model, the characters are stored in two
 * stacks.  Characters before the cursor are stored in a stack named
 * "before"; characters after the cursor are stored in a stack named
 * "after".  In each case, the characters closest to the cursor are
 * closer to the top of the stack.  The advantage of this
 * representation is that insertion and deletion at the current
 * cursor position occurs in constant time.
 */

/* Instance variables */

   CharStack before;    /* Stack of characters before the cursor */
   CharStack after;     /* Stack of characters after the cursor  */
```

# Stack Editor Simulation



```
* Iacxde
 a c x d e
         ^
* J
 a c x d e
^
* F
 a c x d e
  ^
* Ib
 a b c x d e
    ^
* F
 a b c x d e
      ^
* D
 a b c d e
      ^
*
```



# Stack-Based Buffer Implementation

```
/*
 * File: buffer.cpp (stack version)
 * --------------------------------
 * This file implements the EditorBuffer class using a pair of
 * stacks to represent the buffer.  The characters before the
 * cursor are stored in the before stack, and the characters
 * after the cursor are stored in the after stack.
 */

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

/*
 * Implementation notes: EditorBuffer constructor/destructor
 * ---------------------------------------------------------
 * In this representation, the implementation of the CharStack class
 * automatically takes care of allocation and deallocation.
 */

EditorBuffer::EditorBuffer() {
   /* Empty */
}

EditorBuffer::~EditorBuffer() {
   /* Empty */
}
```

# Stack-Based Buffer Implementation

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

void EditorBuffer::moveCursorForward() {
   if (!after.isEmpty()) {
      before.push(after.pop());
   }
}

void EditorBuffer::moveCursorBackward() {
   if (!before.isEmpty()) {
      after.push(before.pop());
   }
}

void EditorBuffer::moveCursorToStart() {
   while (!before.isEmpty()) {
      after.push(before.pop());
   }
}

void EditorBuffer::moveCursorToEnd() {
   while (!after.isEmpty()) {
      before.push(after.pop());
   }
}
```

# Stack-Based Buffer Implementation

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

void EditorBuffer::insertCharacter(char ch) {
   before.push(ch);
}

void EditorBuffer::deleteCharacter() {
   if (!after.isEmpty()) {
      after.pop();
   }
}
```

The End