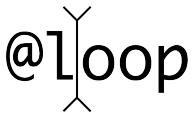
COMP 211: Principles of Imperative Computation, Fall 2014

Programming 6: Editor I



Due: Wednesday, November 5, 2014 by 6pm

For the programming portion of this week's homework, you'll implement a core data structure for a text editor: the gap buffer.

1 Getting the code

The code handout for this assignment is available from the course web page. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

To download and unzip the code, do the following:

1.1 Mac OS

1. On the assignments page of the course web site, right-click the Programming 6 `Code` link, and then click "Save Link as...", and choose to save it in `Documents/comp211/`.

2. Start a new terminal window and do the following:

   ```
   cd comp211
   tar -xzvkf hw6-handout.tgz
   ```

   This will make a directory called `comp211/hw6-handout` and extract all of the code for the assignment.

3. The file `hw6-handout/README.txt` contains a description of the various files, and instructions on running them.

1.2 Windows

1. On the assignments page of the course web site, right-click the Programming 6 `Code` link, and then click "Save Link as...", and choose to save it in your `comp211/`. To find this directory in your web browser, look in

   ```
   C:\cygwin\home\YOURNAME\comp211\
   ```

2. Start a new terminal window and do the following:

```
cd comp211
tar -xzkvf hw6-handout.tgz
```

This will make a directory called `comp211/hw6-handout` and extract all of the code for the assignment.

3. The file `hw6-handout/README.txt` contains a description of the various files, and instructions on running them.

2 Overview: A text editor based on gap buffers

In the next three weeks, you will do three related assignments, which combined will create a simple text editor, like Emacs or Sublime Text. The editor will be very basic: it will keep track of the current contents of a buffer, and a *point* in the buffer, which represents where the cursor is. For example, a buffer containing "the space race" with the cursor on the "a" represents the following editor state:

```
the space race
```

Then you can

- insert a character before the cursor. E.g. if we insert "Z" we'll get

  ```
  the spZace race
  ```

  Note that the cursor is still on the "a".

- delete the character before the cursor, which would give

  ```
  the sace race
  ```

  Note that the cursor is still on the "a".

- move the cursor to the left

  ```
  the space race
  ```

- move the cursor to the right

  ```
  the space race
  ```

Suppose you represented a buffer as an array, keeping track of the last used position:

s	p	a	c	e			

`last = 5`

Then inserting a character at the *end* of the buffer just requires putting the character into the array at position `last`, and incrementing `last`:

s	p	a	c	e	Z		

`last = 6`

However, if you move the cursor and then want to insert in the *middle* of the buffer, you need to move all of the characters after that point down by one:

s	p	Z	a	c	e		

`last = 6`

So inserting a character would be linear in the size of the document, which would be bad for large documents.

This motivates using something called a *gap buffer*. A gap buffer attempts to reduce the overhead of shifting by placing the empty portion of the array in the *middle*, rather than at the end. Hence the name "gap buffer" referring to the "gap" in the middle of the "buffer". The gap is not fixed to any one position. At any time it could be in the middle of the buffer or just at the beginning or anywhere in the buffer. We can immediately see the potential benefits of this approach.

Moving the **cursor** in the text editor corresponds to moving a gap, and thereby providing the unused portion of the array to be used for possible insertions. In the worst case scenario we have to move the gap from the beginning of the text file to the end. But if subsequent operations are only a few indexes apart, we will get a lot better performance compared to keeping the empty space at the end. This is why gap buffers increase performance of repetitive operations occurring at relatively close indexes. This is the common case for a text editor, where you will generally write many consecutive characters in succession.

Implementing a text editor as just one gap buffer is not particularly realistic, because it requires the entire file contents to be stored in a single, contiguous block of memory, which can be difficult to achieve for large files. In the next two weeks, you will fix this problem, but for this week, we'll work with a single gap buffer.

3 Gap Buffer

A gap buffer is an array of characters. The array is logically split into two segments of text - one at the beginning of the array, which grows upwards, and one at the end which grows downwards. The space between those two segments is called the *gap*. The gap is the **cursor** in the buffer. To move the cursor forwards, you just move the gap (assuming that the gap is not empty). To insert a character after the cursor, you place it at the beginning of the gap. To delete the character before the cursor, you just expand the gap.

To implement a gap buffer there are a couple bits of information that we need to keep track of. A gap buffer is represented in memory by an array of elements stored along with

its size (`limit`) and two integers representing the beginning (inclusive, `gap_start`) and end (exclusive, `gap_end`) of the gap (see Figure 1).

```
typedef struct gapbuf_header* gapbuf;
struct gapbuf_header {
    int limit;          /* limit > 0                     */
    char[] buffer;      /* \length(buffer) == limit      */
    int gap_start;      /* 0 <= gap_start                */
    int gap_end;        /* gap_start <= gap_end <= limit */
};
```

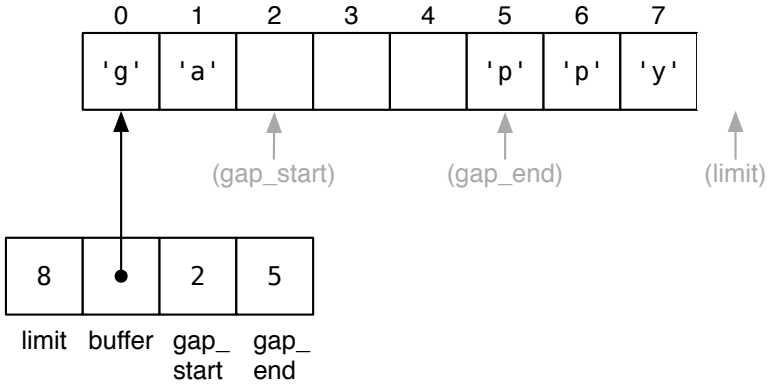


Figure 1: A gap buffer in memory.

3.1 Invariants

Task 1 (3 pts) *A valid gap buffer is non-NULL, has a strictly positive limit which correctly describes the size of its array, and has a gap start and gap end which are valid for the array. Implement the function*

```
bool is_gapbuf(struct gapbuf_header* G)
```

that formalizes the gap buffer data structure invariants. This function should return true when all of the invariants are satisfied, and false otherwise—except, because the `\length` *of an array can only be checked in a contract, an* `//@assert` *should fail if* `limit` *is incorrect.*

Because subsequent assignments will use gap buffers, it is especially important to specify the invariants precisely with `is_gapbuf`. This means that your data structure invariants will need to be very good, and we will test them very thoroughly. They need to permit anything permitted by the specification, and disallow anything that is not allowed by the specification.

Task 2 (2 pts) *Extend the function* `void test_gapbuf()` *with some additional tests. It should try several examples, some of which are valid and some of what are not, and check that* `is_gapbuf` *returns true or false as appropriate.*

You should then run your tests by loading `gapbuf.c0` in coin.

3.2 Operations

Text buffers may allow a variety of editing operations to be performed on them; for the purposes of this assignment, we'll consider only four operations: move forward a character, move backward a character, insert a character, and delete a character. As an example, below is a diagram of a gap buffer which is an array of characters with a gap in the middle (situated between the "p" and the "a" in "space"):

```
the sp[..]ace race
```

To move the gap (the cursor in the text editor) forward, we copy a character across it:

```
the spa[..]ce race
```

To delete a character (before the cursor), we simply expand the gap:

```
the sp[...]ce race
```

To insert a character (say, "i"), we write it into the left side of the gap (shrinking it by one):

```
the spi[..]ce race
```

The gap can be at the left end of the buffer,

```
[..]the space race
```

or at the right end of the buffer,

```
the space race[..]
```

and a buffer can be empty,

```
[...............]
```

or it can be full (this depends on the buffer size (limit))

```
the space ra[]ce!!
```

Note that in an emacs-like interface, where the cursor appears as a highlighted character in the buffer, the cursor will display on the character immediately following the gap. So following the examples above,

```
the sp[..]ace race
```

would display as:

```
the space race
```

while

```
the space race[..]
```

would display as:

```
the space race
```

In the above illustrations, we use dots to indicate spots in the gap buffer whose contents we don't care about. Those spots in the gap buffer don't need to contain the default character `'\0'` or the character `'.'` or anything else in particular.

Task 3 (4 pts) *Implement the following utility functions on gap buffers:*

Function:	***Returns true iff...***
`bool gapbuf_empty(gapbuf G)`	*... the gap buffer is empty*
`bool gapbuf_full(gapbuf G)`	*... the gap buffer is full*
`bool gapbuf_at_left(gapbuf G)`	*... the gap is at the left end of the buffer*
`bool gapbuf_at_right(gapbuf G)`	*... the gap is at the right end of the buffer*

Note that `gapbuf_at_left` means that the gap *starts* at the left end of the buffer, while `gapbuf_at_right` means that the gap *ends* at the right end of the buffer.

Task 4 (8 pts) *Implement the following interface functions for manipulating gap buffers:*

`gapbuf gapbuf_new(int limit)`	*Create a new gapbuf of size limit*
`void gapbuf_forward(gapbuf G)`	*Move the gap forward, to the right*
`void gapbuf_backward(gapbuf G)`	*Move the gap backward, to the left*
`void gapbuf_insert(gapbuf G, char c)`	*Insert the character c before the gap*
`void gapbuf_delete(gapbuf G)`	*Delete the character before the gap*

See page 5 for details.

Task 5 (3 pt) *Give good pre- and post-conditions for all of these functions:*

1. *All functions should require and ensure that the gap buffers are valid.*
2. *For* `new`*,* `forward`*,* `backward`*, and* `delete`*, you should give sufficiently strong pre-conditions that the operation can be successfully performed without failing (e.g. you cannot move the cursor forward if it is already all the way at the right of the buffer, and similar conditions for backward/insert/delete). You should also should document any special properties of the result in the post-condition. Hint: use* `empty`*,* `full`*,* `at_left`*, and* `at_right` *to do this.*

3.3 Testing

To test `empty`, `full`, `at_left`, `at_right`, you can put some tests in `test_gapbuf`, which can use the gap buffers you constructed to test `is_gapbuf`.

Alternatively, you can wait until you have implemented the next group of functions: you should be using `empty`/`full`/`at_left`/`at_right` in the contracts for `new`/`insert`/`delete`/`forward`/`backwa` so if you run the latter with `-d`, that will run the former, and you will notice if any of them are wrong. However, this may be more confusing than writing tests directly, because you will have to consider the behavior of these additional functions.

For testing `new`, `insert`, `delete`, `forward`, `backward`, you can test interactively by compiling and running the provided `gapbuf-test.c0`.

```
% cc0 -d -o gapbuf-test gapbuf.c0 gapbuf-test.c0
% ./gapbuf-test
```

Typing a character inserts that character, except for < (move left) and > (move right) and ^ (delete). We use this syntax rather than the arrow and delete keys so that it is possible to write an input down to communicate it. E.g. Try entering

```
space race<<<<<<<<<^p<<the >>^p>>>>>>>>!!<<<<<<<^^^^^great
```

and seeing what happens.

Note that all 9 of the functions listed above must be defined before you can compile `gapbuf-test`, but you can write and test some functions before writing others. We recommend that you first go through and make very simple "dummy" (wrong) implementations for each function, like

```
gapbuf gapbuf_new(int limit)
{
  return NULL;
}

void gapbuf_forward()
{
}
```

Then, as you fill in individual functions, you can test them with `gapbuf-test`. You will need to implement `new` first, and then `insert`, before you can test the remaining functions.