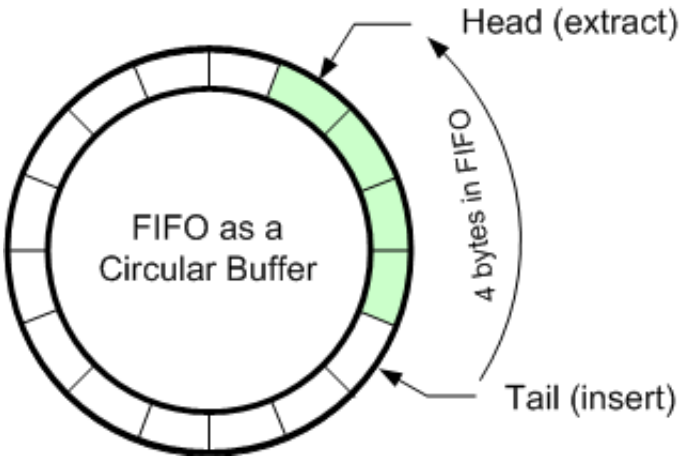
CSE 232, Lab Exercise 10

The Problem

We are going to work on making our own classes with private data members and special accessors. We are going to build a `CircularBuffer` class, a common data structure with well known accessors.

Some Background

Our `CircularBuffer` will be a data structure that stores `long`. A `CircularBuffer` is a fixed size FIFO (First In, First Out) data structure. It is essentially a line (a queue). First thing added (the Head position in the diagram) is the first thing read. The next thing added is at the Tail position. It is the last thing added, the last thing that will be readread. The underlying data structure for this approach has a fixed size data structure. It can become empty, it can become full. It does not grow or shrink in size over the course of the run of the program.



Things you can do with your CircularBuffer:

- you can add to the CircularBuffer. An element is added at the Tail position. The write position is then advanced (clockwise in the diagram
- you can remove an element. The element at the Head position is removed. The Head position is then advanced (clockwise in the diagram)
- you can test if it is full
- you can test if it is empty.
- you can report the front element
- you can report the back element

Your Tasks

We are going to make a `CircBuf` class with these characteristics and test it.

The Class

The `CircBuff` class will have an underlying data member `buf_` of type `vector<long>` of **fixed size**. This is a ***private*** data member, which will represent the underlying data array. `buf_` is private, you cannot access it from a main program using the class.

Data Members

- `int sz_` (the maximum size of the fixed size buffer)

- `int cnt_` (the number of active elements in the buffer)
- `size_t head_` (an index to the element that will be read by `front()`)
- `size_t tail_` (an index to where the ***next element*** will be written, where `add()` will put the next element)
- `buf_` , a `vector<long>`, the data being stored.

Function Members

- `CircBuf` constructor, one argument, the **fixed size** buffer of `long`
  - takes a default of 10, thus allowing for a default constructor.
- `CircBuf` constructor:
  - two args: an `initialization_list<long>` and a buffer size
    - if the parameter size is smaller than initialization list, throw a `runtime_error`
  - if successful, initializes the underlying vector to the provided parameter size and copies the values of the `initializer_list` into that vector
- `long front() const` member function, no parameters
  - if `CircBuf` is not empty, returns the long indexed by `head_`.
  - if the `CircBuf` is empty, throws a `runtime_error`
- `long back()const` member function, no parameters
  - if `CircBuf` is not empty, returns the last long (`tail_` -1).
  - if the `CircBuf` is empty, throws a `runtime_error`
- `void remove()` member function, no parameters
  - if `CircBuf` is not empty, advances `head_` by one.
  - if the `CircBuf` is empty, throws `runtime_error`
- `void add(long)` member function, single long parameter
  - if `CircBuf` is not full, places the parameter in `buf_` at the index indicated by `tail_`, advances `tail_` by one with wrap around as described
  - if `CircBuf` is full, throws a `runtime_error`
- `bool empty()const` member function, no parameters
  - returns `true` if the `CircBuf` is empty, `false` otherwise.
- `bool full()const` member function, no parameters.
  - returns `true` if the `CircBuf` is full, `false` otherwise.

Friends

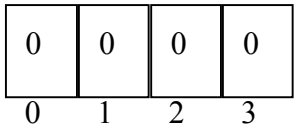
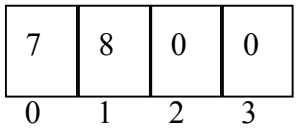
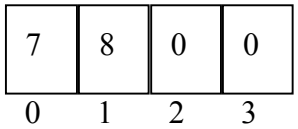
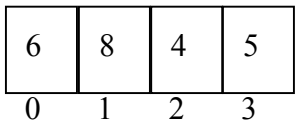
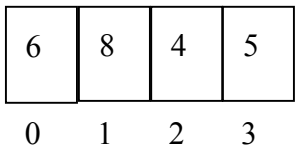
- `ostream& operator<<(ostream &out, const CircBuf &cb)`. This is a ***friend*** function (not a member). It prints the ***values*** at the front and back of the buffer (what is indexed by `head_` and one back from `tail_`), the `cnt_` of elements in the buffer and the underlying buffer array
  - if the buffer is empty, prints a message "`CircBuf empty`"
  - See Mimir test cases for examples

Implementation, circbuf.cpp

The most important thing to note is the circular nature of the buffer as implemented in a vector. The two indicies: `front_` (where elements are read from) and `back_` (where elements are added). They can wrap around the buffer like we have done with clock arithmetic:

- if `back_` goes past the last index of the data structure, it "wraps around" to the first index using the modulus operator (%) based on the fixed buffer size
- the same is true for `front_`

- in this way you can keep reusing the underlying buffer.

 head_ = 0 tail_ = 0 cnt_ = 0	Initial situation
 head_ = 0 tail_ = 2 cnt_ = 2	cb.add(7) cb.add(8) remember, tail_ is where the next add will put an element
 head_ = 2 tail_ = 2 cnt_ = 0	cb.remove() cb.remove() Nothing is erased, just ignored!
 head_ = 2 tail_ = 1 cnt_ = 3	cb.add(4) cb.add(5) cb.add(6) note value 8 at indx 1 is still there even though it was removed. Next add will overwrite it.
 head_ = 0 tail_ = 1 cnt_ = 1	cb.remove() cb.remove()

Notes

- `head_ == tail_` in two situations: full and empty. How to differentiate full from empty?
- You have to be careful with a vector in this case. The vector will grow under a push_back but you want this buffer to be of a fixed size. How to deal with that?

Test

You are given `lab10_circbuf.h` . Implement `lab10_circbuff.cpp` and test on Mimir

Extra

For those that find the above relatively straight forward to do, try the follow extra work:

- it might seem a little odd, but you could overload the + operator to place the next element in the buffer (a different way to do add). It would have to allow statements like the following.

```
cb = cb + 5;
cb = 5 + cb;
```

  In this case, the operator would make a new `CircBuf`, with the element added to the end. Thus this would be the addition of a `CircBuf` and a long.

What to do for the fixed size of the newly made CircBuf? You could make the new buffer a straight copy of cb and throw an error if it exceeds the size. If not big enough, you could also make it one bigger. Which would be better?

Uncomment the appropriate parts in lab10_circbuf.h and implement. No tests in Mimir, you're on your own here.

- if you are **really** bored, you could create the addition of two `CircBuf`. It would take the two contents and combine them. **Order here would be important** (contents of the lhs first followed by the rhs).

```
cb = cb + another_buff  # cb contents first, another_buf second
cb = another_buf + cb   # another_buf contents first, cb second
```

Again, what to do with the fixed size of the returned buffer? This is a tougher call. It seems likely that the combined buffers could be too big (bigger than either of the two argument CircBufs). Should we throw an error or make the new CircBuf big enough to hold the result? Interesting problem.

Again, uncomment the appropriate part of lab10_circbuf.h and implement. Again, no tests in Mimir.

See, class design can be lots of fun!?