CO889 Assignment 3: RingBuffer

Andrew Runnalls*

December 2019

1 Introduction

The second assignment (LZ) gave you practice in using some of the container class templates provided by the C++ standard library. In this assignment, you will get some experience of what it is like to write a container class template of your own.

2 Ring Buffers

A **ring buffer** is a data structure which uses an area of storage of fixed size to implement a **queue**, i.e. a sequence of objects (for illustration, think of them as transactions) which are processed using a first-in first-out protocol. In this assignment you'll be writing a class template which implements a ring buffer containing objects of some type `T`. A client of a ring buffer class will be able to add a new object at the end of the queue using the method `push_back`. A client will be able to access the object at the front of the queue (if any) using the method `front`, and when the client has finished dealing with that object, it can be removed from the queue using the method `pop_front`. The ring buffer classes also provide nested types `iterator` and `const_iterator` enabling a client to work through all the objects currently in the queue.

The diagrams that follow illustrate the operation of a ring buffer `rb` with capacity 5 containing objects of type `Tx` representing some sort of transaction. The buffer is implemented as an array in storage with 6 slots, each slot being large enough to hold a `Tx` object. (It makes programming much easier for the number of slots to be one greater than the capacity of the buffer; in particular this makes it easy to distinguish a full buffer from an empty buffer.)

In the first diagram, the client has added three transactions to the buffer using `push_back`: **Tx 1**, **Tx 2** and **Tx 3**:

`begin()`→	**Tx 1**
	Tx 2
	Tx 3
`end()`→	NOT IN USE
	NOT IN USE
	NOT IN USE

Method `front()` will now return a reference to **Tx 1**, and methods `begin()` and `end()` will return iterators designating the slots indicated in the diagram. So a loop of the form:

```
for (auto it = rb.begin(); it != rb.end(); ++it)
```

should visit **Tx 1**, **Tx 2** and **Tx 3** in turn (and then stop).

In the second diagram, the client has finished processing **Tx 1** and **Tx 2** and has removed them from the queue using `pop_front`:

	NOT IN USE
	NOT IN USE
`begin()`→	**Tx 3**
`end()`→	NOT IN USE
	NOT IN USE
	NOT IN USE

Method `front()` will now return a reference to **Tx 3**.

In the third diagram, four more transactions, **Tx 4**, **Tx 5**, **Tx 6** and **Tx 7** have been added using `push_back`. The buffer is now full.

	Tx 7
`end()`→	NOT IN USE
`begin()`→	**Tx 3**
	Tx 4
	Tx 5
	Tx 6

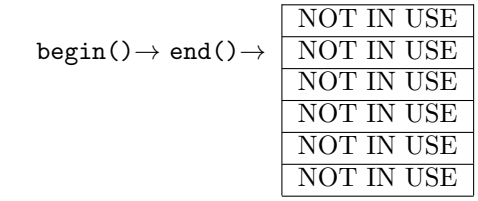
Notice how usage of the `Tx` array wraps around: **Tx 7** is placed in the slot formerly used by **Tx 1**. Conceptually the array represents a *ring* of slots, used in rotation. A loop of the form

```
for (auto it = rb.begin(); it != rb.end(); ++it)
```

*with minor modifications by Radu Grigore

should now visit **Tx 3**, **Tx 4**, **Tx 5**, **Tx 6** and **Tx 7** in turn (and then stop).

In the final diagram, the client has finished dealing with **Tx 3** through to **Tx 7** and has removed them from the queue with `pop_front`:



The buffer is now empty, and the effect of calling method `front()` is undefined.

3 Resources

On `raptor` in the file `/courses/co889/RingBuffer/RingBuffer.hpp` you will find a skeleton implementation of a class template `RingBuffer<T>` along with the associated iterators `RingBuffer<T>::iterator` and `RingBuffer<T>::const_iterator`, which are random-access iterators. (Because everything is templated, all the implementation goes in the header file, so there is no need for a companion file `RingBuffer.cpp`.)

`RingBuffer.hpp` contains three types of comments:

a. Javadoc-like comments, starting with `/**`. (These are processed by the tool `doxygen`.) These comments describe what classes or methods do from the point of view of a client of `RingBuffer`. Many methods lack comments of this kind: `begin()` is an example. In such cases these methods perform their standard function for a container or a random-access iterator, which you can find in the lecture notes. **Hint:** it's worth spending some time thinking what the correct behaviour should be: clever programs implementing the wrong behaviour don't get many marks!

b. Comments starting with `// ***`. These give instructions or constraints that you must follow in preparing your submission for the assessment.

c. Other comments starting with `//`. These explain some details of implementation, or explain some points of syntax that are not covered in the lecture notes.

Also in the directory `/courses/co889/RingBuffer` on `raptor` you will find a file `test_skel.cpp` which you may find useful as a point of departure for your own test programs.

4 Your Task

Your task is to modify the skeleton `RingBuffer.hpp` so as to implement all the methods that are not already implemented for you.

This assignment comes in two flavours:

Baseline: For this you need to implement `RingBuffer<T>` so that it works correctly *when* `T` *is any built-in type* such as `int` or `double`. It is not necessary for your implementation to work correctly when `T` is a class type.

Challenging: For this you need to implement `RingBuffer<T>` so that it works also when `T` is a class type. If `T` is a class type, you may assume that it has a copy constructor, but you may not assume that it has a default constructor, nor that it has an assignment operator.[1] **There are no extra marks for the Challenging version of the assessment.** Nevertheless, I hope some of you will attempt it, though it would be wise to tackle the baseline version first.

To undertake the Challenging version, you will need to find out about the following topics which are not covered in the lecture notes and are (apart from this assessment) beyond the scope of this course:

- `reinterpret_cast` of a pointer. (Do not use C-style casts. Ever.)
- Placement `new`.
- Explicit invocation of a destructor via a pointer.

5 Notes and Constraints

a. You should not modify the file `RingBuffer.hpp` except as indicated in the comments, e.g. by replacing a comment 'Your code goes here' with your code. In particular, do not introduce any additional methods or functions to `RingBuffer.hpp`.

b. Your submission may be evaluated partly by submitting it to a series of automated tests. This means that it is vital that you test your programs thoroughly.

c. The code you add to `RingBuffer.hpp` (at least in the version you submit) should not itself generate any output: this would cause the automated tests to fail. (To test your program you should link it with appropriate `.cpp` files containing your test programs.)

[1] You may find it helpful to use a modified version of `ChattyInt` to test your program.

d. Merit will be given if you avoid duplicating code. So in implementing one method, see if you can do so, at least in part, by calling other methods. (**Hint:** you will find `RingBuffer`'s private method `stepForward` useful for many purposes.)

e. Do not use rvalue-references in your submission.

f. In practice, a ring buffer (or indeed any sort of buffer) may be used with one thread adding transactions to the buffer, and another thread processing (and then removing) the transactions. However, concurrency in C++ is outside the scope of CO889. Do not use any concurrency-related features in your submission.

6 Debugging

See the LZ assessment description for this.

7 If You Need Help ...

...email me (`R.Grigore@kent.ac.uk`), sooner rather than later. Preferably use **plain text** for your mail. Often, for fairness, I will reply *via* a group email, but in doing so I will not reveal your identity.

C++ compiler messages can often be rather longwinded and puzzling. As with all compilers, always concentrate on the very first warning or error message, and try to fix that. But if you're (*a*) still flummoxed and (*b*) using `g++`, then email me the error messages, with the program being compiled as a (plain text) attachment, and I'll try to help.

8 Submission

The assessment is to be submitted on or before Friday 31st January 2019, up to midnight. Submit two files via Moodle: `Ringbuffer.hpp` and `test_skel.cpp`, based on the corresponding provided files. The file `RingBuffer.hpp` should be modified by you according to the above instructions; the file `test_skel.cpp` could be the provided one, but you are encouraged to add tests of your own. You *must* ensure that

```
g++ -std=c++17 test_skel.cpp
./a.out
```

works without errors, on `raptor`. (If you develop your program in an environment other than `raptor`, remember to give yourself plenty of time to check that it compiles correctly on `raptor`, and to *retest your program thoroughly.*)