Task 5. Ring Buffer

Task 5 is devoted to the development of template classes and resource management using the example of the data structure "Ring buffer".

Prerequisites, Goals, and Outcomes

Prerequisites: The students should:

- be able to write custom classes;
- understand the concept of class templates;
- know the basics of resource management in C++;
- understand the principles of the RAII idiom;
- be able to allocate and free memory dynamically using the `new` and `delete` operators;
- understand the design principles of a class *copy constructor*;
- understand how to design a *copy assignment operator* using the Copy-and-Swap idiom.

Goals: This assignment is designed to reinforce the student's skills in designing and implementing a custom template class that manages heap memory resources.

Outcomes: The students successfully completing this assignment will master the understanding of:

- how to implement a class template;
- how to properly handle resources while copying class instances;
- the principles of operation of the data structure “ring buffer”.

Background

One of the most common data structures “queue” implements the FIFO principle of managing elements — first input/first output. Queues are ubiquitous in IT when the number of incoming tasks exceeds the bandwidth of a component processing them. In this case, tasks are buffered, that is, they are queued for execution. The order of their execution must correspond to the order of placing them in the queue. A task that remains in a queue longer than the others must be executed first, as soon as the resources of the processing component are released for that.

The Queue ADT can have different internal implementations, depending on its scope. One such possible implementation is the Ring Buffer. A circular buffer is an array of elements, the end of which is logically connected to the beginning, that is, together they represent a continuous sequence of elements (Fig. 1). In reality, a circular buffer is represented using an ordinary array of a given length, however, when operating with it using a pointer, when the end of the buffer is reached, the pointer moves to the beginning, as if there were no gap between the first and last elements of the array.

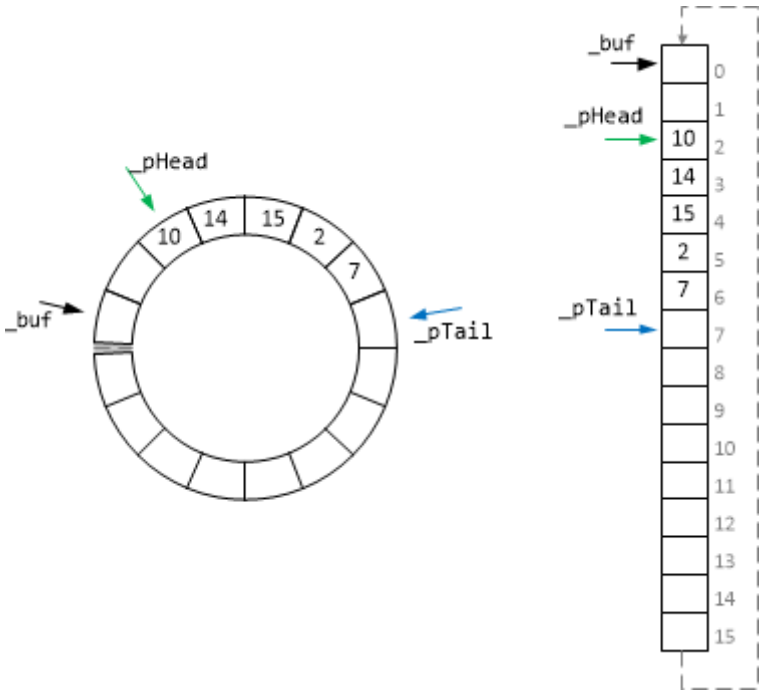

Fig. 1 Ring Buffer and its flat implementation

Ring buffers are especially common in memory-constrained systems and when dynamic dynamic memory allocation is inefficient or undesirable.

The main goal of this project is to implement a queue structure based on a ring buffer for storing elements of an arbitrary data type. To do this, you need to implement the template class `RingBuffer` with a parameter `T`, which determines the type of the stored elements. The buffer elements are stored in a C-style array, the size of which becomes known only at the construction stage of an instance of `RingBuffer` class, so memory for the elements of this array is allocated in the heap. To perform the correct freeing of memory from the array, it is necessary to implement a destructor, and for correct copying of the array a copy constructor and an assignment operation are to be implemented. Auxiliary pointers are used to correctly navigate the circular buffer when adding and retrieving items.

Project structure

The implementation of the `RingBuffer` class template must be put entirely in a header file `ringbuf.hpp`.

Tasks

The development of the class is divided into several stages, at each of which new functional elements are added to the class, expanding its definition.

Begin with the template ringbuf.hpp attached to this assignment. The file contains a header guard and a class template that needs to be populated as you complete stage assignments.

Stage 1. Definition of class fields

The first step is to define the fields that determine the structure of the class. To implement internal storage using a dynamic array, you must define a pointer field of an appropriate type:

```
T* _buf;
```

The size of the array will be passed when creating an instance of the class, and this size must be preserved, for which you need to create a field:

```
size_t _size;
```

To control read and write positions in the buffer, we will create two more pointers:

```
T* _pTail;
T* _pHead;
```

We also need a sign indicating that the buffer is empty:

```
bool _empty;
```

These fields must be properly initialized. To do this, we will define an initialization constructor with a parameter representing the buffer size:

```
RingBuffer(size_t sz);
```

So, after the size of the buffer passed by the constructor parameter becomes known, it is necessary to allocate memory for the array using the `new` operator with square brackets, which indicate the number of array elements:

```
_buf = new T[_size];
```

Also, in the constructor, you need to check that the passed size is not equal to zero, since a zero-length buffer is meaningless. If the parameter defining the buffer size is zero, an exception of the `std::invalid_argument` type must be thrown.

The dynamically allocated memory for storing the array must be correctly freed when the `RingBuffer` object is deleted. To do this, in the destructor, you must use the delete operator with the semantics of square brackets:

```
~RingBuffer()
{
    delete [] _buf;
}
```

Note that the memory is allocated once and is also freed once. Delete cannot be applied to any of the `_pTail` and `_pHead` buffers.

Making a "ring" with `_pTail` and `_pHead` pointers

The `_pTail` and `_pHead` pointers must also be initialized by setting them "in position" of the `_buf` pointer, that is, assigning its value. Note that to allocate memory for `_pTail` and `_pHead` pointers, all three pointers will work with the same buffer, positioned on different elements of it. The `_buf` pointer will always point to the beginning of the array, whereas `_pTail` and `_pHead` will move through the array as they add and remove elements.

The `_pHead` pointer points to the oldest item in the queue, the item to be retrieved next when the `pop()` method is called. The `_pTail` pointer points to the position where the next element will be added (Fig. 1) at the next call to the `push()` method. Note that if the values of these pointers are equal, that is, they point to the same element, then this can correspond to two cases: 1) the buffer is empty, 2) the buffer is full. To distinguish between these two cases, we use the optional `_empty` flag.

Every time when advancing pointers `_pTail` (when adding an element) and `_pHead` (when extracting an element), moving them towards the end of the array, it is necessary to check whether the pointer has gone beyond the upper bound of the array. If so, the value of the pointer is moved to the very first element (the beginning of the array is always accessible using the `_buf` pointer), thereby "looping" the buffer.

This is a preliminary stage, you don't need to upload the results to check them.

Stage 2. Getter method determines the buffer fullness

At this point, you need to add a set of methods that represent the overall internal state of the buffer.

The `getSize()` method should return the size of the memory allocated to be stored in the buffer as a number of elements. This method returns a value equal to the value of the initialization constructor parameter and does not change during the entire lifetime of the object. The possible definition of the method is as follows:

```
size_t getSize() const { return _size; }
```

The `getCount()` method returns the number of items added to the buffer. When adding another element, the value returned by this method is incremented, when extracting, it decreases.

The `getFree()` method returns the number of elements that can be added to the buffer before it is full.

The `isEmpty()` method returns `true` if there are no elements in the method, and `false` otherwise. A possible definition of the method is as follows:

```
bool isEmpty() const { return _empty; }
```

The `isFull()` method returns `true` if the buffer is full, that is, no more elements can be added to it (without extracting the previously added elements), otherwise `false`.

All of these methods do not change the state of the current instance, so they must be marked as constant (for example, see definitions of the methods `getSize()` and `isEmpty()` above).

Most of the methods are trivial to implement. Certain difficulties can arise only when implementing the `getCount()` method. To determine the number of added elements in the buffer (queue size), this method must analyze the exact positions of the `_pTail` and `_pHead` pointers. If they are equal to each other, this means that the buffer is either empty or completely full, and the `_empty` flag comes to the rescue here (and the corresponding `isEmpty()` method that returns its value).

Further, it is necessary to separately check two cases. First, if the value of the `_pTail` pointer is greater than the value of the `_pHead` pointer, we will assume that the pointers are in the "correct" position. Then the number of elements in the buffer is determined as the difference between the values of these two pointers. Second, if the value of the `_pTail` pointer is less than the value of the `_pHead` pointer, then we will assume that the pointers are in the "inverted" state, and the value between them should be calculated accordingly, taking into account the size of the buffer.

The correctness of the implementation of the methods is checked by automatic tests, so you must load the header file with a modified definition of the `RingBuffer` class into the testing system.

Stage 3. The `push()` method

The `push()` method adds another item to the end of the queue if it is not full. If the queue (buffer) is full, the method throws an `std::out_of_range` exception. The method must have exactly the following prototype:

```
void push(const T& value)
```

A new element is added to the position of the `_pTail` pointer, after which it is advanced to the next position in the buffer. If this new position goes beyond the upper bound of the buffer (in terms of element addresses), the `_pTail` pointer is moved to the beginning of the array (which is always accessible using the `_buf` pointer). Adding a new element makes the buffer non-empty, which should be reflected in the value of the `_empty` flag.

The correctness of the implementation of the methods is checked by automatic tests, so you must load the header file with a modified definition of the `RingBuffer` class into the testing system.

Stage 4. The `front()` and `back()` methods

The `front()` and `back()` methods return the oldest element in the queue, that is, the element pointed to by `_pHead`, and the most recently added element, respectively. In the latter case, to access the last added element, it is necessary to go back one position relative to the position of the `_pTail` pointer, taking into account the possible transition of the upper bound of the buffer. Both of these methods must be overloaded for non-const and constant objects:

```
T& front();
const T& front() const;
T& back();
const T& back() const;
```

The correctness of the implementation of the methods is checked by automatic tests, so you must load the header file with a modified definition of the `RingBuffer` class into the testing system.

Stage 5. The `pop()` method

The last of the implemented methods that change the values of the buffer elements is the `pop()` method. This method pops from the queue (and, accordingly, removes from the buffer) the oldest element pointed to by the `_pHead` pointer. In fact, no special actions to "delete" the popped element are needed, it is enough to advance the `_pHead` pointer, checking that it does not go beyond the upper bound of the array. In this case, the position of `_pHead` is adjusted — it is set to the first element of the array, as usual. Before performing actions with pointers, the method must check that the buffer is not empty, otherwise it throws a `std::out_of_range` exception.

If after deleting the next element in the queue (buffer) there are no more elements left, the `_empty` flag should be set.

The correctness of the implementation of the methods is checked by automatic tests, so you must load the header file with a modified definition of the `RingBuffer` class into the testing system.

Stage 6. Copy constructor

The `RingBuffer` class manages low-level resources, i.e. memory, using a pointer. When creating a new instance of a class using the initialization constructor, this memory is allocated correctly by calling the `new T[]` operator. The allocated memory is correctly released in the class destructor using the `delete []` operator. However, when creating a new instance and initializing it with the value of another instance of the class, memory allocation for the buffer of the new instance and copying of elements are not performed, since the default implementation of the copy constructor is called, which performs bit-to-bit copying of the object's field values (*shallow copy*). As a result, two objects begin to share the same memory area, which leads to its incorrect release and, as a result, undefined behavior.

To avoid this, you must correctly implement the copy constructor:

```
RingBuffer(const RingBuffer& other)
```

In the constructor, it is necessary to allocate memory for the buffer array of the new object, copy the value of the size and the `_empty` flag, and also correctly position the `_pTail` and `_pHead` pointers in the new object. To do this, you need to calculate the offset of the `_pTail` and `_pHead` pointers relative to the _buf pointer of the original array, for example:

```
size_t ofs = other._pHead - other._buf;
_pHead = _buf + ofs;
```

The correctness of the implementation of the methods is checked by automatic tests, so you must load the header file with a modified definition of the `RingBuffer` class into the testing system.

Stage 7. Copy assignment operation

Another case of a possible shallow copying of objects of the `RingBuffer` class, leading to an incorrect mutual sharing of resources, is the assignment of the value of one buffer object to another buffer object that was already initialized earlier, and therefore has some value that needs to be overwritten. In this case, it is necessary to correctly free memory from the previous buffer before the memory for the new buffer is reallocated and the elements and other fields of the object are copied. To do this, you need to overload the `operator=`:

```
RingBuffer& operator = (const RingBuffer& rhv)
```

The easiest and safest way to do this with respect to possible exceptions is to use the *copy-and-swap idiom*. To do this, the class must implement a `swap()` function, which exchanges the values of resource handlers, which in the `RingBuffer` class includes all pointers and fields that determine the size and the `_empty` flag. The `swap()` function can be implemented as a static method with two parameters of type `RingBuffer&`:

```
static void swap(RingBuffer& lhv, RingBuffer& rhv)
```

You can use the `std::swap()` standard function to swap the values of the corresponding fields. Note that you cannot apply the `std::swap()` function to `RingBuffer` objects themselves in the implementation of the copy assignment operation, since this will lead to a follow-up call of the copy assignment operation and, as a consequence, infinite recursion.

Submission details

You have to submit a zip file containing only the source code of your application and a `CMakeLists.txt` file. Do not include any auto-generated files. If you use additional resource, include them into `/resource/` subfolder of your zip.