Relazione progetto C++:
buffer circolare

Nome: Mauro
Cognome: Manfredelli
Matricola: 781266
E-Mail: m.manfredelli@campus.unimib.it

Scopo del progetto:
Realizzare una classe cbuffer generica che implementa un buffer circolare di elementi di tipo T. Il buffer deve avere capacità fissa e fornire metodi specifici oltre che a metodi di uso comune.

Classi definite:
Sono presenti due file: cbuffer.h e main.cpp; in cbuffer.h è definita la classe cbuffer con i suoi iteratori (iterator e const_iterator) con la funzione globale check, la classe VoidPosition e la classe IndexOutOfBound per gestire le eccezioni. In main.cpp oltre a metodi di test è definita una struct predicato (utile per il test della funzione globale check).
All'interno dei file ho utilizzato le librerie <ostream> per l'utilizzo di std::ostream nella definizione dell'operatore '<<', <iostream> per l'utilizzo di std::cout per la stampa, <cassert> per l'utilizzo di assert(...) e <stdexcept> per poter utilizzare std::runtime_error; in alcuni casi utilizzerò il lancio di eccezioni, in altri il controllo con assert. Il file main.cpp includerà la classe cbuffer.h per poter definire e utilizzare i buffer circolari.

Classe cbuffer:
La classe cbuffer è templata su un tipo di dato T per permettere la creazione di buffer circolari con tipi di dati generici. Al suo interno è definito il tipo size_type come unsigned int (size, indici, … non possono essere negativi). Sono presenti tre attributi (privati) della classe: T* cb che è il puntatore alla coda FIFO (ossia il cbuffer), size_type size che è la dimensione fissa del cbuffer e

size_type items per tenere il conto di quanti elementi ho inserito nella coda. É inoltre presente un metodo privato shift_left(), utilizzato nell'operazione di cancellazione.
Ho scelto di implementare il cbuffer con un array statico, che inizializzo alla dimensione fissa size che passo come parametro al costruttore; non sarà possibile modificare size dopo la creazione del buffer circolare perchè esso è statico. Tale array lo gestisco come una coda FIFO (first in, first out), ossia il primo elemento che inserisco è il primo che cancello. L'operazione di inserimento corrisponde a una enqueue (metodo insert) dove l'elemento viene inerito in coda (primo posto libero disponibile) e l'operazione di cancellamento a una dequeue (metodo del) che rimuove l'elemento in testa (quello più vecchio). Quando il cbuffer sarà pieno e vorrò inserire un nuovo elemento, cancellerò l'elemento più vecchio (cb[0]) e utilizzerò il metodo privato shift_left() per far diventare cb[1] l'elemento più vecchio (nuovo valore in cb[0]) e 'creare' una posizione libera in coda in cui inserire il nuovo elemento. In questo modo in cb[0] avrò sempre l'elemento più vecchio.
Per sapere se il buffer è pieno uso l'attributo items: se items=size allora il cbuffer è pieno e dunque dovrò cancellare l'elemnto in testa per poter inserire un nuovo elemento in coda (operazione del e inserimento vero e proprio). L'attributo items verrà incrementato ogni volta che effettuo una insert e decrementato ogni volta che effettuo una del: non è presento un decremento esplicito, ma in shift_left quando scambio i parametri temporanei di tmp con quelli della mia classe, items risulta già decrementato.
Per gli attributi size e items ho implementato metodi get per poter leggere il loro valore. Essendo il cbuffer di dimensione fissa, per la size non deve essere presente il metodo set (non modificabile); allo stesso modo per items, che mi serve per tener conto di quanti elementi ho inserito, non ci deve essere un metodo di settaggio (non modificabile).
Per leggere o scrivere nel buffer è possibile usare i metodi get_cbuffer e set_cbuffer.
N.B.:La classe deve consentire sia accesso in scrittura che accesso in lettura.

> Per mantenere la struttura del buffer circolare ottima ho deciso di permettere l'acceso in scrittura e in lettura solo a posizioni del cb in cui è presente un'informazione consistente. In questo modo non sarà possibile scrivere in una posizione qualsiasi, creando dei buchi nel buffer circolare che ne cambiano la struttura.
> Per gestire eventuali accessi a posizioni, diciamo, non disponibili, ho deciso di introdurre delle classi d'eccezione che stampano a video un messaggio (se gestita con un blocco try - cacth) che mi permette di individuare da quale metodo è scaturita l'eccezione e di che tipo di eccezione si tratta, se di accesso 'negato' o index non valido.

La stessa funzionalità di get e set è fornita dal metodo value e dall'opeatore [ ].

Se nella firma del metodo è presente “const T&” come parametro di ritorno e “const” alla fine della firma, allora tale implementazione è usata per la lettura, al contrario, se non sono presenti, per la scrittura.
Per il copy constructor e per l'operatore '=' della classe cbuffer ho implementato due versioni differenti:

- la prima crea uno copia di un cbuffer passato, formato da elementi dello stesso tipo generico T.
- la seconda crea una copia di un cbuffer passato, formato da elementi di tipo generico diverso Q; tali metodi saranno dunque templati su un tipo di dato Q e useranno lo static_cast per convertirli al tipo T (i metodi per questo tipo di implementazione posso lanciare eccezioni).

Fuori dalla classe cbuffer ho definito l'operatore '<<' per la stampa di un cbuffer su stream di output tramite il comando std::cout << .
All'interno della classe cbuffer ho definito due iteratori: iterator e const_iterator. Iterator permette di iterare sul cbuffer permettendo sia la scrittura che la lettura, mentre const_iterator permette di iterare sul cbuffer in sola lettura.
Per costruttori e distruttori ho inserito stampe di debug per la verifica. (per rimuoverle è necessario modificare il makefile).

Classe iteraor:
Tale classe, definita all'interno di cbuffer, è un forward iterator di lettura e scrittura. Posso muovermi solo in avanti, leggendo e scrivendo nelle diverse posizioni. L'ordine con cui itero sugli elementi è dal più vecchio al più nuovo, non posso fare accesso random a una posizione (se si desidera fare un accesso random si possono usare i metodi della classe).
Come attributo ha un puntatore 'ptr' a T che rappresenta l'elemento del cbuffer a cui punta l'iteratore. Il costruttore a cui passo un certo puntatore 'T* p' a cui inizializzare ptr lo definisco privato e sarà usato dai metodi begin() e end() per tornare l'iterator di inizio e l'iterator di fine.
Per l'iteratore di fine tornato dal metodo end() ho ragionato nel seguente modo: dovrebbe tornare l'iteratore alla prima posizione vuota disponibile, ma questo ragionamento è applicabile solo al caso in cui il buffer non è pieno, infatti se il buffer è pieno, end() dovrebbe tornare l'iteratore all'elemento più vecchio che però corrisponderebbe a begin(). Dunque, quando il buffer è pieno, l'iteratore di fine punta a un elemento che non fa parte del buffer circolare, mentre quando non lo è punta alla prima posizione libera. Nel caso di buffer vuoto begin() e end() punteranno alla stessa posizione, infatti non c'è nessun valore su cui iterare.
Al suo interno inserisco anche metodi di confronto tra itearator e const_iterator (posso farlo grazie a friend class 'const_iterator').

Classe const_iterator:

Come iterator è un forward iterator definito all'interno della classe cbuffer, ma è di sola lettura. Stessi attributi di iterator e stessi metodi implementati, compresi i confronti con iteratori di tipi iterator. La differenza sostanziale tra i due è che il const_iterator non deve permettere la modifica dei dati puntati, quindi per 'operator*' e 'operator->' il tipo di dato tornato sarà const T*.
I metodi begin() e end() sono definiti in modo analogo a quelli di iterator, solo che tornano un const_iterator e hanno il modificatore const alla fine della firma.

Struct predicato:

Definisco questa struct nel file main.cpp. Mi serve per definire un predicato (torna true o false) unario per diversi tipi di dati.
Supponiamo di dichiarare un predicato pred: se eseguo pred(x) con x intero verificherà che x sia pari, pred(x) con x double verificherà che x sia maggiore di 5.5 e cosi via, per diversi tipi di dati. Questo comportamento lo implemento con diversi 'bool operator()(...)' all'interno della struct. Questa struct particolare la userò per il test della funzione globale check definita nel file cbuffer.h.

Funzione check:

Funzione gloabale che prende come parametri un generico cbuffer cb (const perchè non deve essere modificato) e un generico predicato unario P. Questa funzione sarà dunque templata su due tipi di dati diversi: T che è il tipo di elementi contenuti nel cbuffer e P che è il tipo del predicato unario che gli passo (io nel test della check definisco 'predicato pred' dunque P corrisponderà a predicato). All'interno del metodo controllerò con un ciclo for che tutti gli elementi di cb rispettino o meno il predicato unario passato.

Classi VoidPosition e IndexOutOfBound:

Queste classi implementate nel file cbuffer.h, servono per il lancio di eccezioni generate durante l'accesso al buffer circolare tramite operator [ ], metodi get, metodi set e metodo value, sia in scrittura che in lettura. VoidPosition è l'eccezione per l'accesso in lettura e scrittura a posizioni nel cbuffer consentite, ma non ancora riempite: non posso scrivere per evitare di modificare la sua struttura e non posso leggere perchè l'informazione contenuta al suo interno non è consistente. IndexOutOfBound è l'eccezione, anche questa sia per accesso in lettura sia in scrittura, per indici fuori dal range [0,1,...,size-1]. Per i metodi insert e del (enqueue e dequeue) non è previsto lancio di eccezioni, ma solo controllo con assert: non posso inserire un nuovo elemento se la size del buffer

circolare è 0 e non posso cancellare se non sono presenti elementi all'interno del cbuffer.
Queste eccezioni devono essere 'catturate' inserendo le operazioni di scrittura e lettura in blocchi try – catch per evitare la chiusura del programma.

Test di verifica:
Nel file main.cpp implemento diversi metodi per verificare il corretto funzionamento della classe cbuffer. Tali metodi utilizzano diversi tipi di dati e stampano a video il risultato delle diverse operazioni di test. Ho inserito un test per ogni funzionalità che ho implementato per capire meglio anche le scelte programmative e implementative che ho fatto.
