CIRCULAR BUFFER

Antonio Briola
806906
a.briola@campus.unimib.it

Università degli Studi Milano Bicocca

INTRODUZIONE

Il progetto proposto richiede la realizzazione di una classe "*cbuffer*" generica che implementa un buffer circolare di elementi di tipo T (*Figura 1*).

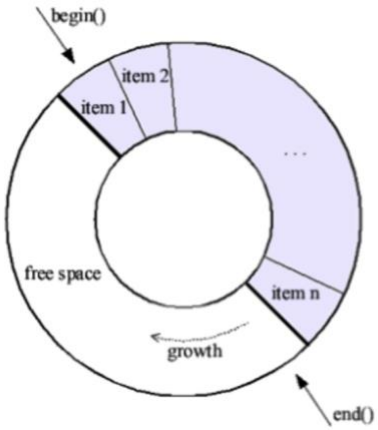


Figura 1

Il buffer ha una capacità fissa, decisa a costruzione. L'inserimento accoda gli elementi finché il buffer non è pieno. Una volta riempito, i nuovi dati vengono scritti partendo dall'inizio del buffer e sovrascrivendo i vecchi.

IMPLEMENTAZIONE

1) Variabili di appoggio

In fase di implementazione, nella sezione private del codice, sono state dichiarate le seguenti variabili:

1) T *_buffer;
2) bool *_is_occupied;
3) size_type _size;
4) size_type occupied_size;
5) size_type _head;

6) size_type _tail;

La prima variabile è un puntatore alla posizione in memoria della prima cella del buffer circolare di elementi di tipo T. La seconda variabile è un puntatore alla posizione in memoria della prima cella di un buffer di booleani di appoggio che tiene traccia di quali celle del buffer circolare contengano effettivamente un valore in un determinato momento del tempo. La terza variabile, di tipo unsigned int, ridefinito per comodità come size_type, tiene traccia della dimensione totale del buffer. La quarta variabile tiene traccia del numero di celle effettivamente occupate nel buffer. La quinta e la sesta variabile, invece, tengono traccia di quali siano rispettivamente testa e coda (per la definizione di testa e coda si legga oltre) del buffer nel tempo.

2) Costruttori

Si è scelto di implementare oltre al costruttore di default, unico utilizzabile nel caso in cui si volesse creare un array di cbuffer, altri quattro costruttori:

1) explicit cbuffer(size_type size)
2) cbuffer(size_type size, const T &value)
3) cbuffer(size_type s, Q begin, Q end)
4) cbuffer(const cbuffer &other)

Il primo costruttore, dichiarato come esplicito al fine di impedire conversioni implicite o "copy-initialization", crea un buffer circolare con un numero di celle pari al parametro ricevuto in input. Come emerge in fase di testing, nel caso in cui il parametro fosse un valore negativo, verrebbe sollevata un'eccezione di tipo std::bad_alloc. Ulteriori altri test sul solo, corrente costruttore sono riportati nel file *main.cpp* e nessuno di essi ha disatteso le aspettative sull'output.

Il secondo costruttore crea un buffer circolare con un numero di celle pari al parametro size (passato per valore), ognuna delle quali contiene il valore di default value (passato per reference). Come emerge in fase di testing, nel caso in cui il parametro fosse un valore negativo, verrebbe sollevata un'eccezione di tipo std::bad_alloc. Ulteriori altri test sul solo, corrente costruttore sono riportati nel file *main.cpp* e nessuno di essi ha disatteso le aspettative sull'output.

Il terzo costruttore crea un cbuffer a partire a partire da una sequenza di dati generici Q identificata da una coppia di iteratori generici. Questo costruttore prende in input: la dimensione del buffer, l'iteratore di inizio sequenza, l'iteratore di fine sequenza. Per testare questo costruttore sono stati seguiti i seguenti passaggi:

1) Creazione di un nuovo buffer circolare vuoto;
2) Inserimento di un primo valore in coda;
3) Inserimento di un secondo valore in coda;

4) Testing del nuovo costruttore.

In seguito ai due inserimenti in coda l'output ottenuto è quello illustrato in *figura 2*.

```
cbuf16 iniziale : 0 0 0 0
cbuf16: 1 2 0 0
*cbuf16.begin(): 1
*cbuf16.end(): 0
```

Figura 2

Si è quindi innanzitutto testato il caso in cui il parametro fosse un valore negativo: in questo caso verrebbe sollevata un'eccezione di tipo `std::bad_alloc`. Si sono poi testati tre differenti casi:

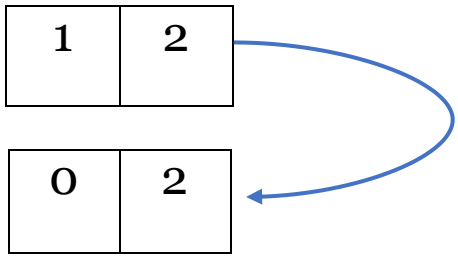
1) Dimensione del nuovo buffer minore della dimensione del buffer di partenza.
2) Dimensione del nuovo buffer uguale alla dimensione del buffer di partenza.
3) Dimensione del nuovo buffer maggiore della dimensione del buffer di partenza.

Nel primo caso otteniamo un output uguale a quello mostrato nella *figura 3*.

```
cbuffer::cbuffer(int)
cbuf16 iniziale : 0 0 0 0
cbuf16: 1 2 0 0
*cbuf16.begin(): 1
*cbuf16.end(): 0
cbuffer::cbuffer(size_type s, Q begin, Q end)
cbuf18 : 0 2
*cbuf18.begin(): 0
*cbuf18.end(): 0
```

Figura 3

Il processo di riempimento per il secondo buffer (`cbuf18`) è il seguente:



Esso è dovuto al fatto che, essendo il buffer circolare, terminato lo spazio, si ha una sovrascrittura delle celle precedentemente utilizzate. Il secondo e il terzo caso non presentano particolarità degne di nota.

Il quarto ed ultimo costruttore crea un buffer circolare a partire da un buffer precedentemente creato e passato come parametro per reference (scelta dovuta al fatto che non possiamo conoscere a prescindere la dimensione del buffer). Tutti i test effettuati su

quest'ultimo costruttore non hanno disatteso le aspettative sull'output e non hanno presentato particolarità degne di nota.

3) Metodi

Partendo dalle premesse precedentemente esplicitate circa la struttura di un buffer circolare diciamo che i primi due metodi implementati degni di nota risultano essere i metodi head() e tail(). Quest'ultimo, in particolare, nell'output, differisce profondamente dall' iteratore end(). Il metodo head() ritorna il contenuto più anziano fra quelli delle celle del buffer circolare. Il metodo tail(), invece, ritorna il contenuto della prima cella libera (o sovrascrivibile) del buffer circolare. L'iteratore begin() punta alla cella di memoria contenete il valore più anziano fra quelli delle celle del buffer circolare. L'iteratore end(), invece, punta all'ultima locazione di memoria scritta (o sovrascritta). Per meglio capire quanto visto fino a questo momento si osservi attentamente la *figura 4*.

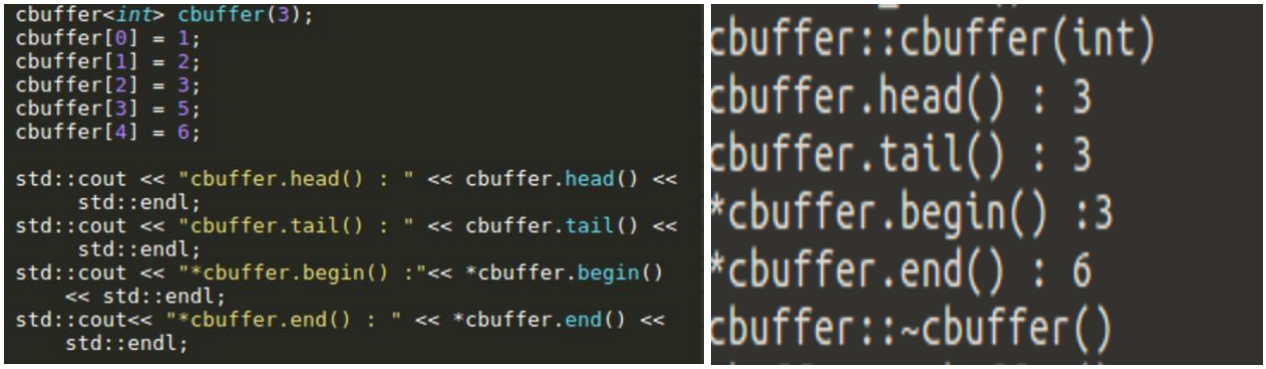


Figura 4

L'esempio sopra riportato permette di riflettere anche sugli altri due aspetti peculiari della mia implementazione: la differenza fra l'utilizzo dell'operatore [ ] e l'utilizzo della insert_in_tail(). Per approfondire l'argomento si osservi la *figura 5*, una rivisitazione dell'esempio precedente.

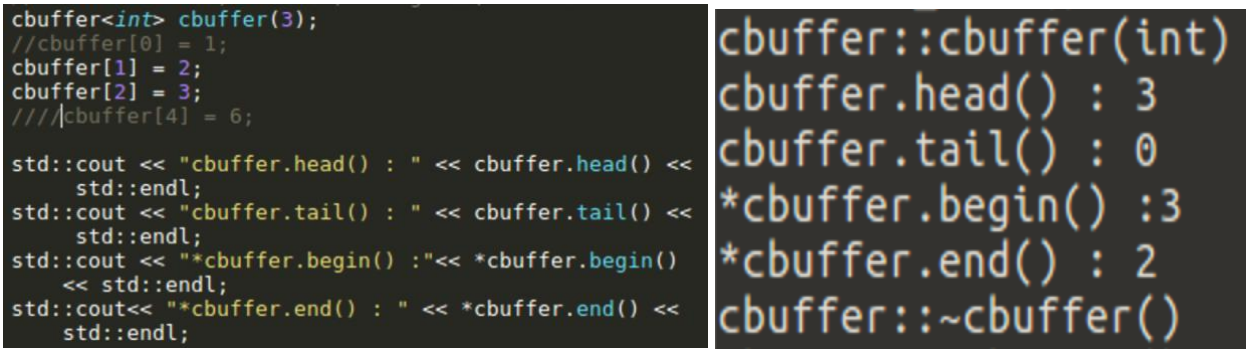


Figura 5

Si noti innanzitutto che il riempimento del buffer circolare attraverso l'operatore [ ] impone l'aggiornamento di volta in volta della testa e della coda della struttura dati. La testa sarà infatti l'ultimo elemento inserito e la coda sarà il primo elemento scrivibile o sovrascrivibile

rispetto alla testa. Questa scelta implementativa è stata effettuata al fine di salvaguardare l'integrità della struttura dati nel caso in cui l'utente decidesse di riempire il buffer circolare "a macchia di leopardo". In base a quanto appena detto il modo corretto e consigliato di riempire la struttura dati sarebbe quello lineare (dalla prima all'ultima cella) attraverso l'utilizzo dell'operatore insert_in_tail(). Questa scelta farebbe sì che a buffer completamente pieno head e tail coincidano mentre l'iteratore begin() punti all'elemento più anziano del buffer e l'iteratore end() punti all'ultimo elemento del buffer; l'inserimento di un ulteriore elemento andrebbe a sovrascrivere la head e a far slittare head e la tail di una posizione in avanti spostando l'iteratore end() sull'ex head (*figura 6*).

```
cbuffer<int> cbuffer1(3);
cbuffer1.insert_in_tail(1);
cbuffer1.insert_in_tail(2);
cbuffer1.insert_in_tail(3);
cbuffer1.insert_in_tail(4);

std::cout << "cbuffer1 after many insertions in tail
    : " << cbuffer1 << std::endl;
std::cout << "cbuffer1.head() : " << cbuffer1.head()
    << std::endl;
std::cout << "cbuffer1.tail() : " << cbuffer1.tail()
    << std::endl;
std::cout << "*cbuffer1.begin() :"<< *cbuffer1.begin(
    ) << std::endl;
std::cout<< "*cbuffer1.end() : " << *cbuffer1.end() <
    < std::endl;
```

```
cbuffer::cbuffer(int)
cbuffer1 after many insertions in tail : 4 2 3
cbuffer1.head() : 2
cbuffer1.tail() : 2
*cbuffer1.begin() :2
*cbuffer1.end() : 4
cbuffer::~cbuffer()
```

Figura 6

Vorrei infine porre l'attenzione su di un ultimo metodo implementato : set_value(size_type index, const T &value). Esso è stato sviluppato al fine di permettere all'utente di modificare il contenuto di una cella del buffer circolare senza modificare head, tail, begin ed end della struttura dati.

Ogni altro metodo sviluppato è dettagliatamente descritto nella documentazione fornita in fase di consegna. Ulteriori test di unità e di sistema sono stati effettuati su ogni funzione implementata e nessuno di essi ha disatteso le aspettative.

NOTE INTEGRATIVE E OUTPUT

La compilazione dei sorgenti, la pulizia della cartella e la creazione della documentazione con doxygen possono essere eseguite utilizzando il make come specificato di seguito:

1) make : compilazione di default dei sorgenti in modalità debug; informazioni di debug complete in fase di esecuzione dei test e codice di debug per testare l'integrità della classe.
2) make exec : esecuzione del file .exe.
3) make valgrindexec : esecuzione del file .exe tramite Valgrind per il controllo dello stato di memoria.
4) make doxyhtml : generazione della documentazione a partire dal file di configurazione doxygen.
5) make clean : rimozione dei file oggetto e dell'eseguibile creato.

Il progetto è stato testato utilizzando g++ versione:

g++ (Ubuntu 5.4.0-6ubuntu1~16.04.6) 5.4.0 20160609

Ogni singola esecuzione del progetto è stata testata sotto Valgrind versione:

valgrind-3.14.0.GIT

Tale versione (quella meglio mantenuta) è stata appositamente scelta in quanto funzionante anche sotto QtCreator. Ogni esecuzione dell'intero progetto sotto Valgrind non ha riportato alcun memory leak. Emergono invece molti errori di tipo conditional jump dovuti all'accesso ad aree di memoria istanziate ma non inizializzate: tale scenario si è reso necessario al fine di creare una classe di test il più ampia e articolata possibile.