

# CBuffer

---

**Lorenzo Soligo - 806954 - Febbraio 2018**

---

## Prima di tutto: compilazione

- `make doc` per creare la documentazione con Doxygen
- `make` per compilare il progetto C++
- `make debug` per compilare il progetto con stampe di debug e flag `-g`
- `make run_valgrind` per compilare il progetto con stampe di debug e flag `-g` ed eseguire valgrind con la flag `--leak-check=yes`

## Introduzione

La traccia del progetto richiede di scrivere un buffer circolare di dimensione data in fase di costruzione. In poche parole, bisogna aggiungere elementi al buffer "finché c'è spazio", e successivamente bisogna sovrascrivere gli elementi in ordine di inserimento crescente (i.e. dal più vecchio al più nuovo).

## Scelte implementative

### Implementazione del cbuffer

Il progetto, alla luce di quanto studiato durante il corso, si prestava a due diverse implementazioni:

1. nodi concatenati
  - si definisce una `struct node` templata, contenente il valore e il puntatore al nodo `next`
2. array di tipo `T` templato

L'implementazione da me scelta è la prima, in particolare perché la trovo più facilmente riutilizzabile ed estendibile. Inoltre, sfruttando i `node`, possiamo posticipare l'allocazione della memoria fino a quando questa non sarà certamente necessaria; a scapito di una potenziale perdita di località dovuta a puntatori a `next`, abbiamo un vantaggio non indifferente in caso venga istanziato un `cbuffer` di grosse dimensioni e vengano inseriti solo pochi elementi. Tutto sommato, comunque, entrambe le implementazioni risultano valide, e la scelta di una in contrapposizione all'altra appare essere prevalentemente una semplice questione di gusti.

### Implementazione di base

Di seguito, le mie scelte progettuali per quanto riguarda il cuore della classe, con giustificazioni a riguardo.

- **struct node:**
  - il cbuffer si appoggia ad una `struct node` molto semplice. Questa consiste in:
    - un puntatore ad un altro `node`, il `next` del nodo attuale
    - valore templato `T value`
    - costruttore di default
    - costruttore dati `value` e `next` [quest'ultimo, impostato di default a NULL]

- **campi della classe:**

- ho scelto di utilizzare 4 campi:
  - `_size`, la dimensione del cbuffer
  - `_occupied`, il numero di elementi occupati del cbuffer
  - `node* _head`, il puntatore alla testa del cbuffer
  - `node* _tail`, il puntatore alla coda del cbuffer
- la scelta è dovuta al fatto che questi campi sono sufficienti a svolgere tutte le operazioni necessarie e si prestano bene alla manipolazione del cbuffer (inserimento in coda, rimozione della testa, ...)

## Implementazione di metodi, operatori, iteratori, ...

### 1. operator[]

- ho deciso di implementare l'`operator[]` nel seguente modo:
  - l'accesso è consentito solo alle cellette del cbuffer che sono già state istanziate. Ho inoltre implementato due `operator[]`, uno `const` per la lettura e uno standard in scrittura.  
È quindi consentita la *modifica* di cellette esistenti, ma non l'inserimento di nuove cellette, che richiede l'utilizzo di `insert`.

### 2. iteratori

- sono stati implementati sia `const_iterator` che `iterator`. Questo per consentire sia un accesso read/write, che un accesso in sola lettura, quando non necessaria la scrittura.
- tramite una flag booleana `first_time` si riesce ad iterare normalmente con un ciclo for, come ci si aspetterebbe. Sia l'iteratore d'inizio che quello di fine corrispondono al nodo `_head`, ma la variabile consente di far fallire il primo controllo *begin* ≠ *end*

### 3. costruttori

- sono stati implementati vari costruttori:
  1. costruttore di default, `cbuffer()`
    - questo costruttore non è particolarmente utile, in quanto viene inizializzato un cbuffer di dimensione 0. L'unico motivo per utilizzarlo è: istanziare un cbuffer vuoto ed utilizzarlo per copiarci dentro un altro cbuffer tramite `operator=`
  2. costruttore secondario data la size, `cbuffer(unsigned int size)`
  3. costruttore copia `cbuffer(const cbuffer &other)`
  4. costruttore dati `size`, iteratore d'inizio, iteratore di fine
    - la conversione dei dati viene lasciata al compilatore. Nel caso vi sia una qualsiasi eccezione, il cbuffer viene svuotato e l'eccezione viene ritornata al chiamante
    - è lasciata all'utente l'accortezza di passare una size sensata, ovvero di non dare (per esempio) un iteratore di 5 elementi e `size = 10`.

### 4. insert

- ho scelto di implementare la insert in modo che, nel caso non si riesca ad allocare la memoria, il cbuffer venga svuotato e l'eccezione venga ritornata al chiamante. Questo perché mi sembra

insensato lasciar proseguire l'esecuzione se non c'è neanche spazio sufficiente ad inserire un nuovo elemento nel cbuffer.

#### 5. `evaluate_if`

- il metodo è stato implementato come da specifica. Viene fatto uso dell'operatore condizionale `?` per comodità.

#### 6. `clear`

- il metodo `clear` è reso pubblico al fine di rendere facile per l'utente svuotare il cbuffer

#### 7. `get_node`

- è un metodo privato a cui si appoggia `operator[]`

#### 8. `size`, `occupied`

- danno, rispettivamente, dimensione e numero di cellette occupate

## Test

Con (non molta) fantasia, ho nominato i test:

- `test buono`: istanzia un cbuffer di interi vuoto e verifica che le operazioni `insert` e `remove_head` non facciano danni.
- `test non molto buono`: istanzia un cbuffer di `std::string` e lo maltratta con `insert`, `operator[]` e `remove_head`. Numerose assert verificano che tutto avvenga come dovrebbe.
- `test costruttore iteratori`: come dice il nome, costruisce il cbuffer a partire da un iteratore, in particolare un array di caratteri, ed esegue le solite operazioni basilari.
- `test cbuffer di cbuffer di voci`: istanzia un cbuffer di `cbuffer<voce>` ed esegue le normali operazioni. L'assenza di memory leaks anche in questo test fa ben sperare, così come la stabilità dell'`operator[]` anche in presenza di una `struct`.
- `test costruttore copia`: autoesplicativo, utilizza il costruttore copia su un po' di cbuffer di vari tipi diversi.
- `test evaluate if`: testa il metodo `evaluate_if` su dei cbuffer di varie strutture, con vari funtori.
- `test operator quadre cattivo`: testa a fondo `operator[]`
- `test clear poi riempi`: chiama `clear` su un cbuffer e lo riempie nuovamente

## Informazioni varie

- in fase di compilazione e testing, sono state utilizzate le flag `-Wall -Wextra -pedantic` di `g++` ai fini di porre particolare attenzione alla conformità agli standard e di non dimenticare nulla
- svariati test nel file `main.cpp` sfruttano `assert` per verificare la corretta esecuzione del tutto ed, eventualmente, stampare una stringa d'errore significativa, senza dover leggere decine di righe sullo standard output
- in fase di sviluppo è stato utilizzato `git`, in particolare tramite la nota piattaforma GitHub. La repository è ora pubblica @ [github.com/lollones/cbuffer](https://github.com/lollones/cbuffer). Ho inoltre avuto modo di provare il sistema di *continuous integration* Travis-CI per eseguire test di compilazione automatici.