Relazione di progetto C++ del 09/02/2025

Cattaneo Francesco – matricola 90041 – f.cattaneo39@campus.unimib.it

Introduzione

Viene implementato un array dinamico ordinato generico `SortedArray`, che utilizza i template per supportare qualsiasi tipo di dato `T`. L'ordinamento dell'array è scelto liberamente dall'utente.
I in ogni istante, la dimensione dell'array è corrispondente al numero degli elementi effettivamente inseriti.
La classe supporta gli iteratori di tipo `random_access`, e nel `main.cpp` del progetto sono presenti tutti i test per verificare il corretto funzionamento della struttura.

Struttura della classe

Per garantire flessibilità nell'ordinamento degli elementi, `SortedArray` utilizza un comparatore personalizzabile, definito come membro privato della classe. Alla classe sono stati assegnati due template: T, il quale sarà uguale al tipo del dato dell'array, e Comparator, il quale rappresenta il tipo di funtore che viene usato per ordinare l'array. Quest'ultimo deve essere un funtore un oggetto che implementa l'operatore `()`.
Grazie ai template, la classe può funzionare con qualsiasi tipo di dato che supporta gli operatori di comparazione `operator<` ed `operator==`, e che supporta l'operatore di assegnamento `operator=`.

Per quanto riguarda i costruttori, sono disponibili i seguenti:

- **Costruttore di default**, che inizializza un array vuoto
- **Costruttore con comparatore**, che si comporta come il costruttore di default, ma dove si può specificare un comparatore personalizzato

- **Costruttore di copia**, che consente di creare una copia di un `SortedArray`, anche con tipi di dato differenti e con un comparatore diverso

La classe inoltre presenta alcune funzioni aggiuntive per l'accesso e la manipolazione dei dati, che potrebbero essere utili in una struttura dati come un array:

- `top()` restituisce il primo elemento, `_end()` restituisce l'ultimo elemento
- `top()` e `end()` restituiscono iteratori agli estremi dell'array
- `size()` restituisce la dimensione dell'array
- `isEmpty()` indica se l'array è vuoto
- `contains(const T& value)` verifica la presenza di un elemento value all'interno dell'array
- `clear()` svuota l'array

`insert()` e `remove()` sono stati implementati come richiesto dalle specifiche.
Ho scelto di usare il comparatore per determinare la posizione corretta per l'inserimento del nuovo elemento. Quindi, copia tutti gli elementi presenti nell'array fino all'indice trovato al passo precedente, inserisce il nuovo valore, e continua ad copiare il resto degli elementi. Infine, aggiorna la dimensione dell'array.
`remove()` si comporta in maniera più basilare: prima conta quante volte è presente, poi alloca un nuovo array di dimensione ridotta e copia solo gli elementi diversi da quello da rimuovere. Se l'array diventa vuoto, viene svuotato con clear().
Sono presenti due funzioni di rimozione, aggiuntive, e private, che servono come funzioni helper per le funzioni di modifica degli elementi:

- `_remove(size_t index)` rimuove l'elemento all'indice dato, e reduce la dimensione dell'array. Visto che si parte già da un array ordinato, quando si elimina un elemento in questa maniera non è necessario riordinare l'array
- `removeOne(const T& remove)` funziona come remove(), ma elimina soltanto la prima occorrenza dell'elemento che si vuole eliminare

Con il seguente array è possibile anche modificare direttamente gli elementi all'interno dell'array, pur mantenendo l'ordine degli elementi. Per farlo, il valore da sostituire viene prima

rimosso con una funzione di rimozione, e poi inserito il nuovo valore usando insert(). Sono presenti tre varianti della funzione di modifica:

- **`edit(size_t index, const T& newValue)`** modifica un elemento restituendo il valore sostituito. Usa la funzione helper **`_remove()`** per rimuovere l'elemento presente prima della sostituzione all'indice dato.
- **`editPre(size_t index, const T& newValue)`** modifica un elemento restituendo il valore prima della sostituzione.
- **`_edit(const T& oldValue, const T& newValue)`** modifica tutte le occorrenze di un valore. Usa la funzione helper removeOne, per evitare di avere comportamenti indesiderati con l'array.

Per facilitare la scrittura dei test inoltre, è stato sovraccaricato operator<<, che permette la stampa di tutti gli elementi contenuti nell'array.

Supporto agli iteratori

La classe supporta iterazione tramite una classe interna `const_iterator`. È stata implementata in questo modo per garantire che questi iteratori possano essere usati esclusivamente con `SortedArray`. Inoltre, sono stati implementati solo iteratori costanti per permettere all'array di essere ordinato anche durante l'iterazione.
Per quanto riguarda la gestione degli errori, gli iteratori verificano innanzitutto la validità dell'array prima di accedere agli elementi, e in secondo luogo, viene verificato anche se si tenta di accedere a indici validi con gli iteratori. In entrambi i casi, se si tenta di accedere ad un array invalido, o ad indici invalidi, vengono lanciate le apposite eccezioni.
L'operatore di dereferenziazione controlla se il puntatore è valido prima di restituire il valore.

Gestione delle eccezioni

Come detto prima, il controllo degli indici è effettuato non solo nella classe `const_iterator`, ma anche per tutte le altre funzioni in `sortedArray` che accedono ad elementi tramite indici. Quindi, prima che questo venga effettuato, viene effettuato un controllo per verificare che l'indice sia compreso nell'intervallo valido, e se non lo è, viene sollevata un'eccezione di tipo `std::out_of_range`.

Operazioni su array vuoti, come accedere al primo o all'ultimo elemento, vengono controllate sollevando eccezioni per evitare accessi non definiti.

L'autoassegnamento viene prevenuto nell'operatore di assegnamento, per evitare operazioni inutili o comportamenti indesiderati.

In tutte le operazioni che comportano l'allocazione dinamica di memoria, se un'eccezione viene sollevata durante l'allocazione o la copia dei dati, la memoria precedentemente allocata viene rilasciata prima di propagare l'errore, per garantire che non ci siano memory leaks, e che `SortedArray` rimanga in uno stato consistente.

Test della classe

Nel file `main.cpp` sono stati implementati diversi test, ognuno dei quali segue la stessa struttura:

1. Viene stampata a schermo una breve definizione di ciò che verrà verificato
2. Vengono stampate tutte le informazioni utili a comprendere come viene svolto il test
3. Dopo l'operazione testata, viene mostrato a schermo tutto ciò che è necessario per capire se la classe si sta comportando come previsto
4. Una serie di `assert` controlla a runtime se il comportamento effettivo corrisponde a quello atteso

Con questa serie di passaggi, viene mostrato a schermo tutto quello che è necessario per capire il comportamento della classe in certi scenari, e grazie agli `assert()`, si può verificare se il comportamento che si manifesta è quello atteso. Si è cercato di testare tutte le funzionalità della classe, in tutti i casi possibili e/o utili da verificare.

I test sono suddivisi nei seguenti gruppi:

- **Test dei costruttori**
    - Costruttore di default: verifica che l'array venga inizialmente creato vuoto
    - Costruttore di copia: verifica il funzionamento della copia tra due array in diverse situazioni: stesso tipo di dati e stesso comparatore, tipo di dati diverso e

stesso comparatore, stesso tipo di dati e comparatore diverso, tipo di dati diverso e comparatore diverso. Questo permette di testare la correttezza del costruttore in tutti i casi possibili.
    - Costruttore di copia con funzione di comparazione complessa
    - Costruttore di copia con tipi complessi e diversi iteratori
    - Costruttore da un intervallo di iteratori generici
- **Test dell'operatore di assegnamento**
    - Assegnazione tra array: verifica il comportamento dell'assegnazione in diversi scenari: stesso tipo di dati e stesso comparatore, tipo di dati diverso e stesso comparatore, stesso tipo di dati e comparatore diverso, tipo di dati diverso e comparatore diverso
    - Assegnazione con tipi composti e comparatori diversi
    - Autoassegnamento
- **Test con tipi base**
    - Inserimento di elementi nell'array
    - Accesso agli elementi: accesso a un elemento valido, accesso a un elemento fuori dai limiti, uso delle funzioni `top()` ed `_end()`
    - Ricerca di un elemento nell'array con `contains()`
    - Rimozione di elementi: rimozione di un elemento presente una sola volta, rimozione di elementi presenti più volte, tentativo di rimuvoere un elemento non presente
    - Funzione `clear()` e `isEmpty()`, per svuotare l'array e verificare che sia effettivamente vuoto
    - Accesso ad elementi in array vuoti
    - Modifica di elementi: modifica di un elemento dato un indice, modifica di tutte le occorrenze di un elemento nell'array, tentativo di modifica di un elemento in un indice non valido, tentativo di modifica di un elemento non presente nell'array.
- **Test con tipi complessi**
    - Gli stessi test eseguiti sui tipi base vengono ripetuti con il tipo `Point`, per verificare il corretto funzionamento con tipi personalizzati.
- **Test della funzione di filtro**

    - Viene testata la funzione di filtro usando diversi funtori.
- **Test del `const_iterator`**
    - Costruttore di default e costruttore di copia
    - Assegnazione di un iteratore a un altro
    - Accesso al primo e all'ultimo elemento dell'array
    - Operatore `->`
    - Pre-incremento/post-incremento e pre-decremento/post-decremento dell'iteratore
    - Operatore di uguaglianza e disuguaglianza
    - Spostamento dell'iteratore di una certa distanza all'interno dell'array
    - Calcolo della differenza tra due iteratori.