Relazione Progetto C++ settembre 2022

Nome: Cristian
Cognome: Piacente
Matricola: 866020
Mail: c.piacente@campus.unimib.it
Progetto: Multiset ordinato

STRUMENTI UTILIZZATI

Nella realizzazione del progetto ho utilizzato:

- **W**indows **S**ubsystem for **L**inux, versione Ubuntu 20.04.4 LTS;
- **GCC** (compilatore G++) **9.4.0**, target x86_64-linux-gnu;
- GNU Make 4.2.1;
- Valgrind versione **valgrind-3.19.0**.

Per quanto riguarda il processo di compilazione, nel makefile utilizzo il flag -Wall per stampare messaggi di warning e -std=c++0x per utilizzare C++11 (per il fatto che alcuni compilatori non recenti potrebbero non riconoscere nullptr).

Dopo aver terminato il progetto ho anche testato su:

- **Macchina virtuale** del corso, su cui vi sono installati l'OS Ubuntu 20.04.4 LTS, GCC versione 9.4.0, GNU Make 4.2.1, Valgrind versione valgrind-3.15.0;
- **Windows 10** Home **64-bit** versione 21H2 (con GNU Make 3.81 e compilando da cmd), alcune note:
  - all'inizio utilizzando la versione di MinGW suggerita su e-learning (GCC 11.2.0) all'esecuzione dell'eseguibile avevo un problema di punto di ingresso _ZSt28__throw_bad_array_new_lengthv
    - ho risolto scaricando **MinGW con GCC versione 9.4.0** (in particolare, MinGW-W64 i686-posix-dwarf), target i686-w64-mingw32, così ho potuto testare anche su ambiente Windows;
  - non ho testato con DrMemory, avendo già utilizzato Valgrind su Linux.

SCELTE IMPLEMENTATIVE SULLA STRUTTURA DATI

Per l'implementazione del Multiset ordinato (insieme che può contenere duplicati), dalla traccia è stato richiesto di minimizzare l'uso della memoria non memorizzando i duplicati di un elemento. Poiché è necessario fare il minimo uso della memoria per la struttura dati, ho deciso di definire una **struct interna**, non visibile all'esterno della classe, che mi rappresenta una coppia **<valore, occorrenze>** (in particolare questa struct chiamata multi_element possiede solo questi due dati membro) e di rappresentare i dati contenuti nel Multiset tramite un **array dinamico di questa struct**. Utilizzo un array e non una struttura dati differente come una lista proprio per garantire il minimo uso della memoria, poiché utilizzando una struttura a lista avrei bisogno di un puntatore next per ogni "nodo" (struct), che non rispetterebbe i requisiti del progetto.

Dopo questa prima importante osservazione, continuando a leggere il testo è stata lasciata libera scelta sulla categoria di iteratore di sola lettura: decido di utilizzare un **iteratore bidirectional**. Questa scelta è stata fatta perché innanzitutto non ha alcun senso l'iteratore di categoria random access, perché non si tratta ad esempio di un buffer in cui è possibile usare l'operator[] per accedere tramite indice, e poi non ho utilizzato un iteratore forward solo perché la struttura dati è ordinata e quindi ha senso potersi chiedere quale elemento precede un altro, cioè poter decrementare l'iteratore per "tornare indietro".

Nota: da una prima lettura della traccia non avevo capito che deve essere possibile utilizzare l'operator== tra Multiset ordinati con policy di ordinamento diverse, e che deve restituire true nel caso i dati (elementi e molteplicità) siano gli stessi ma ordinati in modo diverso, perché pensavo che in Multiset ordinati fosse fondamentale l'ordine non solo per l'inserimento ma anche per il confronto con l'operatore di uguaglianza; controllando sul forum e rileggendo la traccia in effetti non c'è scritto che l'ordine dev'essere lo stesso, quindi ho **templato l'operator==** per permettere il confronto con un Multiset ordinato con un funtore di ordinamento diverso (e quindi accedo tramite l'interfaccia pubblica poiché si tratta di due tipi di dato diversi, siccome cambia un parametro generico).

CLASSE ORDERED_MULTISET: BREVE DESCRIZIONE

La classe **ordered_multiset** implementa appunto un Multiset ordinato di elementi di tipo generico T, in cui l'utente è libero di scegliere la **policy di ordinamento** tramite un funtore Cmp e sempre l'utente definisce anche l'**uguaglianza tra due elementi di tipo T** tramite un altro funtore Eql, poiché ad esempio potrebbe non essere definito l'operator== per un certo tipo custom.

Oltre ai 4 metodi fondamentali, l'utente è in grado di svuotare tutto il contenuto del Multiset (metodo **clear**), scambiare un Multiset con un altro dello stesso tipo (metodo **swap**), sapere quanti elementi contiene in totale il Multiset (metodo **size**), aggiungere un'occorrenza di un elemento (metodo **add**), rimuovere una singola occorrenza di un elemento (metodo **remove**, lancia un'eccezione di tipo **element_not_found** se non si rispetta la pre-condizione di avere almeno un'occorrenza dell'elemento passato come argomento), sapere il numero di occorrenze per un certo elemento (metodo **multiplicity**), confrontare con l'**operatore di uguaglianza** un Multiset con un altro Multiset che ha tipi di dato e policy di uguaglianza uguali ma che possono essere ordinati in modo diverso, sapere se un elemento è presente (metodo **contains**), ottenere iteratori di categoria bidirectional di sola lettura di inizio e fine sequenza (metodo **begin** e **end**) e infine può inviare un Multiset a uno stream di output tramite **operatore di stream**.

Dettagli implementativi per ciascun metodo sono presenti in documentazione.

PROBLEMATICHE RISCONTRATE

Per quanto riguarda la scrittura di codice per la classe, a parte il dubbio sull'operator== di cui ho parlato prima, i metodi su cui mi sono focalizzato di più sono stati add e remove, mentre per il const_iterator operator++, operator-- e operator==.

I metodi add e remove sono "delicati" perché riguardano l'allocazione e la deallocazione dell'array dinamico di struct: per evitare recovery degli errori lavoro con dati automatici. In particolare, all'inizio pensavo di crearmi prima di tutto un nuovo array temporaneo e poi di lavorarci, ma

questo richiedeva un try-catch per evitare memory leak (in quanto l'array temporaneo è allocato sullo heap e non viene distrutto da solo), quindi ho deciso di crearmi un dato automatico di tipo ordered_multiset e assegnare il nuovo array temporaneo a questo Multiset temporaneo: in questo modo se venisse generata un'eccezione allora il Multiset uscirebbe di scope e verrebbe invocato in automatico il distruttore che pensa alla deallocazione dell'array, evitando memory leak. Solo a fine metodo effettuo una swap tra this e il Multiset temporaneo, lasciando i vecchi dati sul dato automatico che viene distrutto a fine scope.

Ho riscontrato problemi soprattutto sull'incremento e il decremento dell'iteratore, in quanto è necessario che gli elementi vengano "restituiti" con le molteplicità corrette: questo significa che il puntatore a multi_element deve rimanere "fermo" sullo stesso multi_element finché non si esauriscono tutte le occorrenze. Per fare ciò utilizzo un secondo dato membro che mi tiene il conto di quante occorrenze rimangono verso destra. I problemi riguardavano soprattutto accesso a memoria non inizializzata, ad esempio per quanto riguarda l'incremento nel momento in cui costruivo l'iteratore di fine sequenza andavo subito a leggere il numero di occorrenze dalla struct, oppure nel momento in cui mi spostavo al successivo andavo a leggere dalla struct, ma quando incremento l'iteratore che si riferisce all'ultimo elemento non posso poi andare a leggere il numero di occorrenze. Questo problema è stato risolto utilizzando un valore segnaposto (0) e ho descritto in documentazione come funzionano l'incremento e il decremento di un iteratore.

Infine, ho dovuto considerare un corner case per l'operatore di uguaglianza, che riguarda i contatori dati membro degli iteratori: se confronto due iteratori e questi hanno puntatore a multi_element uguale e hanno come contatore di occorrenze rimanenti 0 e 1 (o viceversa) allora possono essere sia uguali sia diversi, a seconda della molteplicità dell'elemento a cui si sta facendo riferimento. Infatti, se abbiamo una sola occorrenza allora i due iteratori sono uguali, diversi altrimenti. Anche questo problema è stato descritto, con due esempi, in documentazione.

FASE DI TESTING

Questa è la **fase a cui ho dedicato più tempo**. All'inizio del main.cpp è possibile **scegliere** se inviare tutto l'output di ciascun test per ciascun tipo su **file** oppure su **console**: se si desidera stampare su file si può scegliere la directory di output alla **riga 39** (ATTENZIONE: la directory deve esistere e il percorso terminare con lo slash, di default il percorso è "./tests_output/" e infatti nella consegna ho lasciato una cartella tests_output con tutti gli output generati dai test). Se è questo il caso, per ciascun tipo di dato testato verrà creato un file .txt contenente tutte le stampe ed è necessario che la riga 35, che definisce il tag **OUTPUT_TO_FILE**, non sia commentata. Per la consegna lascio di default abilitata la stampa su file, poiché non è molto leggibile direttamente da console, nonostante abbia aggiunto dei colori e il grassetto per terminali Linux; se invece si desidera stampare su console è sufficiente commentare la riga 35.

Dopo la definizione delle macro per le stampe con colori per console Linux, ho definito dei **metodi di comodo** (funzioni globali) che mi servono per effettuare i test:

- la conversione di un intero nel corrispondente intero positivo modulo 4 (per test in Z_4)
- la trasformazione di una stringa in uppercase
- il confronto case insensitive tra stringhe
- la stampa [Test #*numero*] colorata che restituisce std::cout perché la richiamo a inizio riga e così posso concatenare altri dati con l'operatore di stream
- la conversione da bool a stringa colorata

- un metodo che ritorna un OK! colorato, utilizzato tipicamente dopo una assert
- un metodo che rimuove tutti i caratteri null terminator da una std::string, poiché se questi vengono aggiunti allora si va a creare una stringa diversa, mi serve per i test con l'elemento di default per il tipo char (ossia l'elemento restituito dal pseudo-costruttore di default)
- un metodo che a partire da un Multiset e una stringa controlla se inviando il Multiset a uno stringstream si va a creare una stringa uguale a quella passata come argomento, per vedere se l'operatore di stream ci dà il risultato atteso
- il conteggio delle occorrenze di un elemento generico di tipo T a partire da una coppia di iteratori di inizio e fine sequenza
- un metodo che estrae gli elementi distinti in ordine a partire da due iteratori, senza utilizzare un Multiset (altrimenti poi non avrebbe senso il test)
- un metodo che prende un std::vector e restituisce una stringa corrispondente a ciò che ci si aspetta inserendo quei dati in un Multiset e usando l'operator<<
- un metodo che prende un vector e ne riempie un altro utilizzando o gli indici pari o gli indici dispari, a seconda di un altro argomento
- un metodo importante che a partire da un vector restituisce una stringa corrispondente a quello che dovrebbe generare un Multiset con quei dati inseriti: ciò mi consente poi nel metodo templato per i test di non passare alcuna stringa come argomento poiché viene tutto generato in automatico
- un metodo che a partire da un Multiset riempie un vector con solo le molteplicità di ciascun elemento
- un metodo che rimuove un singolo elemento da un vector, a partire dal fondo per evitare problemi testando con Z_4 (insieme delle classi di resto modulo 4, per il Multiset è importante l'ordine di inserimento degli elementi poiché si vanno a generare stringhe diverse in base a quale elemento distinto è stato inserito per primo, questo accade perché utilizzo l'overloading dell'operator<< per int per i numeri modulo 4, quindi se ad esempio inserisco -1 e 3 in questo ordine allora nel Multiset avrò una coppia <-1, 2>, altrimenti <3, 2>)
- un metodo che eventualmente setta il buffer di std::cout con quello di un ofstream ottenuto aprendo un file e che, se si stampa su file, restituisce un puntatore al buffer originale di std::cout che poi mi servirà per il ripristino, altrimenti restituisce nullptr.

Dopo i metodi di comodo ho definito una struct che mi impacchetta 4 dati: questi riguardano i dati passati alla mia funzione templata (di cui parlo in seguito) che esegue una serie di test. Il contenuto della struct è il seguente:

- un elemento di tipo T che deve avere esattamente 1 occorrenza all'interno di un primo vector passato come argomento alla funzione di test
- un elemento di tipo T che deve avere almeno 1 occorrenza nello stesso vector
- un elemento di tipo T che deve avere 0 occorrenze sia nel primo vector sia in un secondo vector passato sempre come argomento alla funzione templata
- un numero deciso senza criterio (purché maggiore di zero) corrispondente al numero di add che si andranno ad eseguire per un certo test
- un metodo update che setta i 4 dati membro appena discussi.

Finalmente, alla riga 323, ho definito una **funzione globale templata** chiamata **exec_unit_test** che è il **"cuore" del main**, in quanto è la funzione che prende i seguenti parametri templati

- un funtore di ordinamento **Compare**
- un altro funtore di ordinamento **CompareOther**
- un funtore di uguaglianza **Equals**
- un tipo generico **Type** (quest'ultimo messo a destra perché dedotto)

e i seguenti argomenti

- il **numero del test** per un certo tipo di dato
- una stringa corrispondente al **titolo del test**
- un **vector di Type** non vuoto che rispetta certe condizioni elencate in seguito
- un'istanza della struct descritta in precedenza, che impacchetta **4 argomenti**
- un **altro vector di Type** non vuoto che rispetta anch'esso le seguenti condizioni

in modo che vengano rispettate le seguenti **pre-condizioni**

- test_number > 0

- values_to_add deve contenere esattamente un'occorrenza di four_args.element_with_1_occ, almeno un'occorrenza di four_args.element_with_1_or_more_occs e 0 occorrenze di four_args.element_with_0_occs
- four_args.number_of_add > 0
- values_to_add4 (chiamato 4 perché il 2 e il 3 sono generati dal primo utilizzando rispettivamente indici pari e indici dispari) deve contenere 0 occorrenze di four_args.element_with_0_occ.

Questa funzione esegue una serie di **test verificati in automatico** (cioè generando i dati aspettati ogni volta grazie ai miei metodi di comodo) per il tipo in questione; qui di seguito un elenco di funzionalità testate:

- **Metodi fondamentali** (ctor di default, distruttore testato implicitamente, cctor, op=)
- **Interfaccia pubblica**, che si divide in
    - Costruttore che prende una coppia di iteratori
    - Metodo clear
    - Metodo swap
    - Metodo size
    - Metodo add
    - Metodo remove (+ eccezione custom)
    - Metodo multiplicity
    - Operator==
    - Metodo contains
    - Stampa con iteratori
- **Const correctness**.

In seguito a questa funzione templata, nel main rimangono una serie di struct corrispondenti a funtori di ordinamento e di uguaglianza e metodi che richiamano la funzione templata per eseguire diversi test con diverse combinazioni di funtori per ogni tipo di dato.

Il **vantaggio** di questa funzione è che appunto è in grado di testare tutto riguardo alla classe ordered_multiset per un qualsiasi tipo di dato con qualsiasi funtori di ordinamento e di uguaglianza, lo **svantaggio** è che essendo templata la compilazione potrebbe impiegare qualche secondo in più, in quanto il compilatore deve aggiungere codice binario per ogni versione che specializza la funzione, e inoltre il suo codice non è di facile lettura.

Sono stati testati i seguenti **tipi di dato**:

- int
- char
- float
- std::string
- point (tipo custom)
- person (tipo custom)
- fraction (tipo custom)
- ordered_multiset di interi ("tipo custom").

Non ho eseguito test per i tipi primitivi bool e double, perché non li ho ritenuti significativi in quanto per rappresentare un bool è sufficiente limitarsi a utilizzare i numeri interi 0 e 1, mentre i double sono float a doppia precisione.