# *Indice*

# *Scelte condivise per entrambi i progetti*

## Utilizzo della lingua inglese

Ho scelto di utilizzare l'inglese per il codice e i commenti. Questa scelta è dovuta principalmente al fatto che l'inglese è ampiamente riconosciuto come la lingua standard nella programmazione. Questo rende il codice più accessibile e facilita la collaborazione.

# *Scelte implementative c++*

## Uso di size_t

Ho scelto di usare il tipo ***size_t*** per più motivi:

- è uno standard, quindi garantisce sicurezza e compatibilità su diverse piattaforme e architetture. Inoltre, essendo usato nella Standard Template Library (STL), rende il codice più coerente con le pratiche di STL;
- prevenzione di errori di underflow o overflow. Dato che ***size_t*** è un tipo senza segno, mi permette di essere sicuro che indici o dimensioni non diventino mai negativi, aumentando la sicurezza del codice.

## Utilizzo di un array dinamico per il salvataggio dei dati

Per salvare dati all'interno del mio Set, ho scelto di utilizzare un array dinamico che si raddoppia di dimensione quando è pieno e che si dimezza quando 3/4 della sua capacità non è utilizzata. Questo mi permette di gestire una quantità scalabile di dati, nonostante non possa utilizzare strutture a lista. Ridimensionando la dimensione dell'array su questi parametri riesco a trovare un punto di equilibrio nell'utilizzo della memoria.

## Gestione del ridimensionamento e delle eccezioni in resize

All'interno della mia classe Set, ho definito una funzione privata ***resize***. Questa funzione serve a ridimensionare l'array dinamico secondo i parametri specificati sopra. Ho deciso di tenere questa funzione privata in quanto non è una funzione che deve essere utilizzata dall'utilizzatore finale della classe Set, ma è solamente necessaria al funzionamento interno della classe.

Ho implementato la funzione ***resize*** in modo tale da gestire in modo sicuro il ridimensionamento, lasciando il Set sempre in uno stato consistente. Prima di tutto mi creo una variabile ***new_size*** e mi calcolo la nuova dimensione dell'array (raddoppio o dimezzo la variabile privata ***size***). Successivamente creo un puntatore di tipo **T** di nome ***new_array***, questo verrà poi inizializzato alla dimensione ***new_size*** all'interno di un blocco try catch, in modo tale da catturare e gestire le eccezioni di fallimento di allocazione.

In caso di successo nell'allocazione, verranno copiati i dati del Set in ***new_array***, sarà liberata la memoria occupata dal vecchio array e verranno trasferiti il puntatore e la nuova dimensione alle variabili locali del Set.

```C/C++
for (size_t i = 0; i < _num_elements; ++i) {
  new_array[i] = _array[i];
}

delete[] _array;
_array = new_array;
_size = new_size;
```

In caso di errore durante la fase di allocazione, inizio registrando il messaggio di errore che ha causato l'eccezione. Successivamente libero il nuovo array e re-throwo l'eccezione, in modo tale  da non avere memory leak e di avvisare l'utilizzatore finale dell'eccezione avvenuta. In questo caso, l'array rimarrà intatto, garantendo uno stato consistente.

```C/C++
std::cerr << "Exception caught in resize: " << e.what() << '\n';
delete[] new_array;
throw;
```

## Funzioni di add e remove

Le funzioni **add** e **remove** non utilizzano blocchi try-catch, permettendo la propagazione delle eccezioni al chiamante.

In ***add***, verifico prima l'esistenza dell'elemento e, se necessario, eseguo un ridimensionamento. L'elemento viene aggiunto solo dopo il successo dell'assegnazione, incrementando ***_num_elements*** per mantenere lo stato coerente. In caso di fallimento dell'assegnazione, ***_num_elements*** non viene aggiornato, mantenendo il Set in uno stato consistente.

In ***remove***, cerco l'elemento da rimuovere e lo sostituisco con l'ultimo elemento dell'array, decrementando ***_num_elements*** solo dopo aver completato con successo questa operazione. Questo garantisce che, in caso di fallimento dell'assegnazione, ***_num_elements*** rimanga accurato. Dopo, eseguo un ridimensionamento se il Set è significativamente sottoutilizzato.

Infine, per permettere un utilizzo più semplice del set, le funzioni **add** e **remove** restituiscono un valore booleano per indicare il successo o il fallimento dell'operazione.

## Gestione delle eccezioni nei costruttori e in filter_out, operator+ e operator-

All'interno dei **costruttori** della classe Set e delle funzioni globali ***filter_out***, ***operator+*** e ***operator-*** della mia classe Set, gestisco le eccezioni in questo modo:

1. Catturo eccezioni di tipo std::exception per ottenere e registrare messaggi di errore dettagliati tramite e.what(). Questo permette all'utilizzatore finale di comprendere l'errore.
2. Garantisco coerenza nello stato in caso di eccezione pulendo l'oggetto Set parzialmente costruito con il metodo empty(). Questo assicura che non rimangano risorse non deallocate o elementi parzialmente aggiunti.
3. Dopo aver registrato l'errore e pulito lo stato, rilancio l'eccezione. Questo lascia all'utilizzatore della classe Set la gestione dell'errore.

```C/C++
std::cerr << "Exception caught in ...: " << e.what() << '\n';
new_set.empty(); //solo empty() all'interno dei costruttori
throw;
```

## Altre scelte minori

### Utilizzo di std::cerr in caso non si riesca ad aprire il file

Ho deciso di utilizzare ***std::cerr*** in caso non si riesca ad aprire il file per il salvataggio all'interno della funzione ***save***. Ho deciso di gestire questo caso in questo modo perchè molto spesso è un problema che deriva da mancati permessi per aprire i file da parte dell'utilizzatore, e non mi sembrava opportuno complicare la logica attraverso l'utilizzo di ulteriori throw.

```C/C++
if (!outFile.is_open()) {
 std::cerr << "Failed to open file: " << filename << std::endl;
 return;
}
```

### Utilizzo di throw per errori di indice

Per definire l'operatore di indicizzazione del Set '[ ]' ho considerato l'uso di ***size_t*** al posto di un intero per evitare controlli per valori minori di zero. Tuttavia, ho preferito optare per l'utilizzo dell'ultimo, in quanto ***size_t*** quando negativo viene automaticamente convertito in

un numero positivo (anche se molto grande e quasi certamente al di fuori del range dell'array).
Se l'indice dell'array specificato è al di fuori del range, faccio il throw dell'eccezione ***std::out_of_range***.

```C/C++
if (index < 0 || index >= _num_elements) {
      throw std::out_of_range("Index out of range");
}
```

# *Scelte implementative e di design Qt*

## Gestione anno inserito in "Data"

Nella gestione dell'anno inserito nel campo "Data", ho sviluppato una funzione che identifica il primo gruppo di 3 o 4 cifre consecutive che rappresentano un anno compreso tra 100 e 2024. Se nella stringa sono presenti due anni che rientrano nel range definito, la funzione li estrae e ne calcola la media come valore finale. Nel caso in cui il secondo anno identificato preceda il primo anno nell'ordine di inserimento, lo considero un errore di inserimento e quindi ignoro il secondo anno. Questo approccio garantisce che vengano considerati solo gli anni inseriti correttamente ai fini del calcolo.

## Parsing del csv

Ho scritto la funzione ***parseCsvLine*** per analizzare una riga di un file CSV e suddividerla in campi, restituendo un ***QStringList***. All'inizio, stabilisco un enum ***State*** con due stati, ***Normal*** e ***Quote***, per gestire correttamente i campi racchiusi tra virgolette. La funzione scorre ogni carattere della stringa input: se si trova in uno stato ***Normal*** e incontra una virgola, considera che il campo corrente è terminato, lo aggiunge alla lista dopo averlo trimmato e azzera la variabile ***value*** per iniziare la raccolta del prossimo campo. Se incontra una virgoletta, cambia lo stato in ***Quote***, indicando che è entrata in una sezione racchiusa tra virgolette. Se è già in stato ***Quote*** e incontra una virgoletta, controlla se è seguita da un'altra virgoletta (indicando una virgoletta letterale nel campo) o se la virgoletta chiude il campo. Alla fine del ciclo, aggiunge l'ultimo campo alla lista dopo averlo trimmato. Questo metodo garantisce che tutti i campi vengano correttamente estratti anche se contengono virgolette o virgole.

## Ricerca interattiva nella tabella

Ho implementato un approccio interattivo per la ricerca all'interno della mia applicazione. Invece di richiedere all'utente di premere un pulsante per avviare la ricerca, ho reso il processo automatico: la tabella si aggiorna in tempo reale non appena il campo di ricerca viene modificato. Questo significa che con ogni tasto digitato o cancellato, i risultati vengono immediatamente filtrati.

Per ottimizzare le prestazioni, ho scelto di nascondere le righe che non corrispondono ai criteri di ricerca piuttosto che rimuovere e ricreare i contenuti della tabella ad ogni cambiamento. Questo approccio è più efficiente, nascondere le righe inoltre permette di mantenere l’indicizzazione all’interno della tabella.

## Bottone per cambiare grafico di visualizzazione

Ho optato per l'uso di un pulsante e uno stacked Widget per gestire la visualizzazione dei grafici nella mia applicazione. Questo mi permette di alternare tra i due grafici con un semplice click, presentando ciascun grafico in una visualizzazione più grande e migliorando così la leggibilità dei dati.

Per assicurare che l'interfaccia rimanga organizzata e facilmente navigabile, ho deciso di impostare una dimensione minima per la finestra dell'applicazione. Questo impedisce il "cramming" o l'affollamento di elementi grafici quando la finestra viene ridimensionata, garantendo che l'interfaccia rimanga chiara e utilizzabile anche con cambiamenti nelle dimensioni della finestra. Questa decisione bilancia l'esigenza di flessibilità nell'uso dello spazio con quella di mantenere una presentazione chiara e funzionale dei dati.