Tecniche Programmazione Avanzata
Esercizi OOP

Luca Olivieri, Andre Del Prete
luca.olivieri-1@unitn.it, andrea.delprete@unitn.it

Università di Trento — May 12, 2020

Attenzione: Per risolvere nella loro completezza i seguenti esercizi si richiede di aver affrontato la programmazione a template e l'overloading degli operatori. È comunque possibile implemetare una versione semplificata: specificare il tipo di dato esplicitamente e implementare gli operatori come semplici metodi (*Java-style: in Java non esiste l'overload degli operatori, piuttosto di definiscono metodi. Quindi invece che .operator+() si avrebbe .addWithStuff()*).

Introduzione

La cornice narrativa dei seguenti esercizi coinvolge due team di sviluppo che lavorano su progetti collegati. Dopo diversi meeting tra i project manager è scaturita la necessità di sviluppare le seguenti librerie. L'implementazione di una libreria è assegnata ad un singolo team, o parte di esso. All'altro team, e in generale a chi non è coinvolto nello sviluppo, non interessano i dettagli implementativi, per questo sono state definite delle classi di interfaccia.
I seguenti esercizi non sono valutati. Vogliono essere un'occasione di fare pratica e mettersi alla prova. È incoraggiato collaborare e confrontarsi con i compagni sui possibili svolgimenti. A questo proposito è consigliato crearsi una repository su GitHub e aggiungere possibili collaboratori. Saper discutere e confrontarsi su possibili implementazioni è una capacità importante per un programmatore. Per ogni esercizio implementare test con Catch2. Non c'è un particolare ordine in cui affrontarli.

1 Collection Library

Per collection si intende una generica struttura che possa contenere una serie di oggetti. La struttura base the contiene i dati è sempre un array, ma incartato (*"wrapped"*) da un oggetto che espande le sue funzionalità. Tra le molte collection esistenti distinguiamo, per cominciare:

- **Liste**: inserire e togliere elementi in qualsiasi posizione
- **Coda**: inserire solo all'inizio [`void queue(T element)`] e togliere solo alla fine [`T unqueue()`] (*FIFO - First In First Out*)
- **Stack**: inserire solo all'inizio [`void push(T element)`] e togliere solo all'inizio [`T pop()`] (*LIFO - Last In First Out*)
- **Set**: mentre le altre sono strutture ordinate, il set è solo un insieme di elementi. La particolarità è che uno stesso elemento non può essere inserito più di una volta. *Es.In un set di int = {34, 6, 91, 4} inserire un 6 non cambierà il set.* È possibile fare set solo di oggetti che possono essere confrontati in uguaglianza (*operator==*). Anche se formalmente scorretto, si potrà accedere e iterare gli elementi del Set usando l'operatore [].

Info: Questo esercizio di fatto chiede di implementare una versione semplificata delle strutture presenti nella *Standard Library*. Le sezione **EXTRA** mimica il comportamento di `std::vector`.

Consegna 1

Creare una libreria che implementa le strutture sopra elencate.

- Ogni collection deve essere una classe separata, facente parte di una unica gerarchia.
- Le collection implementano l'interfaccia `Collectable`
- Implementare i metodi che si ritengono necessari perché le strutture siano intuitive da usare (Es. pop, unqueue, ...)
- Le collection devono essere generiche: tramite template possono contenere qualsiasi tipo di oggetto
- **EXTRA:** fare in modo che le collection possano contenere un numero arbitrario di elementi. Vale a dire quando l'utente chiede di aggiungere un elemento in più della capacità massima, la collection alloca un nuovo blocco di memoria che può accogliere un numero n di elementi in più.

Collectable.h

```
template <typename T>
class Collectable {
public:
        // Quanti elementi sono presenti
        virtual int count() = 0;
        // Svuota la struttura
        virtual void empty() = 0;
        // Aggiunge un elemento alla struttura
        virtual void addElement(T& element) = 0;
        // Rimuove e restituisce un elemento
        virtual T removeElement() = 0;
}
```

Info: Ragionare su quali tipi di strutture possono essere ricondotte ad altre. Cominciare con la modellazione della Lista. Su carta. Pensare a quali metodi potrebbero essere richiamati da altre strutture.

Una parte particolarmente complessa può essere la rimozione di un elemento, che costringe a spostare tutti gli altri elementi, oppure di tenere traccia del buco che si è creato nell'array contenitore.

Notare come l'interfaccia non constringa a farsi restituire un elemento in particolare. Cosa però possibile nelle Liste e nei Set. Scegliere una logica da seguire per queste due strutture. In generale è anche possibile farsi restituire un oggetto senza rimuoverlo.

2 Signal Processing Library

Una basica libreria di utility per elaborazione di segnali. Le funzionalità sono le seguenti:

- **Segnale**: Modellazione di un segnale a una dimensione
- **Polinomi**: Valutazione, Campionamento, Derivazione, Integrazione
- **Filtri Digitali**: FIR, IIR
- **Stima Velocità**: Forward Difference, Central Difference, Backward Difference

Polinomi

I polinomi sono contraddistinti da un ordine k e da coefficienti a_i.

- **Valutazione**: Calcola il valore di un polinomio in un punto specificato. *Parametri di ingresso*: t punto nel quale valutare il polinomio. *Uscita*: valore del polinomio $x(t) = \sum_{i=0}^{k-1} a_i t^i$
- **Campionamento**: Campionare un polinomio ad intervalli di tempo specificati. *Parametri di ingresso*: dt tempo tra due campionamenti successivi; $t0$ tempo iniziale del campionamento; N numero di campioni. *Uscita*: un segnale.
- **Derivazione**: Calcola la derivata di un polinomio. *Uscita*: un nuovo polinomio avente coefficienti della derivata.
- **Integrazione**: Calcola l'integrale del polinomio. *Uscita*: un nuovo polinomio avente coefficienti della derivata.

Possibili test: l'integrale della derivata di un polinomio è uguale al polinomio originale; la derivata dell'integrale di un polinomio è uguale al polinomio originale; la derivata di un polinomio del primo ordine è una costante; l'integrale di una costante è un polinomio del primo ordine.

Filtri Digitali

Algoritmi per l'elaborazione di segnali campionati, quindi in tempo discreto. Le specifiche sono nel dominio delle frequenze, ad esempio: passa basso, passa alto, passa banda. Tra i molti esistenti distinguiamo:

- **FIR - Finite Input Response**: Descritti da una combinazione lineare degli ultimi k ingressi: $y(n) = \sum_{i=0}^{k-1} a_i x[n-i]$. Il filtro è caratterizzato dai coefficienti a_i che possono essere facilmente ricavati ad esempio da: http://t-filter.engineerjs.com/. k è l'ordine del filtro.
- **IIR - Infinitet Input Response**: Descritti da $y(n) = \sum_{i=0}^{k-1} a_i x[n-i] - \sum_{i=1}^{l-1} b_i y[n-i]$. Caratterizzato dai coefficienti a_i e b_i. L'ordine del filtro è dato da $max(k, l)$.

Dove x è il segnale originale e y il segnale filtrato. I due filtri implementano entrambi una funzione `filter` che prende in ingresso un oggetto processabile e restituisce un segnale.

Stima Velocità

Il problema della stima della velocità (ovvero la derivata rispetto al tempo) di un segnale è un problema classico dell'ingegneria che appare in svariati contesti. Esistono svariate tecniche per la stima della velocità, ma l'approccio più comune consiste in due passi: filtraggio del segnale con filtro passa basso e successivo calcolo della velocità del segnale filtrato con tecnica delle "differenze finite".

- **Forward Difference**: $\dot{x}[n] \approx \frac{x[n]-x[n-1]}{\Delta}$
- **Central Difference**: $\dot{x}[n] \approx \frac{x[n+1]-x[n]}{\Delta}$
- **Backward Difference**: $\dot{x}[n] \approx \frac{x[n+1]-x[n-1]}{\Delta}$

La classe implementa un metodo statico per ogni tecnica che prende in ingresso un oggetto processabile e restituisce un segnale.

Consegna 2

Creare una libreria che modelli `Polynomial`, `Signal`, `FIRFilter`, `IIRFilter` e `VelocityEstimation`.

- Sia `Polynomial` che `Signal` implementano l'interfaccia `Processable`, rendendoli compatibili con i metodi di elaborazione dei filtri digitali e della stima di velocità.
- `Polynomial`, `FIRFilter`, `IIRFilter` implementano `Serializable`. Questo permette di salvare e caricare gli oggetti da una stringa. Nella pratica significa che è possibile inizializzare un polinomio o un filtro partendo da un file. Idem per salvataggio.
- `VelocityEstimation` è una classe statica: non è possibile crearne un'instanza. Quindi ha uno stato globale ed è una semplice raccolta di funzioni. Eventuali filtraggi vanno fatti esternamente.

Info: Ci si potrebbe porre il dubbio che l'oggetto `Signal` possa essere ricondotto a una delle collection dell'esercizio precedente. È preferibile modellare un segnale come un semplice oggetto a sé stante per una questione di rapidità nell'accedere agli elementi e perché le eventuali funzioni di corredo di aggiungere e togliere elementi non sono applicabili al contesto.

Implementare l'interfaccia `Processable` sulla classe `Polynomial` significa fissare valori di default per il campionamento. Questi possono essere modificati dall'utente.

Serializable.h

```
class Serializable {
public:
        virtual string serialize() = 0;
        virtual void deserialize(string data) = 0;
}
```

Processable.h

```
class Processable {
public:
        virtual Signal getSignal() = 0;
}
```