Luca Parimbelli
Matricola: 1053142
https://github.com/lucaparimbelli/libreriaDigitale-cpp

Documentazione progetto C++
Libreria digitale

Introduzione
Il progetto si pone l'obiettivo di utilizzare i principali costrutti di C++ presentati durante il corso di "Informatica III – modulo programmazione" per implementare una libreria digitale. La libreria digitale consente di svolgere le principali attività tipiche di una libreria: visualizzazione degli articoli presenti nel sistema, ricerca di un articolo, inserimento e cancellazione di nuovi articoli, prestito e restituzione di articoli disponibili.

Struttura
Il progetto è diviso in 4 packages, ognuno dei quali contiene diverse classi che consentono di gestire aspetti differenti del programma. In particolare:

- *libreria*, contiene le classi che servono per modellizzare la struttura della libreria
- *manager*, contiene le classi che consentono di implementare la logica del programma
- *menuCLI*, contiene la classe che consente di gestire l'interfaccia grafica a linea di comando
- *menuQT*, contiene le classi che consentono di gestire l'interfaccia grafica realizzata con QT Creator

Le classi contenute nei packages *libreria* e *manager* vengono utilizzate sia dalle classi di *menuCLI* che da quelle di *menuQT* per realizzare le due interfacce grafiche del progetto, la prima a linea di comando e la seconda realizzata con QT Creator 5.0.0 (*Fig. 1*).

Fig. 1: interfaccia grafica CLI (a sinistra) e QT (a destra)

Classi e relazioni
Le classi e le loro relazioni all'interno di ciascun package sono descritte dai successivi *class diagrams*. Nei diagrammi l'utilizzo di determinati costrutti appare a volte forzato, come ad esempio l'utilizzo dell'ereditarietà privata per *DateTimePub* anziché l'impiego di un campo privato di tipo *DateTimePub*. Questo non rappresenta un errore in quanto l'obiettivo del progetto non è quello di realizzare un programma dall'architettura ben definita e strutturata, bensì è quello di cercare di utilizzare e presentare la maggior quantità di costrutti possibili tipici di C++.

Il package *libreria* è descritto dal seguente class diagram (*Fig. 2*):

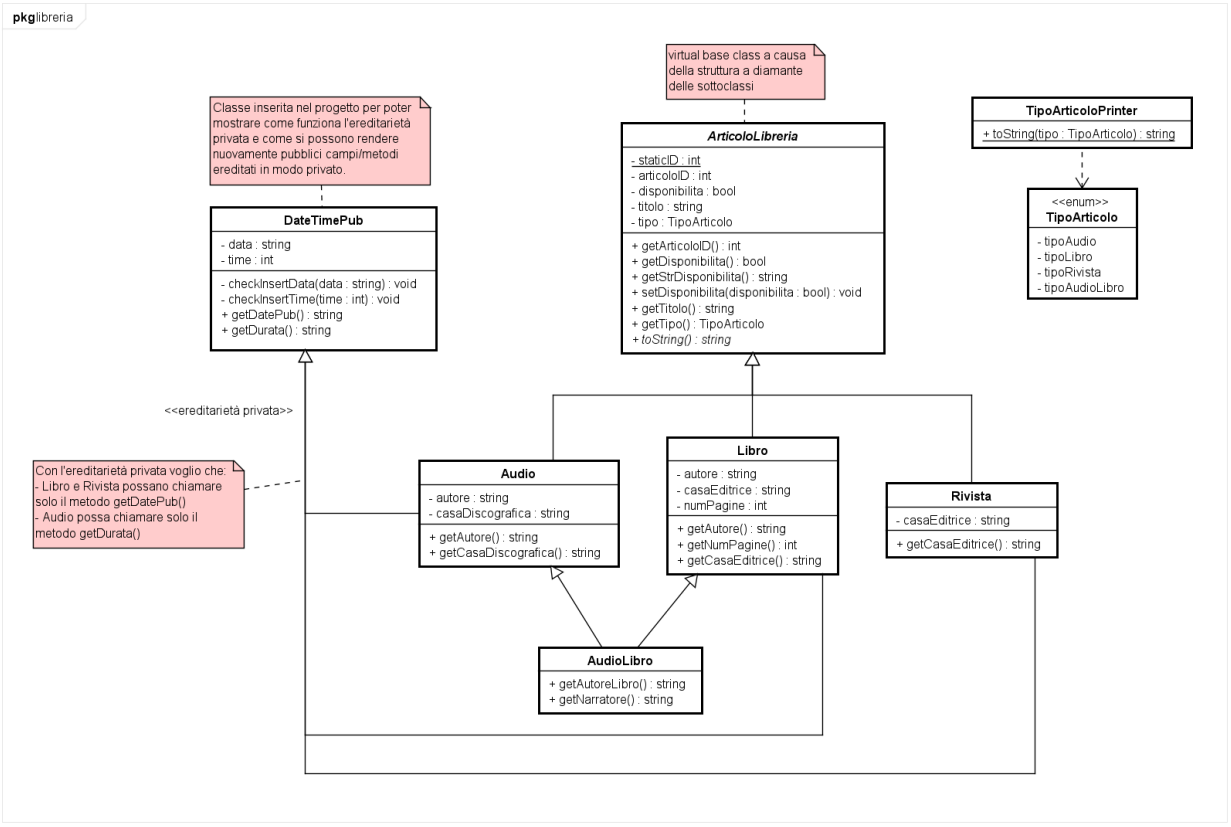


Fig. 2: libreria class diagram

Il package *manager* è descritto dal seguente class diagram (*Fig. 3*):

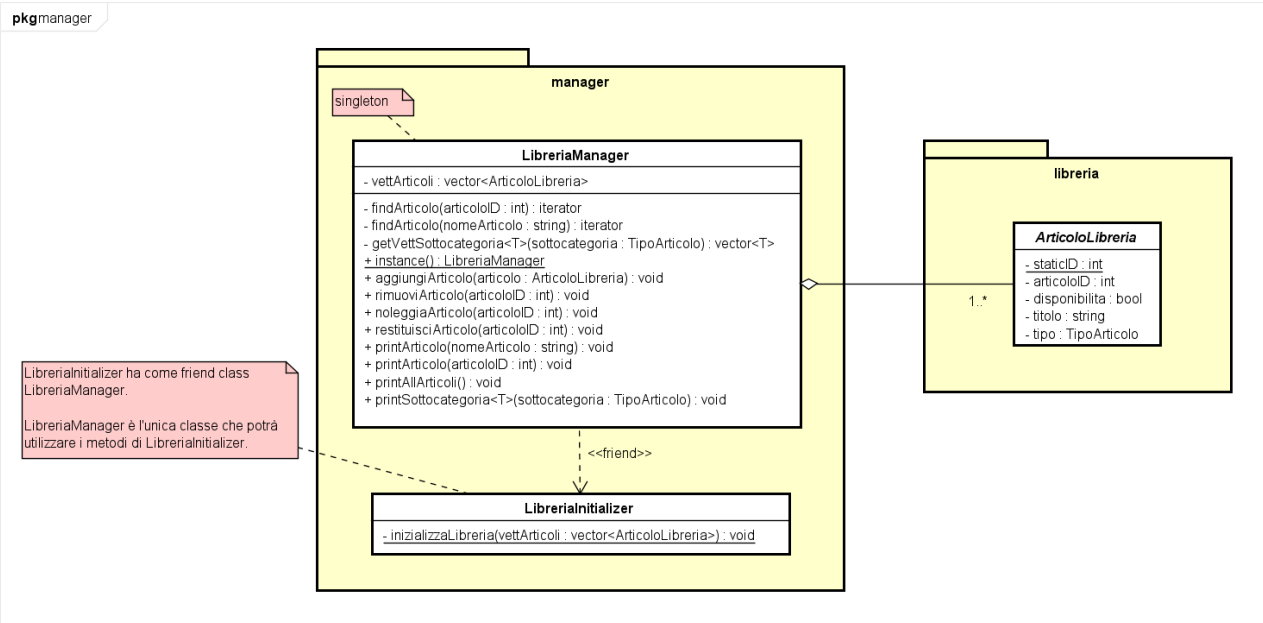


Fig. 3: manager class diagram

Il package *menuCLI* è descritto dal seguente class diagram (*Fig. 4*):

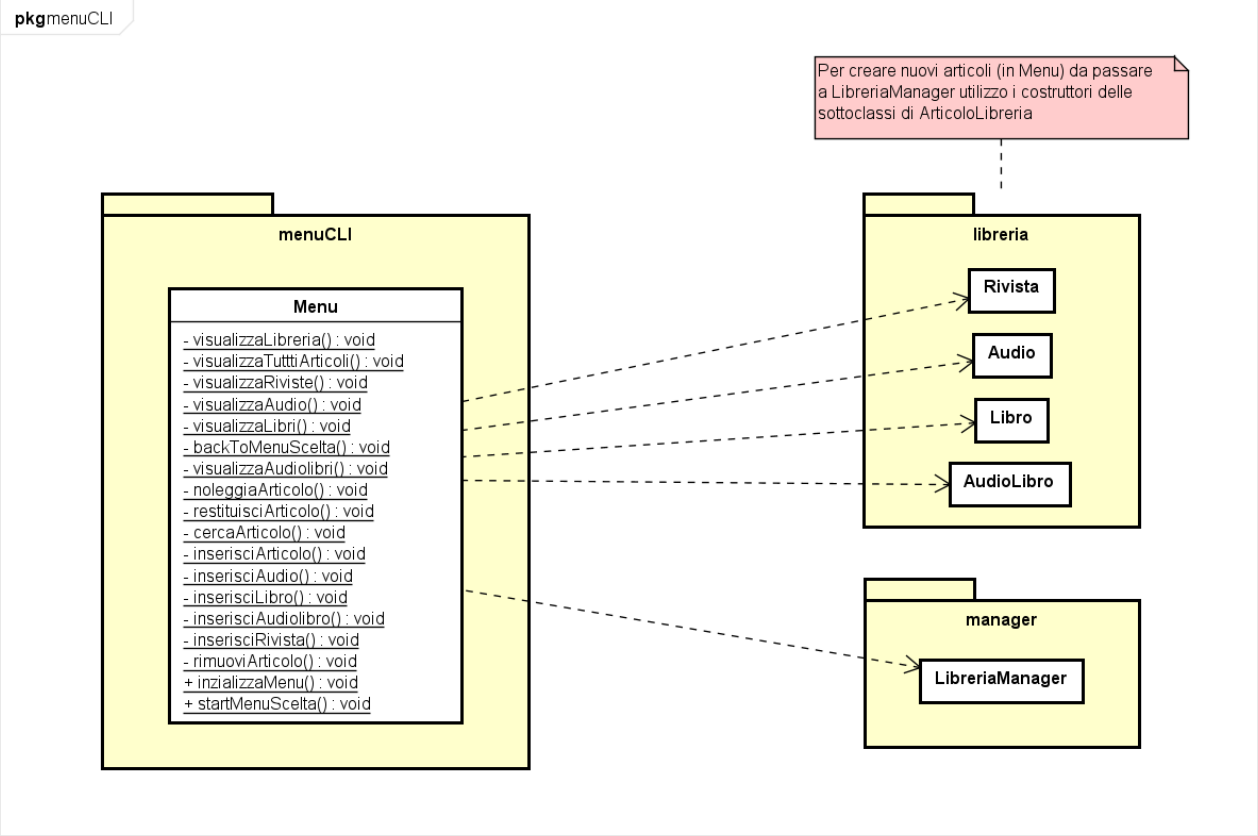


Fig. 4: menuCLI class diagram

Costrutti di C++

Nei successivi sottoparagrafi vengono presentati tutti i costrutti tipici di C++ che sono stati utilizzati per la realizzazione del progetto.

Classi

Come si può vedere in *Fig. 5*, i file relativi alle classi rispecchiano la struttura presentata nei precedenti class diagrams e sono stati correttamente divisi in .h (interfaccia) e .cpp (implementazione).

Fig. 5: classi .h e .cpp

Classe astratta

La classe *ArticoloLibreria* è stata ideata come classe astratta. Come possiamo osservare in *Fig.6*, il metodo *pure virtual* che rende effettivamente la classe astratta è *toString()*, il quale viene implementato in tutte le sottoclassi (override) e prevede di stampare informazioni sulla sottoclasse seguendo un formato di rappresentazione specifico.

```
36    //metodo toString pure virtual da implementare nelle sottoclassi
37    virtual string toString()=0;
```

Fig. 6: metodo pure virtual che rende la classe ArticoloLibreria astratta

Ereditarietà: multipla e privata

Nel progetto, l'ereditarietà multipla si concretizza con le classi Audio, Libro, Rivista e AudioLibro. Ognuna di queste classi eredita da altre due classi. In particolare:

- Audio, Libro, Rivista ereditano in modo pubblico da ArticoloLibreria e in modo privato da DateTimePub. L'ereditarietà privata è stata utilizzata in quanto si è voluto fare in modo che le classi Audio, Libro e Rivista potessero utilizzare solo uno dei due metodi definiti in DateTimePub. In particolare, Libro e Rivista possono invocare solo il metodo *getDatePub()*, mentre Audio può invocare solo il metodo *getDurata()*. Questi metodi sono stati resi nuovamente pubblici dopo essere stati ereditati in modo privato mediante l'utilizzo del costrutto in *Fig. 7*.
- AudioLibro eredita in modo pubblico da Audio e da Libro. Questo porta alle problematiche descritte nel prossimo sottoparagrafo (struttura a diamante e name clashes).

```
14 class Audio: virtual public ArticoloLibreria, private DateTimePub {

16 class Libro: virtual public ArticoloLibreria, private DateTimePub {

15 class Rivista: public ArticoloLibreria, private DateTimePub {

14 class AudioLibro: public Audio, public Libro {
```

Classi Libro e Rivista:

```
34    //rendo nuovamente pubblic getDatePub() di DateTimePub
35    using DateTimePub::getDatePub;
```

Classe Audio:

```
31    //rendo nuovamente pubblic getDurata() di DateTimePub
32    using DateTimePub::getDurata;
```

Fig. 7: ereditarietà multipla e privata

Struttura a diamante e name clashes

La classe AudioLibro eredita in modo pubblico sia da Audio che da Libro, le quali a loro volta ereditano in modo pubblico da ArticoloLibreria. Questo porta alla cosiddetta *struttura a diamante*, con AudioLibro che si ritrova ad avere i campi e i metodi di ArticoloLibreria duplicati. Per eliminare le copie duplicate dei membri della classe base ArticoloLibreria, viene impiegata la parola chiave *virtual* nell'ereditarietà pubblica di Audio e Libro dalla classe base ArticoloLibreria (*Fig. 8*).

Audiolibro eredita inoltre il campo *autore* da entrambe le classi Audio e Libro. Questa situazione è nota come *name clashes*, ossia nomi uguali ereditati da più classi. Per risolvere questa situazione, si utilizza la risoluzione esplicita specificando il nome della superclasse a cui il campo che stiamo utilizzando appartiene (*Fig. 8*). In particolare, nella classe AudioLibro:

- il metodo *getAutoreLibro()* restituisce l'autore del libro, ossia il campo autore della classe Libro
- il metodo *getNarratore()* restituisce l'autore della traccia audio dell'audiolibro, ossia il campo autore della classe Audio

```
15 //eredito virtual per evitare "member duplication" in "AudioLibro"
16 class Libro: virtual public ArticoloLibreria, private DateTimePub {

14 class Audio: virtual public ArticoloLibreria, private DateTimePub {
```

```
26 //risolvo il name clash di "autore" con l'explicit resolution
27 string AudioLibro::getAutoreLibro() {
28     return Libro::getAutore();
29 }
30 string AudioLibro::getNarratore() {
31     return Audio::getAutore();
32 }
```

Fig. 8: virtual base class (sopra) e explicit resolution (sotto)

Initializer list

L'initializer list viene utilizzata nel costruttore di ArticoloLibreria per inizializzare i suoi campi, come mostrato in *Fig. 9*.

```
15 //initializer list
16 //alla creazione di un articolo la disponibilità viene settata a true
17 ArticoloLibreria::ArticoloLibreria(string titolo, TipoArticolo tipo) :
18         articoloID(++staticID), disponibilita(true), titolo(titolo), tipo(tipo) {
19 }
```

Fig. 9: initializer list

Utilizzo parola chiave this

Il codice dei metodi di una classe viene compilato aggiungendo come primo argomento della lista dei parametri un puntatore *this* all'oggetto su cui il metodo viene invocato. Per mostrarne l'utilizzo, anche se non strettamente necessario, in alcuni metodi è stato utilizzato il puntatore *this* (*Fig. 10*).

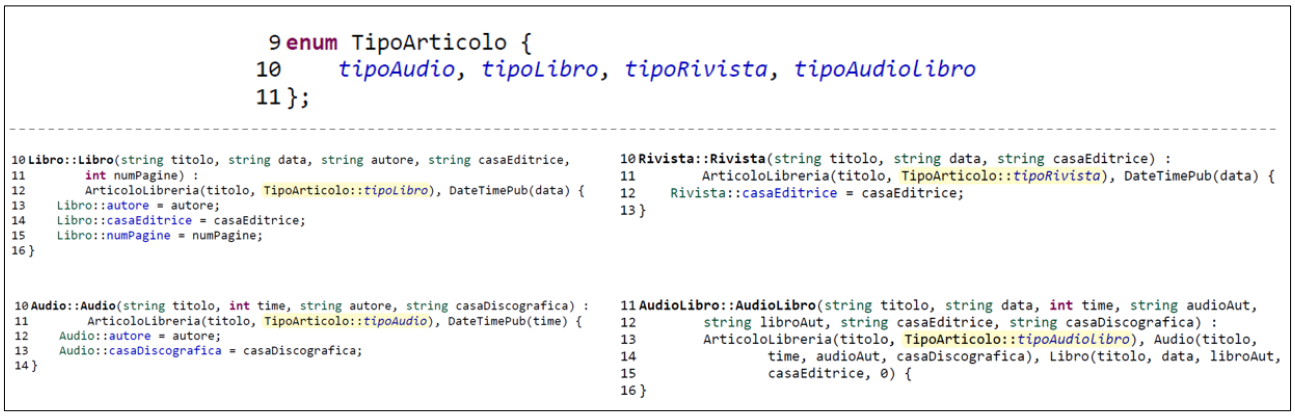
```
28 bool ArticoloLibreria::getDisponibilita() {
29     return this->disponibilita;
30 }
```

```
32 string ArticoloLibreria::getStrDisponibilita() {
33     return (this->disponibilita == true) ? "true" : "false";
34 }
```

Fig. 10: utilizzo parola chiave this

Enum

Il formato specifico ideato per la stringa che contiene le informazioni circa un articolo e che viene restituita dal metodo *toString()* prevede di specificare la tipologia dell'articolo nella prima linea della stringa. Per questo motivo è stato progettato un enum *TipoArticolo* che può assumere solo 4 valori: tipoAudio, tipoLibro, tipoRivista e tipoAudioLibro. Nel costruttore delle sottoclassi, il campo *tipo* di ArticoloLibreria viene inizializzato ad uno dei 4 possibili valori dell'enum a seconda di quale sottoclasse richiama il supercostruttore di ArticoloLibreria. L'implementazione dell'enum e l'inizializzazione del campo *tipo* di ArticoloLibreria da parte delle sottoclassi sono illustrati entrambi in *Fig. 11*.



```
enum TipoArticolo {
    tipoAudio, tipoLibro, tipoRivista, tipoAudioLibro
};
```

```
Libro::Libro(string titolo, string data, string autore, string casaEditrice,
        int numPagine) :
        ArticoloLibreria(titolo, TipoArticolo::tipoLibro), DateTimePub(data) {
    Libro::autore = autore;
    Libro::casaEditrice = casaEditrice;
    Libro::numPagine = numPagine;
}
```

```
Rivista::Rivista(string titolo, string data, string casaEditrice) :
        ArticoloLibreria(titolo, TipoArticolo::tipoRivista), DateTimePub(data) {
    Rivista::casaEditrice = casaEditrice;
}
```

```
Audio::Audio(string titolo, int time, string autore, string casaDiscografica) :
        ArticoloLibreria(titolo, TipoArticolo::tipoAudio), DateTimePub(time) {
    Audio::autore = autore;
    Audio::casaDiscografica = casaDiscografica;
}
```

```
AudioLibro::AudioLibro(string titolo, string data, int time, string audioAut,
        string libroAut, string casaEditrice, string casaDiscografica) :
        ArticoloLibreria(titolo, TipoArticolo::tipoAudioLibro), Audio(titolo,
                time, audioAut, casaDiscografica), Libro(titolo, data, libroAut,
                casaEditrice, 0) {
}
```

Fig. 11: enum TipoArticolo e inizializzazione campo tipo di ArticoloLibreria

Campi statici

La classe ArticoloLibreria utilizza un campo statico denominato *staticID* (*Fig. 12*) per generare automaticamente l'*articoloID* degli articoli che vengono aggiunti alla libreria digitale. Il campo *staticID* è dichiarato nel file .h e inizializzato nel file .cpp. L'*articoloID* è generato a partire dallo *staticID* mediante la tecnica mostrata in *Fig. 9*.

articololibreria.h :

```
    //staticID = campo statico per incrementare automaticamente articoloID
    static int staticID;
```

articololibreria.cpp :

```
//membro statico inzializzato nel .cpp
int ArticoloLibreria::staticID = 0;
```

Fig. 12: membro statico staticID dichiarato nel .h e inizializzato nel .cpp

Metodi statici

Ci sono diverse classi che implementano metodi statici, ossia metodi che appartengono alla classe piuttosto che ad un'istanza della classe e che possono quindi essere invocati senza bisogno di istanziare un oggetto della classe. Le classi che implementano metodi statici sono:

- TipoArticoloPrinter con il metodo *toString(TipoArticolo tipo)*, il quale viene utilizzato per convertire un valore dell'enum TipoArticolo in una delle stringhe AUDIO, LIBRO, RIVISTA, AUDIOLIBRO.
- LibreriaInitializer con il metodo *inizializzaLibreria(vector<ArticoloLibreria*> &vettArticoli)*, il quale inizializza la libreria digitale con degli articoli di default nella fase di avvio del programma
- LibreriaManager con il metodo *instance()* che restituisce (e istanzia alla prima chiamata) l'unica istanza di LibreriaManager disponibile
- Menu che è dotata solo di metodi statici che vengono utilizzati per inizializzare e gestire l'interfaccia a linea di comando della libreria digitale

I metodi statici delle classi in elenco sono illustrati in *Fig. 13*.

TipoArticoloPrinter :

```
16    static string toString(TipoArticolo tipo);
```

LibreriaManager :

```
57    static LibreriaManager& instance();
```

LibreriaInitializer :

```
16    static void inizializzaLibreria(
        vector<ArticoloLibreria*> &vettArticoli);
```

Menu :

```
10 class Menu {
11 private:
12     static void visualizzaLibreria();
13     static void visualizzaTuttiArticoli();
14     static void visualizzaRiviste();
15     static void visualizzaAudio();
16     static void visualizzaLibri();
17     static void visualizzaAudiolibri();
18     static void noleggiaArticolo();
19     static void restituisciArticolo();
20     static void cercaArticolo();
21     static void inserisciArticolo();
22     static void inserisciLibro();
23     static void inserisciAudio();
24     static void inserisciRivista();
25     static void inserisciAudiolibro();
26     static void rimuoviArticolo();
27     static void backToMenuScelta();
28
29 public:
30     static void inizializzaMenu();
31     static void startMenuScelta();
32 };
```

Fig. 13: metodi statici

Visibilità static per un metodo nel file .cpp

Quando si utilizza la parola chiave static al di fuori di una classe, questa stabilisce la “visibilità" della/del variabile/metodo alla quale fa riferimento. In particolare, le/i variabili/metodi static non sono visibili all’esterno del modulo al quale appartengono. Variabili/metodi che non devono essere utilizzati anche all’esterno dell’unità di compilazione alla quale appartengono devono essere quindi dichiarati static. Nel progetto, il metodo ricorsivo con coda *countDigitTail(int num, int sum)* definito nel file datetimepub.cpp è preceduto dalla parola chiave static in quanto è un metodo di supporto che viene utilizzato solo per facilitare l’implementazione di altri metodi della classe DateTimePub (*Fig. 14*).

datetimepub.cpp :

```
65 //funzione tail recursive: restituisce il numero di cifre di cui è composto num
66 //usata solo in questo .cpp, uso "static" per limitare la sua visibilità a questo modulo
67 static int countDigitTail(int num, int sum) {
68     if (num / 10 == 0)
69         return sum + 1;
70     return countDigitTail(num / 10, sum + 1);
71 }
```

Fig. 14: metodo countDigitTail in datetimepub.cpp

Friend classes

Una friend class può accedere ai membri private e protected di un’altra classe. Nel progetto, la classe LibreriaManager è dichiarata friend class di LibreriaInitializer (*Fig. 15*). Questo consente alla classe LibreriaManager di invocare l’unico metodo (statico) di LibreriaInitializer (*inizializzaLibreria(…)*), il quale inizializza gli elementi della libreria digitale in fase di caricamento del programma. Con questo meccanismo riusciamo a costruire una classe (LibreriaInitializer) con un metodo statico che non potrà mai essere invocato se non all’interno della classe LibreriaManager.

Classe LibreriaInitializer :

```
17    friend class LibreriaManager;
```

Fig. 15: LibreriaManager friend class di LibreriaInitializer

Overload

L'overload di un metodo si ha quando il metodo preso in considerazione ha lo stesso nome di altri metodi, ma presenta una diversa definizione dei parametri (tipo e/o numerosità). Il compilatore sceglie, in fase di compilazione, quale metodo dovrà essere invocato nella successiva fase di esecuzione sulla base del tipo degli argomenti che vengono passati come parametri al metodo richiamato. Nel progetto, l'overload viene principalmente sfruttato per poter definire più costruttori nelle varie classi, ognuno dei quali prende più o meno parametri. In questo modo è possibile definire dei costruttori che consentono di istanziare oggetti che non hanno tutti i campi inizializzati ad uno specifico valore definito dall'utente. Un esempio di overload dei costruttori è riportato in *Fig. 16*.

rivista.h

```
20    Rivista(string titolo, string data, string casaEditrice);
21    Rivista(string titolo, string data);
```

rivista.cpp

```
10 Rivista::Rivista(string titolo, string data, string casaEditrice) :
11         ArticoloLibreria(titolo, TipoArticolo::tipoRivista), DateTimePub(data) {
12     Rivista::casaEditrice = casaEditrice;
13 }
14
15 Rivista::Rivista(string titolo, string data) : Rivista(titolo,data,"null"){
16 }
```

Fig. 16: overload del costruttore Rivista

Override e binding dinamico

Per avere override e binding dinamico in C++ è necessario soddisfare due requisiti:

1. avere un metodo in una sottoclasse con la stessa segnatura di un metodo virtual della superclasse
2. la chiamata al metodo deve avvenire tramite riferimento o puntatore

Nel progetto, il metodo principale di cui viene fatto l'override è il metodo *toString()* della classe ArticoloLibreria (*Fig. 17*). Tutte le sottoclassi presentano una diversa implementazione di tale metodo in quanto ogni sottoclasse contiene informazioni di tipo diverso. La chiamata a tale metodo avviene sempre tramite puntatore poiché il metodo *toString()* è sempre invocato a partire da variabili di tipo *ArticoloLibreria** memorizzate all'interno di un vector (*vector<ArticoloLibreria*> vettArticoli*). Abbiamo quindi binding dinamico per il metodo *toString()*: a seconda dell'istanza puntata da una variabile di tipo ArticoloLibreria, verrà chiamato il metodo *toString()* di una sottoclasse piuttosto che un'altra.

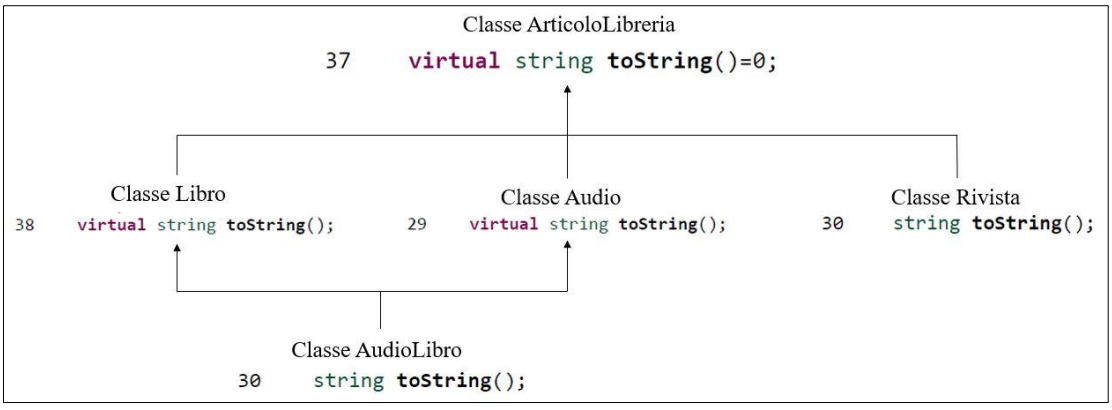


Fig. 17: override del metodo toString()

Distruttori virtual

Con lo scopo di evitare memory leak ed essere sicuri che tutte le istanze di classi derivate puntate da puntatori di tipo la classe base vengano eliminate, alcuni distruttori sono stati correttamente dichiarati virtual. In questo modo si ha la certezza che le istanze di Libro, Audio, Rivista e AudioLibro puntate da puntatori di tipo ArticoloLibreria (*vettArticoli*) vengano correttamente eliminate quando viene utilizzata il delete. L'utilizzo dei distruttori virtual è mostrato in *Fig. 18*.

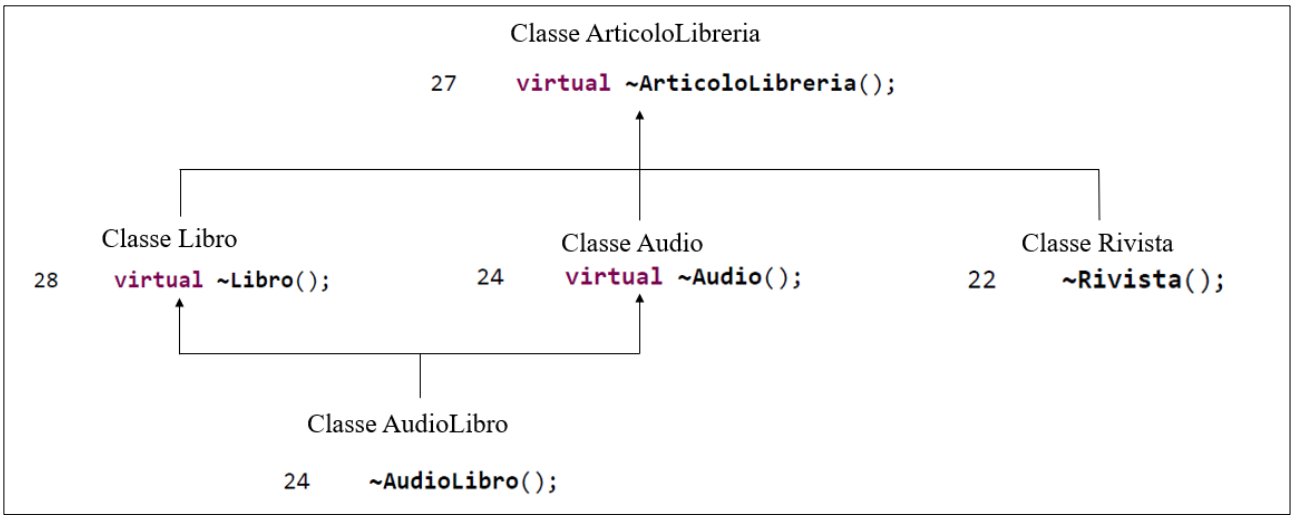


Fig. 18: distruttori virtual

Design pattern: singleton

La classe LibreriaManager è la classe che gestisce la logica del programma: inizializza la libreria digitale all'avvio del programma inserendo diversi articoli in un campo denominato *vettArticoli* e fornisce metodi utili per la manipolazione di quest'ultimi. Per come è strutturata LibreriaManager, vi è la necessità di avere un un'unica istanza di questa classe in modo da garantire che le altre classi che operano sulla libreria digitale operino sempre sulla stessa istanza di *vettArticoli*. Per questo motivo, la classe LibreriaManager è stata progettata e implementata seguendo i principi espressi dal design pattern *singleton*: esiste solo una istanza di LibreriaManager e dunque una sola istanza del campo *vettArticoli*, sul quale operano tutte le classi che utilizzano i metodi offerti da LibreriaManager. L'implementazione del design pattern singleton in C++ è riportata in *Fig. 19*.

libreriamanager.h

```
15 //singleton
16 class LibreriaManager {

18 private:
22     LibreriaManager();

24     LibreriaManager(LibreriaManager const&) = delete;
25     void operator=(LibreriaManager const&) = delete;

40 public:
41     //singleton: instance() ritorna l'unica istanza della classe
42     static LibreriaManager& instance();
```

libreriamanager.cpp

```
17 LibreriaManager& LibreriaManager::instance() {
18     // static = instanzia solo alla prima chiamata
19     static LibreriaManager instance;
20     return instance;
21 }
```

Fig. 19: design pattern singleton per la classe LibreriaManager

Singleton: copy constructor e assignment operator

Come possiamo osservare in *Fig.19*, il copy constructor e l'assignment operator della classe LibreriaManager sono posti a "=*delete*". La parola chiave *delete* in questo contesto indica al compilatore di non generare automaticamente il copy constructor e l'assignment operator per la classe in questione. Questo è in accordo con lo scopo che si cerca di raggiungere con il design pattern singleton in quanto non vogliamo che l'utente possa dichiarare un'altra variabile di tipo LibreriaManager sfruttando questi due metodi che vengono

tipicamente generati automaticamente dal compilatore. Con la parola chiave *delete* infatti si impedisce che si possano verificare le seguenti situazioni:

1. Utilizzo del copy constructor
   LibreriaManager x = LibreriaManager::instance();

2. Utilizzo dell'assignment operator
   LibreriaManager x;
   x = LibreriaManager::instance();

Singleton: internal static variables
Le internal static variables sono variabili dichiarate all'interno di una funzione con la parola chiave static prima del tipo. La caratteristica di queste variabili è che mantengono il loro stato tra una chiamata e l'altra della funzione, evitando quindi di essere re-inizializzate ogni volta che la funzione viene richiamata. Nel progetto, solo la variabile *instance* del metodo *instance()* di LibreriaManager è una internal static variables (*Fig. 19*). Questa variabile viene inizializzata solo la prima volta che il metodo *instance()* viene invocato, consentendo quindi di raggiungere lo scopo previsto dal singleton pattern con delayed creation: evitare di costruire l'istanza singleton a load time, costruendola solo quando viene invocato il metodo instance().

Singleton: return by reference
Il metodo *instance()* di LibreriaManager, nel restituire l'unica istanza della classe LibreriaManager, realizza un return by reference. Questo significa che il metodo restituisce un puntatore implicito al valore di cui si fa il return, ovvero l'unica istanza *instance* di LibreriaManager. Dato che le internal static variables vengono memorizzate nello spazio di memoria delle variabili globali (*data*), la variabile riferimento restituita dal metodo *instance()* punterà alla cella di memoria in *data* dove è memorizzata la variabile *instance.*

STL: Standard Template Library
Per l'implementazione della libreria digitale, vengono utilizzati i *vector* della STL. In particolare, la classe LibreriaManager presenta un membro *vettArticoli* che è un vector di variabili di tipo *ArticoloLibreria**. Questo vector rappresenta il nucleo dei dati del progetto, in quanto tutte le operazioni svolte sugli articoli della libreria digitale passano attraverso questo *vector*. Per poter manipolare i dati contenuti nel *vector* sono stati implementati diversi metodi in LibreriaManager che si basano sull'utilizzo degli *iterators*. In *Fig. 20* vengono presentati degli estratti del progetto nei quali si utilizzano *vector* e *iterators*.

libreriamanager.h

```
16 class LibreriaManager {
17
18 private:
19
20     vector<ArticoloLibreria*> vettArticoli;
```

libreriamanager.cpp

```
23 //return iterator a end() se non trova l'elemento cercato
24 std::vector<ArticoloLibreria*>::iterator LibreriaManager::findArticolo(int articoloID) {
25     vector<ArticoloLibreria*>::iterator cercato = vettArticoli.end();
26     for (vector<ArticoloLibreria*>::iterator i = vettArticoli.begin(); i != vettArticoli.end
   (); i++)
27         if ((*i)->getArticoloID() == articoloID)
28             cercato = i;
29     return cercato;
30 }

90 void LibreriaManager::printArticolo(int articoloID) {
91     vector<ArticoloLibreria*>::iterator cercato = findArticolo(articoloID);
92     if (cercato != vettArticoli.end()) {
93         cout << (*cercato)->toString() << endl;
94     } else {
95         cout << "Errore! Articolo non trovato." << endl;
96     }
97 }
```

Fig. 20: utilizzo vector e iterators in LibreriaManager

Template: metodi generici
Nella classe LibreriaManager sono stati ideati e implementati due metodi generici che consentono di estrarre dal *vector vettArticoli* dei "sotto-vettori" contenenti solo le istanze di uno specifico tipo tra Audio, Libro, Rivista e AudioLibro. L'utilizzo dei template rende il programma molto più flessibile in quanto con solo 2 metodi generici si consegue lo stesso risultato che si sarebbe potuto ottenere con la scrittura di 8 metodi non generici. I metodi generici utilizzati per la generazione dei "sotto-vettori" sono mostrati in *Fig. 21*.

```
16 class LibreriaManager {
17
18 private:

30     //template devono essere implementati nel .h
31     template<typename T> vector<T*> getVettSottocategoria(TipoArticolo sottocategoria) {
32         vector<T*> vettSottoCat;
33         for (vector<ArticoloLibreria*>::iterator i = vettArticoli.begin(); i !=
   vettArticoli.end(); i++)
34             if ((*i)->getTipo() == sottocategoria) {
35                 T* temp = dynamic_cast<T*>(*i);
36                 vettSottoCat.push_back(temp);
37             }
38         return vettSottoCat;
39     }


40 public:

54     template<typename T> void printSottocategoria(TipoArticolo sottocategoria) {
55         vector<T*> vett = getVettSottocategoria<T>(sottocategoria);
56         for (typename vector<T*>::iterator i = vett.begin(); i != vett.end(); i++)
57             cout << (*i)->toString() << endl;
58     }
```

Fig. 21: utilizzo template per metodi generici in LibreriaManager

Smart pointers

Nel progetto, gli smart pointers vengono utilizzati per gestire i dialogs dell'interfaccia grafica realizzata con QT Creator. In particolare, ogni volta che un nuovo dialog (finestra) viene creato, la sua gestione è affidata ad uno smart pointer che si occupa di chiamare in automatico il delete del dialog una volta terminato il suo utilizzo (ossia nel momento in cui l'utente chiude la finestra grafica puntata dallo smart pointer). Alcuni esempi relativi all'utilizzo degli smart pointers sono riportati in *Fig. 22*.

```
void MainWindow::on_inserisciLibroButton_clicked()
{
    //apro una nuova finestra (dialog) per la visualizzazione dell'inserimento nuovo libro
    unique_ptr <inserisciNuovoLibroDialog> newWindow (new inserisciNuovoLibroDialog());
    newWindow->setModal(true);
    newWindow->exec();

}

void MainWindow::on_inserisciAudioButton_clicked()
{
    unique_ptr <InserisciNuovoAudioDialog> newWindow (new InserisciNuovoAudioDialog());
    newWindow->setModal(true);
    newWindow->exec();

}
```

Fig. 22: smart pointers per la gestione dei dialogs

Higher Order Functions:

Per poter semplificare l'implementazione dell'interfaccia grafica con QT Creator, sono state ideate e implementate diverse HOF che permettono di trasformare il testo che viene stampato a stdout (console) dalla classe LibreriaManager in stringhe di tipo QString, le quali possono poi essere utilizzate per stampare informazioni sull'interfaccia grafica realizzata con QT. In particolare, le HOF utilizzate prendono sempre come parametro in input un metodo appartenente alla classe LibreriaManager e lo utilizzano per stampare a stdout

del testo che, dopo esser stato adeguatamente memorizzato in un buffer in formato string, viene convertito in QString e restituito. Alcuni esempi delle HOF utilizzate sono mostrati in *Fig. 23*.

```
//funzione di supporto per convertire da stdout (console) a QString
//argomento = funzione della classe LibreriaManager senza parametri
QString MainWindow::fromSTDtoQStringVoidArgument(void (LibreriaManager::*func)()){

    //preparo il buffer per la lettura del testo da stdout
    stringstream buffer;
    cout.rdbuf(buffer.rdbuf());
    //stampo in stdout chiamando la funzione di LibreriaManager passata come parametro
    //il buffer leggerà quello che viene stampato a stdout
    (LibreriaManager::instance().*func)();
    //trasformo il testo del buffer in una stringa
    std::string text = buffer.str();
    //converto la std::string in una Qstring
    QString res = QString::fromStdString(text);

    return res;
}

//argomento = funzione della classe LibreriaManager con parametro di tipo TipoArticolo
QString MainWindow::fromSTDtoQStringTipoArticoloArgument(void (LibreriaManager::*func)
(TipoArticolo), TipoArticolo tipoArt){
    stringstream buffer;
    cout.rdbuf(buffer.rdbuf());

    (LibreriaManager::instance().*func)(tipoArt);

    std::string text = buffer.str();
    QString res = QString::fromStdString(text);

    return res;
}
```

Fig. 23: HOF per convertire da stdout a QString