6/09/2025

# ACCESSO AI FILE – LETTURA E SCRITTURA

L'astrazione dello *stream* in C++ è un esempio perfetto di come il linguaggio concepisca la separazione fra **concetto logico** e **dettagli tecnici**. Dal punto di vista teorico, lo stream è un modello unidirezionale o bidirezionale di comunicazione: non si pensa più in termini di "chiamate di sistema per aprire e leggere un file", ma in termini di "consumare" o "produrre" un flusso di byte. L'oggetto stream diventa l'interfaccia unica che incapsula le complessità.
In C++ l'accesso ai file è modellato tramite il concetto di stream: un flusso astratto di byte che scorre fra il tuo programma e una sorgente o destinazione esterna. Per i file, la libreria `<fstream>` fornisce tre tipi di stream: `std::ifstream` per consumare dati (input), `std::ofstream` per produrli (output) e `std::fstream` per fare entrambe le cose sullo stesso handle. Questi oggetti non lavorano da soli: ciascuno è associato a un buffer, tipicamente una `std::filebuf`, che incapsula i dettagli a basso livello delle chiamate di sistema.
Il modello idiomatico in C++ è RAII, Resource Acquisition Is Initialization. Significa che la gestione della risorsa non si fa con "apri" e "chiudi" sparsi nel codice, ma legando la risorsa alla vita di un oggetto. Quando costruisci uno stream, acquisisci la risorsa (apri il file); quando l'oggetto esce dallo scope, il distruttore rilascia in modo deterministico ciò che possiede (svuota i buffer, chiude il file).
È possibile invocare `open()` e `close()` esplicitamente, ma il punto principale rimane: la responsabilità della chiusura è automatica e coincide con la fine della vita dell'oggetto.
Aprire un file non è un'operazione puramente logica: coinvolge il sistema operativo ed è fallibile. Il fallimento non si esprime con un'eccezione di default, ma con lo stato interno dello stream. Subito dopo l'apertura conviene verificare l'esito tramite `is_open()`, che risponde alla domanda "l'associazione con un file è effettivamente attiva?".
Un aspetto teoricamente interessante è la combinazione di modalità di apertura, che definisce la semantica dello stream. La modalità `in` autorizza la lettura; `out` abilita la scrittura; `app` vincola il puntatore di scrittura alla coda, garantendo che ogni operazione aggiunga dati senza intaccare il contenuto preesistente; `trunc` ordina di troncare il file all'apertura, azzerandone la lunghezza; `ate` posiziona subito il puntatore alla fine, ma senza impedire spostamenti successivi; `binary` chiede al sistema di non applicare alcuna trasformazione di testo. I default riflettono le intenzioni tipiche: un `ifstream` nasce per leggere e quindi usa `in`; un `ofstream` nasce per produrre nuovo contenuto e, se non gli dici altro, sovrascrive (`out|trunc`); uno `fstream` è neutro e pretende che tu dichiari cosa vuoi fare. Questi flag non sono meri interruttori: definiscono il contratto con il file. Ad esempio, `app` non è equivalente a "vai alla fine e scrivi": è una garanzia di append atomico, mentre `ate` è solo una posizione iniziale del cursore.
La lettura e la scrittura avvengono su due piani, formattato e non formattato. Il piano formattato usa gli operatori >> e << e i manipolatori per convertire automaticamente tra tipi C++ e rappresentazioni testuali; è comodo ma dipende dalla locale e dai separatori, quindi è sensibile al layout del testo. Il piano non formattato lavora a livello di byte o blocchi (`get`, `read`, `put`, `write`) e non fa alcuna interpretazione: è quello da usare quando contano performance, controllo e riproducibilità.

## FUNZIONI MEMBRO

Le **funzioni membro** in C++ sono funzioni dichiarate all'interno di una classe e servono a definire il **comportamento** degli oggetti di quella classe. Se le variabili membro descrivono lo stato dell'oggetto (cioè i suoi dati), le funzioni membro stabiliscono cosa quell'oggetto può fare.
Quando scrivi una classe, in realtà stai progettando un modello che unisce **dati** e **operazioni**. Le funzioni membro sono le operazioni. A differenza delle funzioni "libere" (cioè quelle dichiarate fuori da una classe), le funzioni membro hanno un legame diretto con l'istanza della classe su cui vengono invocate. Questo è reso possibile dal puntatore implicito `this`, che viene passato in automatico a ogni funzione membro e che permette di accedere ai dati di quell'oggetto specifico. Per esempio, se hai una classe `Cerchio` con una variabile `raggio`, potresti definire una funzione membro `area()` che calcola e restituisce l'area del cerchio. Quando chiami `mioCerchio.area()`, quella funzione viene eseguita prendendo il valore del `raggio` contenuto in quell'oggetto particolare.

Dal punto di vista concettuale, le funzioni membro sono fondamentali perché: realizzano l'**incapsulamento**, cioè nascondono i dettagli interni e offrono all'esterno solo i metodi necessari; consentono di collegare in modo naturale dati e comportamenti; permettono di progettare oggetti che non sono semplici contenitori di variabili, ma entità attive con responsabilità e azioni proprie.

### Puntatori a funzioni libere

Una **funzione libera** è una funzione dichiarata fuori da qualsiasi classe o struct, cioè indipendente. In C e C++ è possibile creare un puntatore che memorizza l'indirizzo di una di queste funzioni. Questo puntatore può poi essere usato per invocare la funzione in maniera indiretta.

Dal punto di vista teorico, il puntatore a funzione libera è un semplice indirizzo in memoria che indica dove inizia il codice di quella funzione. Non ha alcun contesto, non "sa" nulla di oggetti o classi: riceve solo i parametri espliciti e li passa alla funzione quando viene chiamata.

Ex:

```
#include <iostream>

void saluta() {
   std::cout << "Ciao dal puntatore a funzione libera!" << std::endl;
}

int main() {
   void (*p)();   // dichiarazione di puntatore a funzione libera
   p = &saluta;   // assegno l'indirizzo della funzione
   p();          // invocazione indiretta
}
```

## PUNTATORI A FUNZIONI MEMBRO

Un **puntatore a funzione membro** in C++ è un tipo particolare di puntatore che non memorizza solo l'indirizzo di una funzione, ma anche l'informazione che quella funzione appartiene a una certa **classe**. Questo è necessario perché, a differenza di una funzione "libera", una funzione membro non è indipendente: viene sempre invocata su un oggetto specifico della classe a cui appartiene, cioè ha un **parametro implicito** chiamato `this`.

### Sintassi:

```
R (Classe::*nome_puntatore)(parametri);
```

dove:

- **R** è il tipo di ritorno della funzione membro,
- **Classe** è il nome della classe a cui appartiene la funzione,
- **parametri** è la lista dei tipi dei parametri della funzione,
- **nome_puntatore** è il nome della variabile puntatore.

Se una funzione membro è dichiarata `const`, il puntatore deve essere dichiarato come `void (Classe::*)() const`.

```
(HarlObj.*ptr)();    // se hai un oggetto
(HarlPtr->*ptr)();   // se hai un puntatore all'oggetto
```

Un **function pointer normale** `(void (*p)())` punta a una funzione che non ha legame con nessuna classe. Puoi chiamarla senza contesto.

Un **member function pointer** `(void (Classe::*)())` punta a una funzione che richiede un oggetto di tipo `Classe` per essere chiamata.

```
#include <iostream>

class Harl {
public:
   void parla() {
      std::cout << "Ciao dal puntatore a funzione membro!" << std::endl;
   }
};

int main() {
   Harl h;
   void (Harl::*p)();   // dichiarazione puntatore a funzione membro di Harl
   p = &Harl::parla;    // assegno la funzione membro

   (h.*p)();          // invoco sul mio oggetto
}
```

**SWITCH**

Nel C++98, lo `switch` è un costrutto di selezione multipla che permette di eseguire blocchi di codice diversi in base al valore di un'unica espressione. La caratteristica fondamentale è che l'espressione all'interno dello `switch` deve essere di tipo intero o di tipo enumerativo. Non è quindi possibile usare stringhe, tipi floating point o oggetti complessi: questa è una limitazione precisa del C++98. Se si prova, ad esempio, a scrivere `switch` su un `double` o su una `std::string`, il compilatore genera un errore.

La logica dello `switch` si basa sul confronto diretto tra il valore dell'espressione e una serie di etichette, i `case`, che devono essere costanti valutabili a tempo di compilazione. In C++98 non si possono avere valori dinamici o espressioni complesse nei `case`, ma solo costanti letterali o costanti simboliche definite con `#define` o `const`. Questo rafforza la natura deterministica dello `switch`, che funziona come una tabella di salti fissi calcolata dal compilatore. Il flusso del programma nello `switch` si comporta in modo particolare: quando il valore corrisponde a un `case`, l'esecuzione entra in quel punto e prosegue sequenzialmente anche nei casi successivi, a meno che non ci sia un `break`. Questo meccanismo, noto come *fall-through*, è parte integrante del C++98 ed è molto utile quando si vogliono raggruppare più valori sotto la stessa azione. Ad esempio, si possono scrivere più `case` uno dopo l'altro senza codice e farli terminare con un unico blocco di istruzioni, che verrà eseguito per tutte quelle alternative. Tuttavia, se il programmatore dimentica un `break` senza volerlo, il flusso continua nel `case` successivo ed esegue codice non previsto, creando errori logici difficili da trovare. Per questo, nelle buone pratiche di programmazione in C++98 si sottolinea sempre l'importanza di inserire i `break` in modo esplicito. Un altro elemento centrale è la clausola `default`. In C++98 essa non è obbligatoria, ma è fortemente consigliata. Quando è presente, raccoglie tutti i valori dell'espressione che non corrispondono a nessun `case`. È l'equivalente dell'`else` in una catena di condizioni, ma inserito nello schema compatto dello `switch`. La sua presenza aumenta la robustezza del codice, perché garantisce un comportamento anche di fronte a valori inattesi.  Dal punto di vista teorico, lo `switch` in C++98 non è un'alternativa generalizzata all'`if`. È piuttosto uno strumento pensato per la gestione di scelte multiple su valori discreti e finiti. La sua utilità principale emerge quando si ha un singolo valore da confrontare con molte alternative, come nel caso di menu numerici, gestione di comandi enumerati o risposta a stati codificati. In questi scenari, la leggibilità e l'efficienza del codice risultano molto superiori rispetto a una lunga catena di `if` ed `else if`.