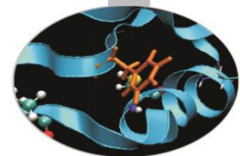
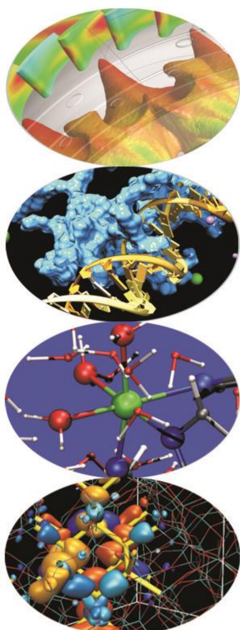
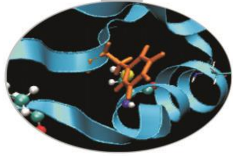
CINECA SCAI
SuperComputing Applications and Innovation



I/O da FILE



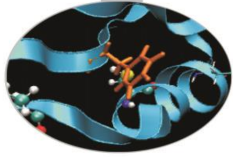
CINECA



Indice

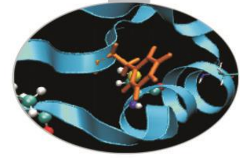
- **Gli stream**
- **Classi, oggetti e librerie per l’I/O**
- **cout, cin**
- **I manipolatori di stream, la libreria <iomanip>**
- **I flag di formattazione**
- **Gerarchia di dati**
- **Creazione di file**
- **Apertura di file**
- **Chiusura di file**
- **File ad accesso sequenziale**
- **File di numeri**
- **File ad accesso diretto**

Gli stream

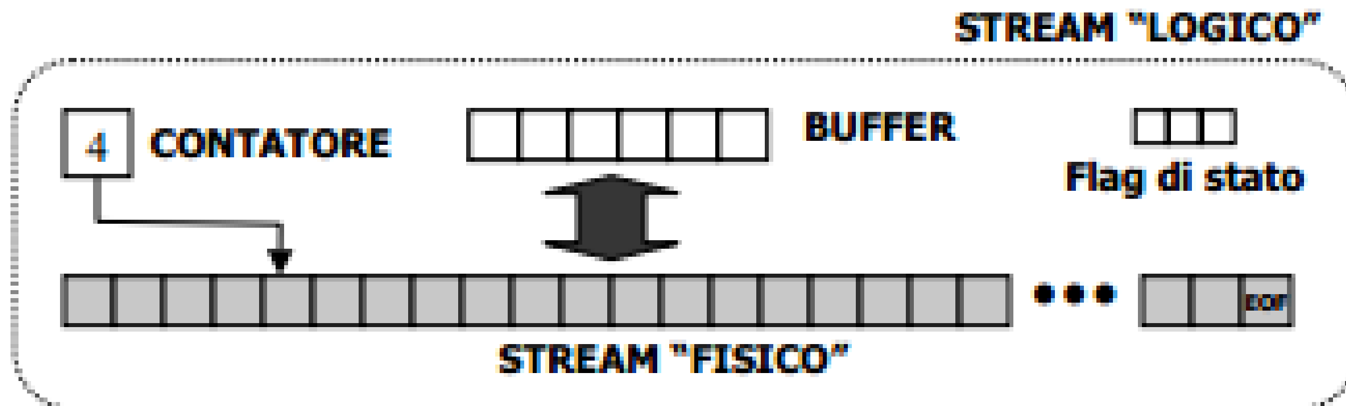


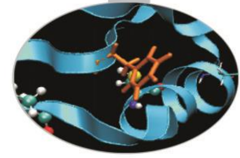
- Alla base delle operazioni di I/O, in C++, sta il concetto di stream (flusso) di byte tra la memoria principale ed i dispositivi di input (la tastiera, i file di sola lettura) e quelli di output (il video, i file di scrittura).
- Possiamo distinguere tra due modalità di I/O: formattato, ovvero ad alto livello, leggibile dall'utente e non formattato cioè a basso livello, comprensibile solo dalla macchina.
- La modalità non formattata è preferibile quando si debba trattare con grandi moli di dati.

Gli stream

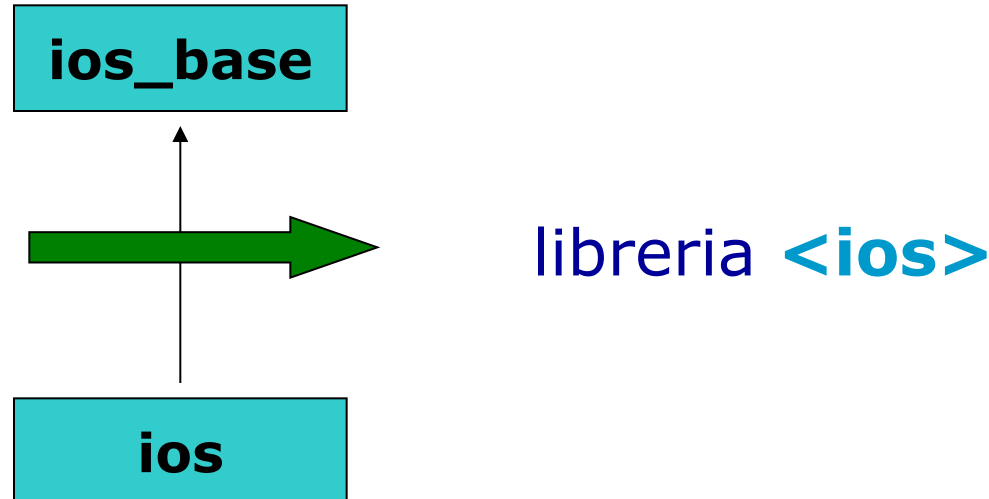


- Uno stream è un'astrazione che si riferisce a un flusso di dati da un'origine (produttore) a una destinazione (consumatore)



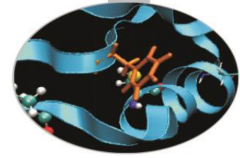


Classi, oggetti e librerie per l'I/O

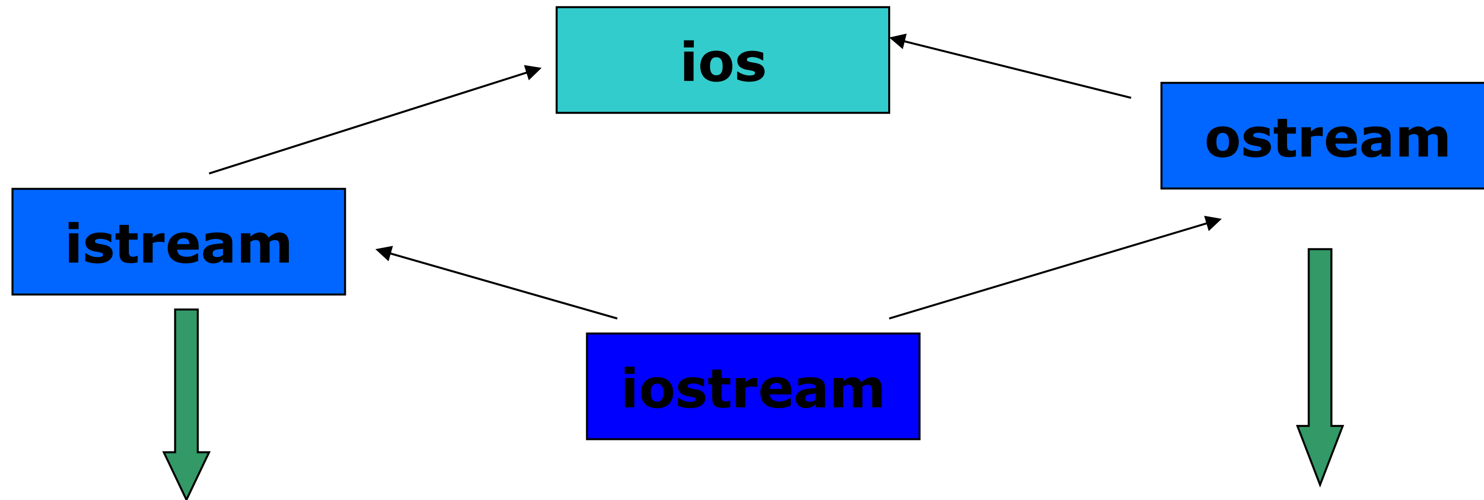


ios_base ed **ios** rappresentano le classi base per eccellenza, da cui derivano tutte le altre classi preposte all’I/O.

Nei compilatori più recenti la classe **ios** ha inglobato la classe ios_base.



Gli stream

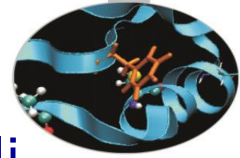


libreria **<istream>**, contiene la dichiarazione dell’oggetto **cin** per la lettura da standard input.

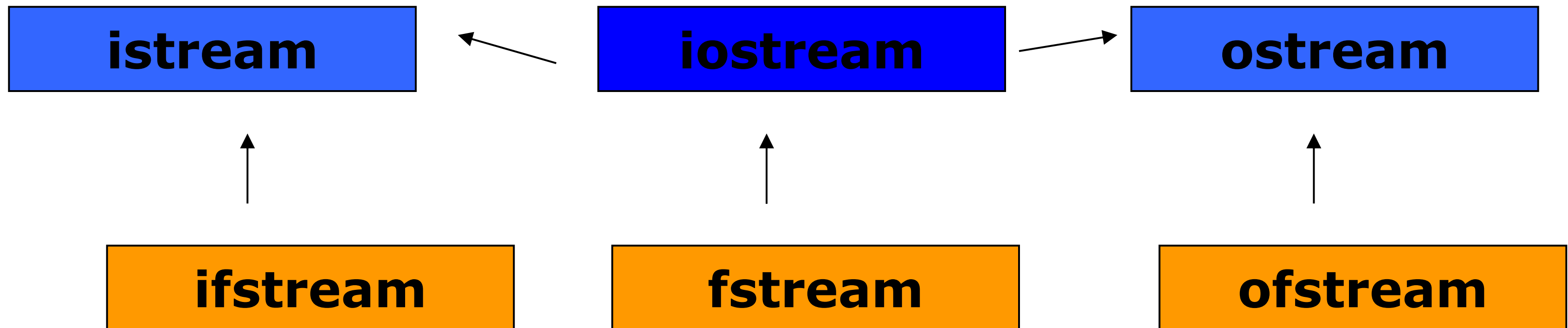
libreria **<ostream>**, ha in sé la dichiarazione degli oggetti **cout**, **cerr** e **clog** per la scrittura su standard output e standard error.

libreria **<iostream>**, contiene la dichiarazione di tutti e quattro gli oggetti sopra citati. E’ l’unica libreria da includere nel codice per eseguire operazioni di I/O su standard device.

Gli stream

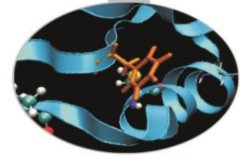


La classe **fstream** gestisce operazioni sia di input che di output da file.



La classe **ofstream** gestisce operazioni di output su file
La classe **ifstream** gestisce operazioni di input da file

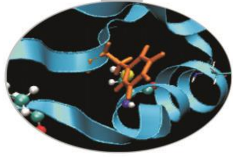
libreria **<fstream>**. Da includere nel codice per eseguire ogni operazione di I/O su file. Non contiene la dichiarazioni di particolari oggetti.



Gli stream

La gerarchia delle **classi**

```
                 ios_base
                    ↑
                   ios
          ↗                  ↖
     istream                ostream
     ↑      ↖            ↗       ↑
ifstream       iostream        ofstream
                  ↑
               fstream
```



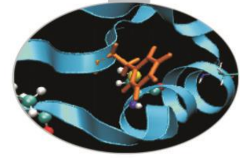
cout

- E' un oggetto della classe ostream; la sua dichiarazione risiede all'interno della libreria <ostream>.
- Tipicamente viene utilizzato insieme con l'operatore di inserimento nello stream `<<`; può far uso del metodo pubblico `put(char)` di ostream, per la scrittura di un singolo carattere su standard output.

- esempio: uso di `<<` e di `put(char)` con `cout`.

```
#include<iostream>
using namespace std;

int main(){
        cout << "Hi!" << endl;
        cout.put('H').put('i').put('!').put('\n');
        return 0;}
```

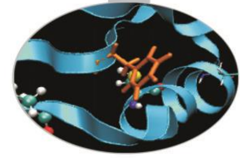


cout

- `cout`, così come ogni altro oggetto delle classi ostream ed fstream, può servirsi di vari metodi pubblici della classe **ios** per la formattazione dell'output. Lo stile di formattazione scelto è valido per ogni istruzione che appartiene al blocco in cui il metodo è stato chiamato.

Tra le funzioni di formattazione principali possiamo citare:

`precision(int)`	indica il numero totale delle cifre (decimali e non) con cui scrivere un double. Se chiamata senza argomento restituisce l'impostazione corrente;
`width(int)`	ampiezza di campo per l'output del valore di una variabile. Senza argomento restituisce il valore corrente;
`setf(ios::nome_flag)`	attiva il flag di formattazione nome_flag;
`unsetf(ios::nome_flag)`	disattiva il flag di formattazione nome_flag;
`flags()`	attiva uno o più flag di formattazione. Restituisce un long che specifica lo stato dei flag attivati;
`fill(char)`	specifica i carattere di riempimento dei campi giustificati. Se non è specificato alcun argomento, verrà utilizzato il carattere di default (blank space);



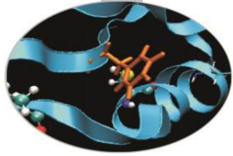
cout

- esempio: uso delle funzioni width, fill e precision .

```
#include<iostream>
using namespace std;
int main(){
   double a=11, b=3;
   for(int i=0; i<5; i++){
        cout.width(10);
        cout.fill('*');
        cout.precision(i+1);
        cout << a/b << endl;
   }
   return 0;
}
```

- output:

```
*********4
*******3.7
******3.67
*****3.667
****3.6667
```



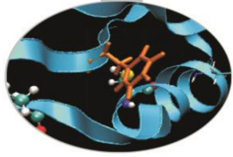
cin

- E’ un oggetto della classe istream; la sua dichiarazione è posta all’interno della libreria <istream>.
- Viene utilizzato, di solito, insieme all’operatore di estrazione dallo stream >>, ma può anche chiamare alcuni dei metodi pubblici della classe istream, come: get(), per la lettura di un carattere); getline(char*), per la lettura di una stringa in cui è ammesso anche il carattere di spazio; eof(), che restituisce 1 o 0 (true o false) a seconda che sia stato raggiunto o meno il carattere di End Of File (EOF).
- Per mezzo della funzione width(int) della classe **ios** è possibile definire l’ampiezza del campo di input.

Esempio: uso di get().

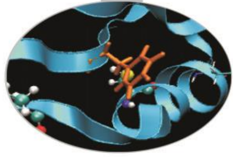
```
#include<iostream>
using namespace std;
int main(){
     char c;
     cout << "Insert a sentence:" << endl;
     while( (c=cin.get()) != EOF)
        cout << c;
     return 0;}
```

I manipolatori di stream



- Sono funzioni che servono per controllare la formattazione dei dati nelle procedure di I/O e rappresentano un'alternativa alle funzioni della classe ios viste in precedenza.
- Si dividono in due categorie:
  - semplici;
  - parametrizzati.
- La differenza sta nel fatto che i manipolatori parametrizzati richiedono un argomento, quelli semplici no.
- Per fare uso dei manipolatori parametrizzati è necessario includere la libreria **<iomanip>**; per i manipolatori semplici è sufficiente <iostream>.
- In generale ogni manipolatore può essere visto come un operando degli operatori << e >> che influenza la stampa o la lettura degli oggetti e delle variabili che li seguono nell'istruzione in cui compaiono.
- esempio:

```
cout << n << " " << hex << n << endl;
cout << n << " " << setbase(16) << n << endl;
```



La libreria <iomanip>

- E’ costituita da sei differenti manipolatori parametrici:

  - **resetiosflags(**ios::*nome_flag***)**: annulla il flag di formattazione ios::*nome_flag* attivato in precedenza;

  - **setiosflag(**ios::*nome_flag***)**: attiva il particolare flag di formattazione ios::*nome_flag*;

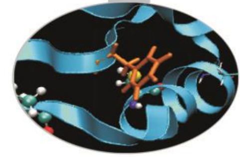
  - **setbase(**int**)**: impone la scrittura di numeri in una determinata base. I valori consentiti sono 8, 10 e 16;

  - **setfill(**char**)**: indica il carattere di riempimento per i campi giustificati;

  - **setprecision(**int**)**: determina il numero totale delle cifre (intere e decimali) con cui deve essere scritto un numero reale;

  - **setw(**int**)**: specifica il numero minimo di caratteri da utilizzare nella scrittura della successiva espressione.

La libreria <iomanip>

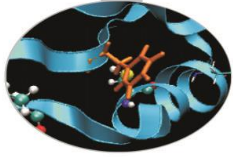


Esempio: uso di setw, setfill e setprecision.

```
#include<iostream>
#include<iomanip>
using namespace std;

int main(){
   double a=11, b=3;
   for(int i=0; i<5; i++){
        cout << setw(10) << setfill('*')
             << setprecision(i) << a/b << endl;
   }
   return 0;
}
output:
*********4
*********4
*******3.7
******3.67
*****3.667
```

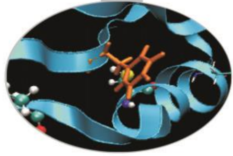
I flag di formattazione



- Sono definiti tramite l'istruzione enum all'interno della classe **ios** (ios_base per i compilatori più datati).
- Vengono usati come argomento da passare a due dei manipolatori parametrici ed alle funzioni della classe ios che si occupano della formattazione dell'output.
- I principali sono:

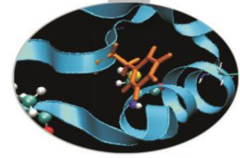
```
ios::skipws        salta gli spazi bianchi in input
ios::left          giustifica a sinistra
ios::right         giustifica a destra
ios::internal      giustifica il segno di un numero a sinistra
                   ed il suo valore a destra
ios::dec           interi in base decimale
ios::oct           interi in base ottale
ios::hex           interi in base esadecimale
ios::showbase      mostra la base di un intero (0 per ottale
                   0x per esadecimale)
```

I flag di formattazione



ios::showpoint	punto decimale sempre presente nel formato dei numeri reali
ios::uppercase	lettera maiuscola E nella notazione scientifica e 0X per la base esadecimale
ios::showpos	mostra il segno di un numero
ios::scientific	notazione scientifica
ios::fixed	numero fisso di cifre decimali
ios::floatfield	formato di default per numeri reali(numero variabile di cifre decimali)

I flag di formattazione



- Esempio1: flag di formattazione e funzione `setf`.

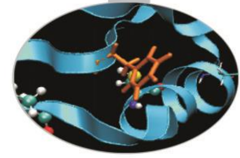
```
#include<iostream>
using namespace std;
int main(){
   double a=3;
        for(int i=0; i<5; i++){
        cout.setf(ios::showpoint | ios:: fixed);
        cout.setf(ios::right  | ios::showpos
                 |ios::internal);
        cout << (i+1)/a << endl;
   }
   return 0;
}
```

- output:

```
+0.333333
+0.666667
+1.000000
+1.333333
+1.666667
```

SuperComputing Applications and Innovation

I flag di formattazione



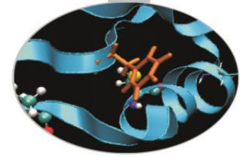
- Esempio2: flag di formattazione e manipolatore `setiosflags`

```
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
   double a=3;
   for(int i=0; i<5; i++){
        cout << setiosflags(ios::showpoint | ios::fixed)
             << setiosflags(ios::right | ios::showpos
                           |ios::internal)
             << (i+1)/a << endl;
   }return 0;
}
```

- output:

```
+0.333333
+0.666667
+1.000000
+1.333333
+1.666667
```

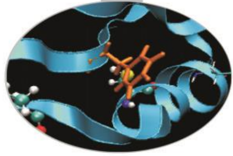
Gerarchia di dati



- I dati che possono essere trattati da un computer seguono una precisa gerarchia alla base della quale sta il *bit* che, come sappiamo, può assumere soltanto i valori 0 ed 1.
- Un insieme di 8 bit costituisce un *byte* che permette di codificare ogni *carattere*. Per esempio il byte 01001101 codifica la lettera M.
- Un gruppo di caratteri definisce un *campo*. Ad ogni campo è associata una precisa informazione. La parola "Mario" è un esempio di campo.
- Un insieme di più campi costituisce un *record*. Tipicamente, in C++ le classi e le strutture sono esempi di record.
- Un gruppo di record correlati dà origine ad un *file*.
- La gerarchia dei dati si può rappresentare nel seguente modo:

  bit → byte/carattere → campo → record → file

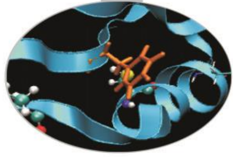
- In ogni record esiste un campo speciale, detto *chiave del record*, che identifica univocamente il record stesso e ne facilita il recupero.



Creazioni di file

- Un file è trattato dal C++ come uno *stream* (flusso) sequenziale di byte.
- Quando un file viene aperto, ad esso è associato automaticamente uno stream che rappresenta un canale di comunicazione tra il file stesso ed il programma.
- La fine di un file è segnata da un *marcatore di end of file* o da uno specifico numero di byte registrato in una struttura dati gestita dal sistema.
- Per far uso di file è necessario includere all'interno del programma l'header file `<fstream>` che contiene la definizione delle classi `ifstream` (input da file), `ofstream` (output da file) e `fstream` (input/output) da file.
- L'**apertura** di un nuovo file richiede la **creazione di un oggetto di una classe stream**.
- Al costruttore della classe selezionata vengono inviati il nome del file ed, eventualmente, la modalità di apertura che coincide con un metodo della classe **ios**:

```
xfstream nome_oggetto(nome_file,ios::modalità_apertura);
```



Apertura di file

- Ogni file può essere aperto in due modi differenti: uno ricalca la regola generale di sintassi per la creazione di file vista in precedenza, l'altro fa uso della funzione membro **open** presente in ognuna delle classi stream.
- **File di output**
- L'apertura di un file di output, associato all'oggetto outFile per es., può essere realizzata attraverso la notazione:

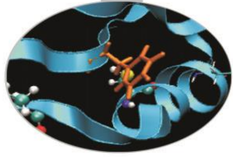
```
ofstream outFile ("nome_file.dat", ios::out);
```

- oppure usando la funzione membro **open**(*lista_argomenti*) della classe ofstream:

```
ofstream outFile; // dichiarazione dell'oggetto di classe ofstream
outFile.open("nome_file.dat", ios::out);
```

- Il parametro **ios::out** indica che la modalità di apertura del file è "in sola scrittura". Di default, ogni oggetto appartenente alla classe ofstream gode di questa proprietà.

Apertura di file



- **File di input**

Un file di input è associato ad un oggetto della classe ifstream (inpFile per es.) e viene dichiarato seguendo la regola generale:

```
ifstream inpFile("nome_file.dat", ios::in);
```

oppure:

```
ifstream inpFile; // dichiarazione dell'oggetto di classe ifstream
    inpFile.open("nome_file.dat", ios::in);
```

Il parametro **ios::in** indica che la modalità di apertura del file è "in sola lettura". Ogni oggetto della classe ifstream è creato, di default, con questa modalità.

- **File di input/output**

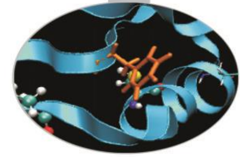
I file di input/output sono associati ad oggetti della classe fstream (ioFile per es.) e vengono aperti nel modo seguente:

```
fstream ioFile("nome_file.dat", ios::in|ios::out);
```

ovvero:

```
fstream ioFile; // dichiarazione dell'oggetto di classe fstream
    ioFile.open("nome_file.dat", ios::in|ios::out);
```

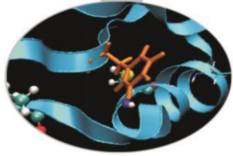
Chiusura di file



- Tutti i file vengono chiusi *automaticamente* quando il programma termina.
- Se l'oggetto associato ad un file ancora aperto viene distrutto, il file è chiuso automaticamente quando il *distruttore* viene invocato.
- E', tuttavia, possibile chiudere esplicitamente un file utilizzando la funzione membro **`close()`**, comune a tutte le classi stream, ad es.

```
outFile.close();
inpFile.close();
```

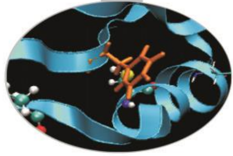
- Le modalità di apertura e chiusura dei file sono le stesse indipendentemente dal fatto che il file sia ad accesso *sequenziale* o ad accesso *diretto*.



Modalità apertura dei file

• `ios::app`	Aggiunge l'output alla fine del file.
• `ios::ate`	Apre un file e si sposta alla fine di esso
• `ios::in`	Apre un file in input.
• `ios::out`	Apre un file in output.
• `ios::trunc`	Elimina il contenuto del file se esiste. Di default si comporta così anche ios::out.
• `ios::nocreate`	Se il file non esiste, l'operazione open fallisce.
• `ios::noreplace`	Se il file esiste, l'operazione open fallisce.

File Sequenziali / accesso diretto



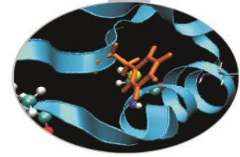
Sequenziale:

- ✓ Uno stream di testo è costituito da una sequenza di caratteri ed è organizzato in linee
- ✓ ciascuna linea è terminata dal carattere '\n' (newline)
- ✓ la terminazione del file è indicata dal carattere speciale EOF

Accesso diretto:

- ✓ Uno stream binario è costituito da una sequenza di byte

File ad accesso sequenziale



- Nei file ad accesso *sequenziale*, in prima approssimazione, possiamo dire che i record sono acceduti *nell'ordine in cui sono stati scritti* dal primo fino all'ultimo.
- La scrittura di dati su file ad accesso sequenziale avviene semplicemente per mezzo dell'operatore di inserimento nello stream **<<**, ad.es:

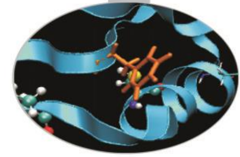
```
outFile << nome_variabile1 << nome_variabile2 << endl;
```

- La lettura da file ad accesso sequenziale si avvale, invece, dell'operatore di estrazione dallo stream **>>**:

```
inpFile >> nome_variabile1 >> nome_variabile2;
```

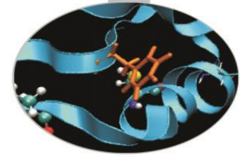
- L'uso degli operatori **>>** e **<<** è, sostanzialmente, ciò che permette di capire che un file è ad accesso sequenziale ed implica l'utilizzo di un modello ***formattato*** di input/output ove *la dimensione dei record non è costante*.

File ad accesso sequenziale



- Per recuperare un dato in maniera sequenziale, il programma dovrebbe tutte le volte cominciare a vagliare i dati uno dopo l'altro, a partire da quello iniziale, finché non trova quello desiderato.
- Per velocizzare queste operazioni, le classi istream e ostream forniscono metodi che permettono di conoscere e di modificare il valore associato al *puntatore di posizione del file*, ovvero il numero d'ordine (0,1,2,…,n; di tipo *long int*) del byte corrispondente alla locazione di memoria, attualmente puntata sul file, dalla quale leggere o sulla quale scrivere.
- La classe `istream` mette a disposizione le funzioni membro `tellg` e `seekg` (**get**). La prima serve per conoscere la locazione di memoria attualmente puntata sul file e la seconda consente di specificare la posizione da cui deve cominciare la successiva operazione di input.
- Analogamente la classe ostream annovera fra le sue funzioni membro `tellp` e `seekp` (**put**).

File ad accesso sequenziale



- La sintassi seguita per l'uso delle funzioni seekg e seekp è la seguente:

```
nome_oggetto.seekx(numero_byte, direzione_ricerca);
```

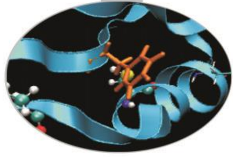
- Le modalità di posizione possibili, associate alla direzione di ricerca, sono:

`ios::beg`	posizionamento relativo all'inizio dello stream (default);
`ios::cur`	posizionamento relativo alla locazione corrente;
`ios::end`	posizionamento relativo alla fine dello stream.

- **Esempi:**

`inpFile.seekg(0);`	posizionamento all'inizio del file associato a inpFile
`inpFile.seekg(n);`	posizionamento sull'n-simo byte del file
`inpFile.seekg(n, ios::cur);`	posizionamento in avanti di n byte dalla posizione corrente
`inpFile.seekg(n, ios::end);`	posizionamento all'indietro di n byte dalla fine del file
`inpFile.seekg(0, ios::end);`	posizionamento alla fine del file

File ad accesso sequenziale



- Più semplice è la sintassi da seguire nell'utilizzo delle funzioni tellg e tellp:

```
nome_variabile=nome_oggetto.tellx();
```

Esempio:

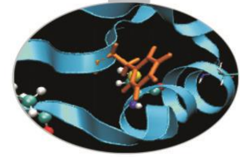
```
 long int locazione;
ifstream inpFile;
inpFile.open("input_data.dat", ios::in);
locazione=inpFile.tellg();
```

- Nel modello formattato i campi ed i record hanno dimensione variabile, dunque l'aggiornamento di un record può causare problemi di sovrascrittura. Per evitare questo inconveniente, senza rinunciare al modello formattato, è necessario riscrivere l'intero file.

Esempio:

aggiornare il record: *20 marzo 2006* in *20 maggio 2006* provoca la sovrascrittura del 6 di 2006 sul record successivo.

Esempio

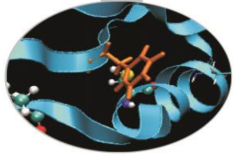


esempio: scrittura e rilettura di oggetti della classe MyClass da file formattato

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;
class MyClass{
   friend ostream& operator<<(ostream&, const MyClass&);
   friend istream& operator>>(istream&, MyClass&);
   private:
   int a;
   char c;
   string s;
   public:
   MyClass(){a=1; c='a'; s="myObject";}
   void setInt(int a){this->a=a;}
   void setChar(char c){this->c=c;}
   void setString(string s){this->s=s;}
};
ostream& operator<<(ostream& out, const MyClass& obj){
   out << obj.a << " " << obj.c << " " << obj.s << endl;
   return out;
}
```

SCAI SuperComputing Applications and Innovation

Esempio

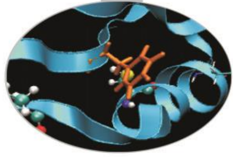


```
istream& operator>>(istream& in, MyClass& obj){
   in >> obj.a >> obj.c >> obj.s;
   return in;
}
int main(){
   MyClass A;
   MyClass B;
   fstream data("data.dat",ios::in|ios::out);
   A.setString("uno");
   B.setInt(2);
   B.setChar('b');
   B.setString("due");
   data << A;
   data << B;
   cout << data.tellg() << endl;
   data.seekg(0);
   MyClass obj;
   while(data >> obj){
         cout << obj;
   }
   return 0;
}
```

- output:

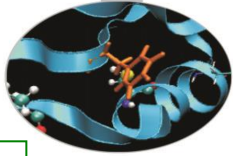
```
16
1 a uno
2 b due
```

File numerici ad accesso sequenziale



- L'uso degli operatori >> e << permette di convertire in maniera trasparente i dati numerici per metterli sotto forma di caratteri in un file.
- Tuttavia sovente nell'utilizzo di file di dati numerici si desidera controllare in maniera accurata il formato dei dati. A tal fine è necessario utilizzare delle conversioni esplicite.
- Come abbiamo visto, è possibile realizzare conversioni esplicite attraverso le funzioni setf ed unsetf o i manipolatori di stream congiuntamente con i flag di formattazione.

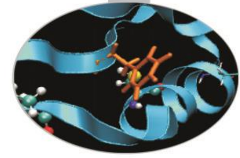
Esempio: formato



```
#include<fstream>
#include<iomanip>
int main(){
   int intero=12345;
   double decimale=12.345;
   ofstream outFile;
   outFile.open("myfile.dat",ios::out);
   outFile << setiosflags(ios::left) << setw(15) <<"Numero" << setw(15)
           << endl;
   outFile << setw(8) <<setfill('*') << setiosflags(ios::right)
           << setiosflags(ios::showbase) << intero << endl;
   outFile << setw(5) << hex << setfill('^') << setiosflags(ios::right )
           << setiosflags(ios::showbase) << intero << endl;
   outFile << setw(7) << setbase(8) << setfill('%')
           << setiosflags(ios::right) << setiosflags(ios::showbase)
           <<intero << endl;
   outFile << setw(10) << setiosflags(ios::internal | ios::showpos |
           ios::fixed) << setfill('$') << decimale << endl;
   outFile << setw(10) << setiosflags(ios::scientific) << setfill('$')
           << decimale << endl;
   return 0;
}
```

- myfile.dat:

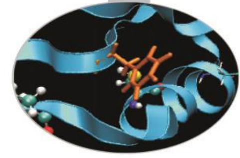
```
Numero
***12345
0x3039
%030071
+12.345000
+12.345$$$
```



Esempio

- Esempio2: scrittura su file di una matrice e di due vettori

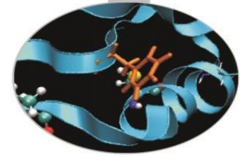
```
#include<fstream>
#include<iomanip>
using namespace std;
int main(){
   double mat[3][3];
   double vect[]={5,6,7};
   double prod[3];
   for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
               mat[i][j]=i+1+(static_cast<double>(j)+2)/10;
        }
   }
   for(int i=0; i<3; i++){
        prod[i]=0;
         for(int j=0; j<3; j++){
                prod[i] += mat[i][j]*vect[j];
         }
   }
   fstream matrice;
   matrice.open("mat.dat",ios::out);
```



Esempio

```
matrice << "Prodotto matrice-vettore" << endl;
   matrice << endl;
   matrice << setw(30) << setiosflags(ios::right) << "matrice"
           << setw(10) << "vettore" << setw(10) << "risultato" << endl;

   for(int i=0; i<3; i++){
       for(int j=0; j<3; j++){
           matrice.width(10);
           matrice.precision(4);
           matrice.setf(ios::right|ios::showpoint);
           matrice << mat[i][j]; }
       matrice.width(10);
       matrice.precision(4);
       matrice.setf(ios::right|ios::showpoint);
       matrice << vect[i];
       matrice.width(10);
       matrice.precision(4);
       matrice.setf(ios::right|ios::showpoint);
       matrice << prod[i] << endl;
   }
   return 0;
}
```



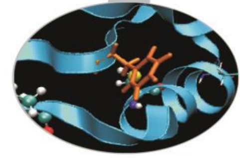
Esempio

mat.dat:

Prodotto matrice-vettore

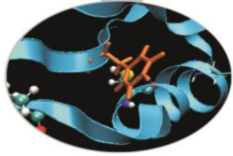
matrice			vettore	risultato
1.200	1.300	1.400	5.000	23.60
2.200	2.300	2.400	6.000	41.60
3.200	3.300	3.400	7.000	59.60

File ad accesso diretto



- Nei file ad accesso *diretto* (o *casuale*) i singoli record hanno lunghezza fissa e possono essere acceduti direttamente, senza dover passare per gli altri record. L'I/O di un file ad accesso diretto è *non formattato*.
- La posizione esatta di un record in byte (numero d'ordine), relativamente all'inizio del file, può essere calcolata *con esattezza* moltiplicando la dimensione del record per la posizione sul file del record cercato diminuita di un'unità (vedi esempio).
- In un file ad accesso diretto i dati possono essere aggiornati senza problemi di sovrascrittura. Non è più necessario riscrivere l'intero file.
- Anche l'inserimento di nuovi dati o la cancellazione di vecchi non va a danneggiare gli altri dati presenti nel file.
- Le funzioni `tellp e seekp` sono di grande utilità quando si usano file ad accesso diretto.

File ad accesso diretto



- La scrittura su un file ad accesso diretto avviene tramite la funzione membro write della classe ostream, che segue una sintassi del tipo:

```
nome_oggetto.write(lista_argomenti);
```

- La lista di argomenti della funzione write si compone essenzialmente di due parametri, il primo dei quali rappresenta il dato da scrivere e deve essere di tipo ***const char****, il secondo, invece, è un intero e coincide con la dimensione del dato stesso (facilmente calcolabile facendo uso della funzione sizeof).

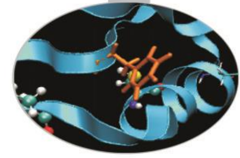
Se si vuole mettere su file un dato che non è di tipo const char*, bisogna convertirlo a puntatore a carattere a locazione costante attraverso l'operatore **reinterpret_cast**:

```
reinterpret_cast<const char*>(&nome_dato)
```

che prende come argomento un indirizzo di memoria poiché esegue un casting a puntatore.

Esempio:

```
ofstream outFile("output_data.data",ios::out);
int numero = 20;
outFile.write(reinterpret_cast<const
   char*>(&numero),sizeof(int));
```



File ad accesso diretto

- La lettura di dati da un file ad accesso diretto viene effettuata utilizzando la funzione membro read della classe istream:

```
nome_oggetto.read(lista_argomenti);
```

- ove la lista argomenti si compone del nome del dato da leggere, che deve essere di tipo ***char****, e della dimensione del dato stesso, di tipo int.
- Qualora si debbano leggere dei dati che non siano di tipo **char***, essi vanno convertiti a puntatori a carattere attraverso l'operatore:

```
reinterpret_cast<char*>(&nome_dato)
```

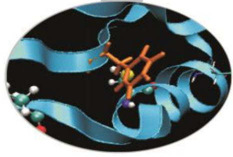
che prende come argomento l'indirizzo del dato di cui si vuol fare il casting.

Esempio:

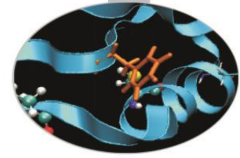
```
ifstream inpFile("input_data.dat",ios::in);
int numero;
inpFile.read(reinterpret_cast<char*>(&numero), sizeof(int));
```

- L'uso delle funzioni `write` e `read` è ciò che ci consente di stabilire se siamo di fronte ad un file ad accesso diretto.

File ad accesso diretto



- Oltre alle funzioni read e write la classe fstream mette a disposizioni altre funzioni:
- get()
- get(char &)
- get(char *buffer, int n, [char delimitatore])
- getline(char *buffer, int n [char delimitatore])
- put(char);



File ad accesso diretto

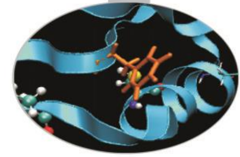
- Le funzioni di ingresso/uscita non formattate si aspettano solo array di caratteri
- Array di 3 interi, ciascun elemento occupa 4 byte

Sequenza di byte

- Un byte può essere rappresentato da un char → qualsiasi sequenza di byte può essere rappresentata da un array di char

```
reinterpret_cast<char *>(<indirizzo_oggetto>)
```

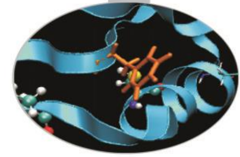
Reinterpreta l'indirizzo dell'oggetto come l'indirizzo a cui inizia una sequenza di byte.



File ad accesso diretto

Esempio: lettura e scrittura di oggetti della classe MyClass su file ad accesso di diretto.

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;
class MyClass{
   friend ostream& operator<<(ostream&, const MyClass&);
   friend istream& operator>>(istream&, MyClass&);
   private:
   int a;
   char c;
   string s;
   public:
   MyClass(){};
   MyClass(int a, char c, string s){
     this->a=a; this->c=c; this->s=s;}
};
ostream& operator<<(ostream& out, const MyClass& obj){
   out << obj.a << " " << obj.c << " " << obj.s << endl;
   return out;
}
```



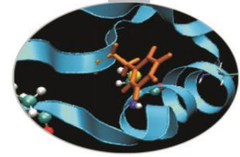
Esempio

```
istream& operator>>(istream& in, MyClass& obj){
   in >> obj.a >> obj.c >> obj.s;
   return in;
}
int main(){
   MyClass A(1,'a',"uno");
   MyClass B(2,'b',"due");
   MyClass C(3,'c',"tre");
   MyClass D(4,'d',"quattro");
   fstream data("data.dat",ios::in|ios::out);
   data.write(reinterpret_cast<const char*>(&A),sizeof(MyClass));
   data.write(reinterpret_cast<const char*>(&D),sizeof(MyClass));
   data.write(reinterpret_cast<const char*>(&C),sizeof(MyClass));
   data.seekg(0);
   MyClass obj;
   cout << "Content of data.dat" << endl;
   for(int i=0; i<3; i++){
       data.read(reinterpret_cast<char*>(&obj),sizeof(MyClass));
       cout << obj;
   }
   data.seekg(0)
```

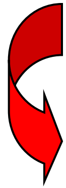
riposizionamento all'inizio del file

scrittura sul file non formattato data.data

lettura dal file non formattato data.data



Esempio



```
data.seekp(1*sizeof(MyClass));
  data.write(reinterpret_cast<const char*>(&B),sizeof(MyClass));
```

riscrittura del secondo record

```
  data.seekg(0);
  cout << "Content of data.dat:" << endl;
  for(int i=0; i<3; i++){
      data.read(reinterpret_cast<char*>(&obj),sizeof(MyClass));
      cout << obj;
  }

return 0;
}
```

Output:

```
Content of data.dat
1 a uno
4 d quattro
3 c tre
Content of data.dat:
1 a uno
2 b due
3 c tre
 data.dat:
```