

# C++

Introduzione, stream e STL

C++: STREAM

# STREAM

- ▶ Prendiamo in mano di nuovo il primo programma...

```
▶ #include <iostream>
using namespace std;
int main() {
    cout << "Ehi!" << endl;
    return 0;
}
```

- ▶ Stream: astrazione di una periferica che può inviare e ricevere dati. Può essere vista come una sorgente o destinazione di sequenze di caratteri di lunghezza indefinita
- ▶ Generalmente associati ad una periferica fisica (come un file su disco, una tastiera, la console, ...) così che i caratteri in in/out su questa astrazione vengano rediretti in in/out sulla periferica fisica

# STREAM

- ▶ Input dall'utente è sequenza di lunghezza indefinita di `char`
- ▶ Output all'utente è sequenza di lunghezza indefinita di `char`
- ▶ Conti e operazioni vanno eseguiti su tipi numerici

- ▶ 

```
double number = 42.0;
string number = "42.0"
```
- ▶ 

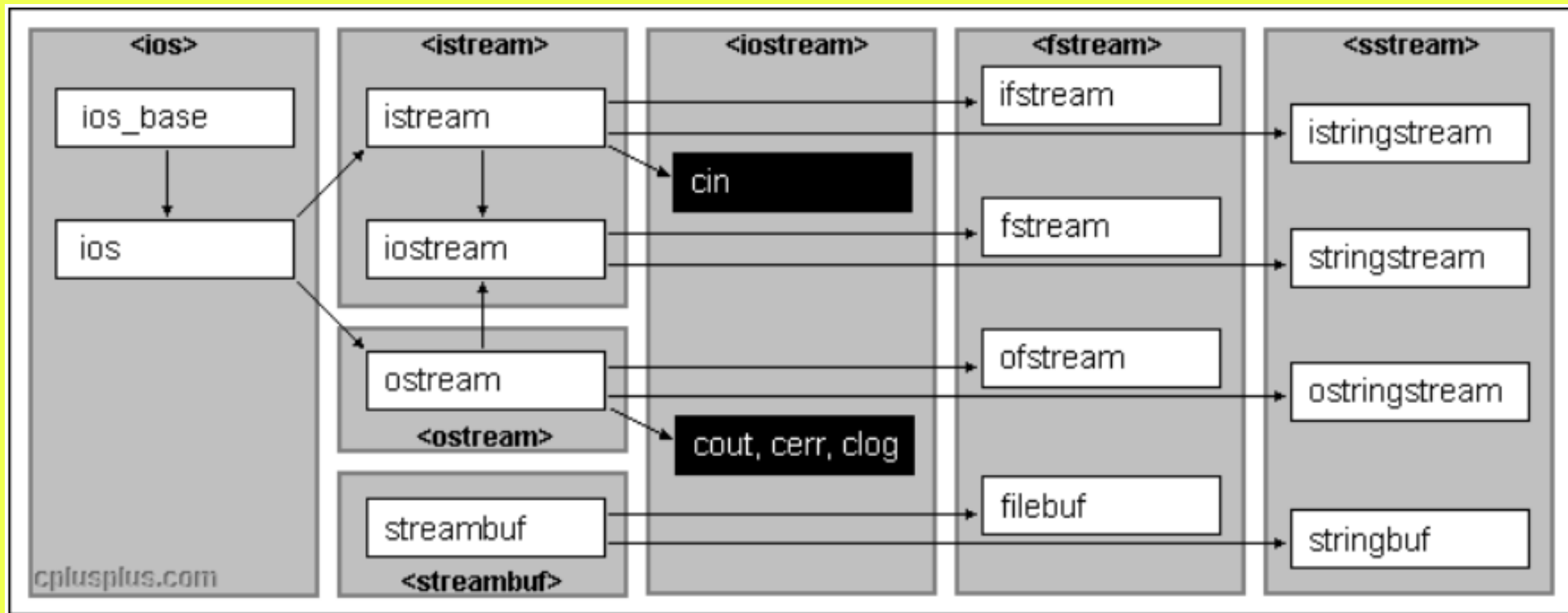
```
void printDouble( string s ) {
    cout << s * 2 << endl;
} // Questo non compila!
```
- ▶ 

```
void appendQuattro ( int n ){
    cout << n + "4" << endl;
} // Questo compila
```
- ▶ 

```
void appendQuattro ( int n ){
    cout << n + '4' << endl;
} // Anche questo compila
```

# STREAM

- In questo caso `std::cout` è uno stream che può solo ricevere dati. La sua funzione è di prendere dei dati in ingresso e visualizzarli sulla console. Stream come questi sono detti **output stream** (ostream).



# STREAM

- ▶ Idea più generale: conversione tra dati e la loro rappresentazione in stringa (o binaria!)
- ▶ Es.:  
cout è stream che accetta stringhe  
invio dati allo stream tramite l'operatore "string insertion" <<  
questo operatore converte dati in string e lo invia a cout

# STREAM

- ▶ E **input stream** (istream)?
- ▶ Questa roba come la vedete?
- ▶ 

```
int x;  
cin >> x;
```

# STREAM

- ▶ E **input stream** (`istream`)?

- ▶ Questa roba come la vedete?

- ▶ 

```
int x;  
cin >> x;
```

- ▶ Gli stream da cui è possibile estrarre dei dati sono detti input stream (`istream`)  
Per prelevare dati da esso si utilizza l'operatore string extraction `>>`  
`>>` preleva un dato dallo stream e lo converte nel tipo appropriato
- ▶ In questo caso, `std::cin` è l'analogo di `cout`, ma funziona al contrario, prelevando un dato dalla console

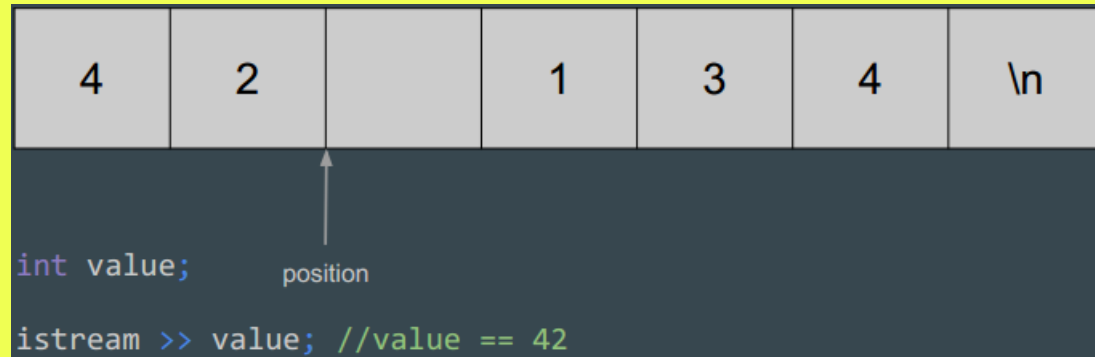


# STREAM

- ▶ Possiamo usare `ostream` non solo per inviare dati alla console, ma per molto di più, per esempio salvare dati su un file.
- ▶ `ofstream` fanno esattamente questo
- ▶ In maniera speculare: `ifstream` serve a importare dati da un file.
- ▶ Sono oggetti: possiedono variabili che ne denotano lo stato, numero totale dei caratteri letti, flag di formattazione, ecc...

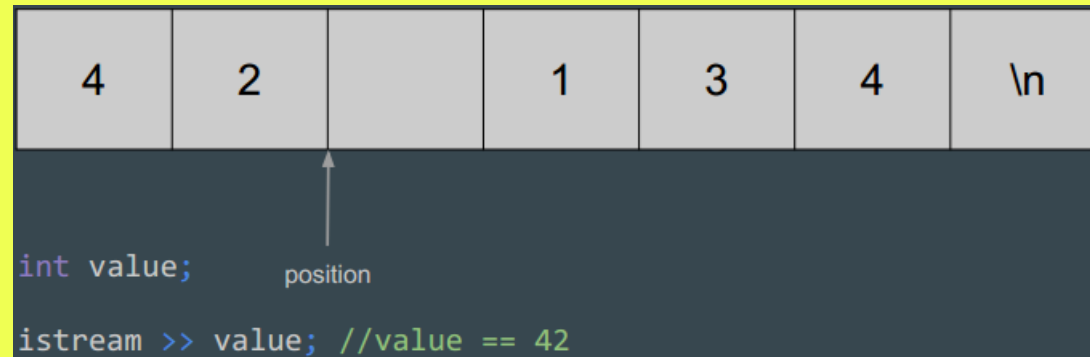
# STREAM

- ▶ Es. Lettura di un int da file

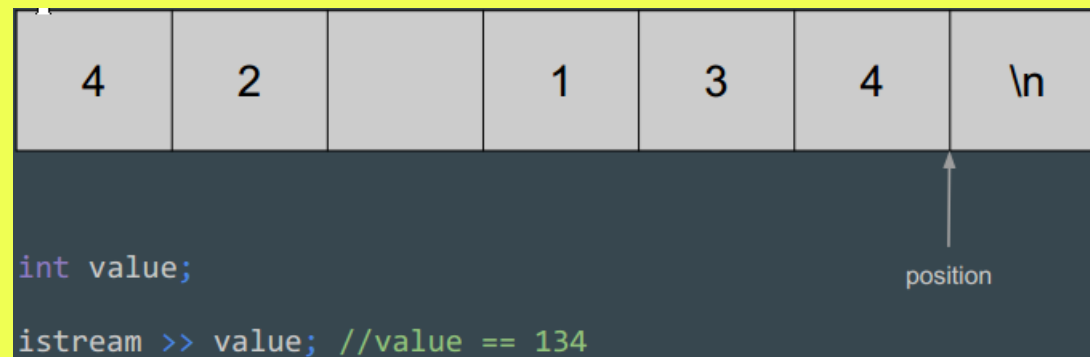


# STREAM

- ▶ Es. Lettura di un int da file

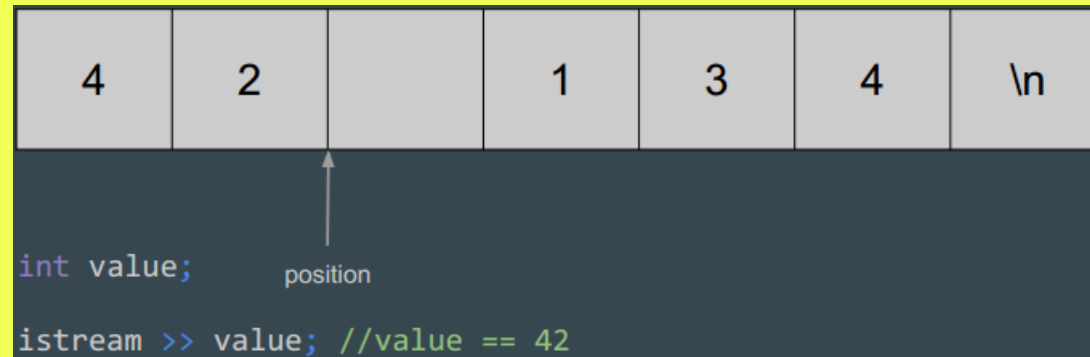


- ▶ Una lettura successiva ignora qualsiasi spazio e legge l'intero successivo

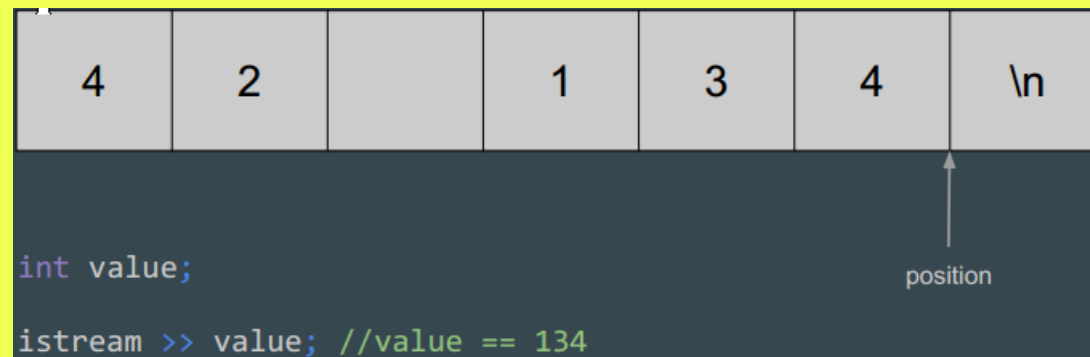


# STREAM

- ▶ Es. Lettura di un int da file



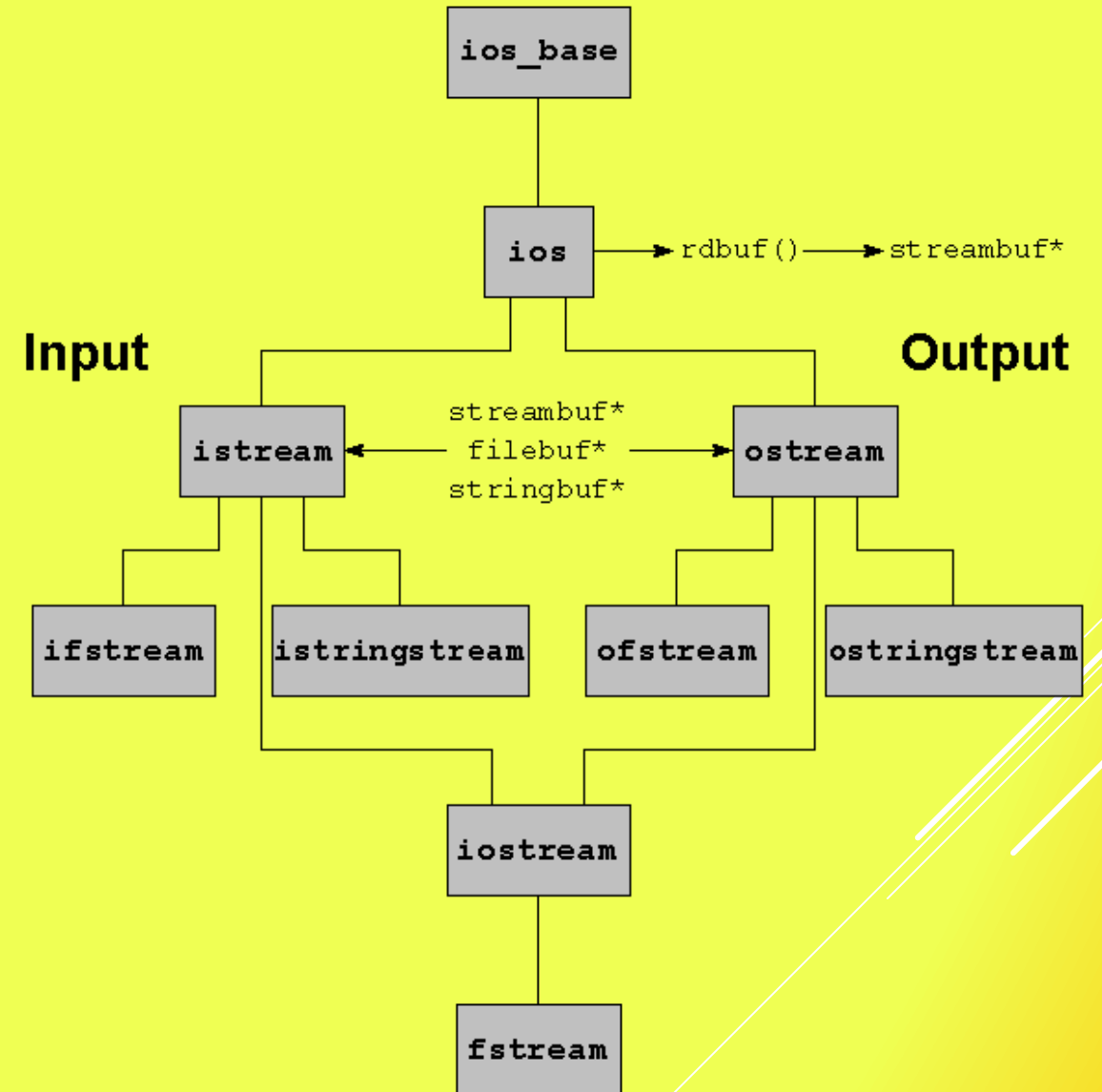
- ▶ Una lettura successiva ignora qualsiasi spazio e legge l'intero successivo



Se non ci sono più caratteri da leggere, fail bit viene impostato a true

# STREAM

- ▶ Gli oggetti stream appena visti hanno un comportamento simile a prescindere dal tipo di i/o su cui di appoggiano.
- ▶ Tutti gli stream sono oggetti definiti come sottoclassi di `ios_base`: essa definisce il "core" di tutte le operazioni di I/O
- ▶ `ios`, è sottoclasse di `ios_base`; implementa la comunicazione con il buffer usato nello stream (streambuffer: oggetto responsabile del I/O con la device interessata)
- ▶ Nota: `ios_base` non ha costruttori pubblici.



# STREAM

- ▶ Domanda:  
cosa succede se input stream legge un tipo di dato non corretto?
- ▶ Es.
- ▶ 

```
int x = 100;  
cin >> x;
```
- ▶ E in console scrivo:
- ▶ Testo!

# STREAM

- ▶ Domanda:  
cosa succede se input stream legge un tipo di dato non corretto?
- ▶ Es.
- ▶ 

```
int x = 100;  
cin >> x;
```
- ▶ E in console scrivo:
- ▶ Testo!
- ▶ Risposta:  
<http://www.cplusplus.com/reference/istream/istream/operator%3E%3E/>

# STREAM

- ▶ Ultimo tipo di stream che vediamo è lo stringstream
  - ▶ Contrariamente a istream e ostream, stringstream non servono ad inviare dati
  - ▶ Usati come **buffer** di stringhe oppure
  - ▶ Usati per convertire dati da un formato ad un altro (vedi esercizio)
- 
- ▶ Internamente, tutti gli stream memorizzano la sequenza di dati all'interno di un buffer
  - ▶ Istream lo utilizza per conservare dati che non abbiamo ancora richiesto
  - ▶ Ostream lo utilizza per conservare dati che non ha ancora emesso
  - ▶ (usare funzione flush per forzare svuotamento del buffer!)



# STREAM

- ▶ Perché possiamo concatenare più stringhe negli ostream (x es.)?
- ▶ Nota: gli operatori sono visti da C++ come delle vere e proprie funzioni e, come tali, hanno un return value.
- ▶ Questo vale anche per l'operatore << di uno ostream.

# STREAM

- ▶ Perché possiamo concatenare più stringhe negli ostream (x es.)?
- ▶ Nota: gli operatori sono visti da C++ come delle vere e proprie funzioni e, come tali, hanno un return value.
- ▶ Questo vale anche per l'operatore << di uno ostream.
- ▶ Allora come funziona la concatenazione negli ostream?
- ▶ <http://www.cplusplus.com/reference/ostream/ostream/operator%3C%3C/>
- ▶ `cout << "Eccellente, Smithers" << endl;`

# STREAM

- ▶ Perché possiamo concatenare più stringhe negli ostream (x es.)?
- ▶ Nota: gli operatori sono visti da C++ come delle vere e proprie funzioni e, come tali, hanno un return value.
- ▶ Questo vale anche per l'operatore << di uno ostream.
- ▶ Allora come funziona la concatenazione negli ostream?
- ▶ <http://www.cplusplus.com/reference/ostream/ostream/operator%3C%3C/>
- ▶ `cout << "Eccellente, Smithers" << endl;`
- ▶ `(cout << "Eccellente, Smithers") << endl;`

# STREAM

- ▶ Perché possiamo concatenare più stringhe negli ostream (x es.)?
- ▶ Nota: gli operatori sono visti da C++ come delle vere e proprie funzioni e, come tali, hanno un return value.
- ▶ Questo vale anche per l'operatore << di uno ostream.
- ▶ Allora come funziona la concatenazione negli ostream?
- ▶ <http://www.cplusplus.com/reference/ostream/ostream/operator%3C%3C/>
- ▶ `cout << "Eccellente, Smithers" << endl;`
- ▶ `(cout << "Eccellente, Smithers") << endl;`
- ▶ `cout << endl;`

# STREAM

- ▶ Non abbiamo ancora parlato di `std::endl`
  - ▶ Non è un semplice `\n` della `printf`
  - ▶ `std::endl` è uno stream manipulator.
- 
- ▶ Stream manipulator sono **funzioni** che modificano il comportamento degli stream
  - ▶ `std::endl` emette un `\n` e richiama il flush dello stream
  - ▶ <http://en.cppreference.com/w/cpp/io/manip>

# STREAM

- ▶ Non abbiamo ancora parlato di `std::endl`
  - ▶ Non è un semplice `\n` della `printf`
  - ▶ `std::endl` è uno stream manipulator.
- 
- ▶ Stream manipulator sono **funzioni** che modificano il comportamento degli stream
  - ▶ `std::endl` emette un `\n` e richiama il flush dello stream
- 
- ▶ <http://en.cppreference.com/w/cpp/io/manip>
  - ▶ <http://www.cplusplus.com/reference/library/manipulators/>
- 
- ▶ `std::cout << "Bella zio!" << std::endl;`

# STREAM

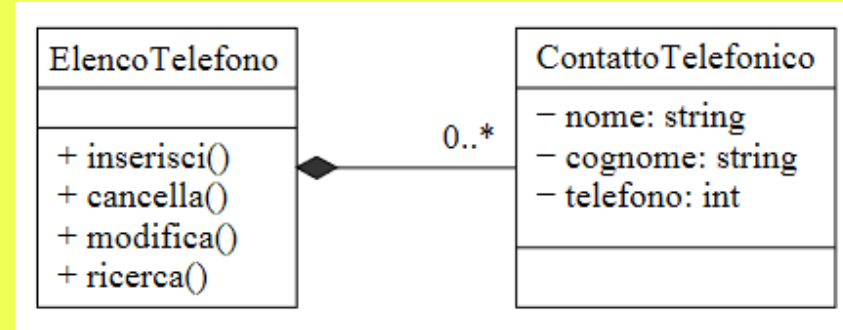
- ▶ Non abbiamo ancora parlato di `std::endl`
  - ▶ Non è un semplice `\n` della `printf`
  - ▶ `std::endl` è uno stream manipulator.
- 
- ▶ Stream manipulator sono **funzioni** che modificano il comportamento degli stream
  - ▶ `std::endl` emette un `\n` e richiama il flush dello stream
- 
- ▶ <http://en.cppreference.com/w/cpp/io/manip>
  - ▶ <http://www.cplusplus.com/reference/library/manipulators/>
- 
- ▶ `std::cout << "Bella zio!" << std::endl;`
  - ▶ `std::endl( std::cout << "Bella zio!" );`

# C++: STL: CONTAINER

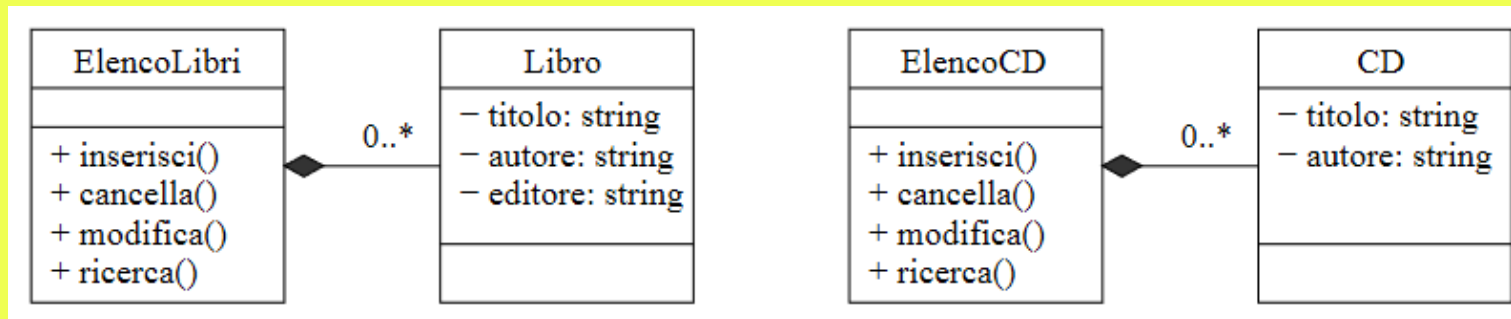


# STL: CONTAINER

- ▶ Partiamo da un esempio: voglio realizzare un sw per la gestione dei miei contatti personali



- ▶ Realizzo classe **ElencoTelefonico** che offre servizi per gestione dei record.
- ▶ Suppongo che **ElencoTelefonico** faccia uso di strutture dati complesse e di algoritmi efficienti; l'ideale sarebbe riutilizzare in futuro il codice per gestire altro



# STL: CONTAINER

- ▶ Per questo mi piacerebbe riutilizzare tutto il codice scritto per `ElencoTelefonico`
- ▶ Obiettivo: progettare un gestore simile a `ElencoTelefonico`, ma che non sia legato ad un tipo specifico, cioè progettare una classe parametrica `Elenco<>`, lasciando indefinita la classe che specifica gli oggetti interni e specificandola come parametro da istanziare durante la compilazione, a seconda delle necessità.
- ▶ `Elenco<ContattoTelefonico>`, `Elenco<Libro>`, `Elenco<CD>`
- ▶ `Elenco<>` ricopre la funzione di "contenitore", parametrico rispetto agli oggetti contenuti

In C++ le classi parametriche sono definite **template**.

- ▶ Termine chiave: programmazione generica (generic programming)

# STL: CONTAINER

- ▶ C++ fornisce già gli strumenti per la creazione di contenitori simili ad Elenco<>
- ▶ Queste strutture sono contenute nella STL, Standard Template Library che fornisce diversi tipi di contenitore.
- ▶ Contenitori in STL di vario tipo perché modellizzano diverse categorie di dato astratto, x es., vettori, pile (stack), liste...
- ▶ A seconda del problema da affrontare, alcuni container sono più vantaggiosi di altri per tempo di accesso, memoria occupata, ecc...

# STL: CONTAINER

- Quali contenitori abbiamo a disposizione? (<http://www.cplusplus.com/reference/stl/>)

## Container class templates

### Sequence containers:

<b>array</b> <small>C++11</small>	Array class (class template )
<b>vector</b>	Vector (class template )
<b>deque</b>	Double ended queue (class template )
<b>forward_list</b> <small>C++11</small>	Forward list (class template )
<b>list</b>	List (class template )

### Container adaptors:

<b>stack</b>	LIFO stack (class template )
<b>queue</b>	FIFO queue (class template )
<b>priority_queue</b>	Priority queue (class template )

### Associative containers:

<b>set</b>	Set (class template )
<b>multiset</b>	Multiple-key set (class template )
<b>map</b>	Map (class template )
<b>multimap</b>	Multiple-key map (class template )

### Unordered associative containers:

<b>unordered_set</b> <small>C++11</small>	Unordered Set (class template )
<b>unordered_multiset</b> <small>C++11</small>	Unordered Multiset (class template )
<b>unordered_map</b> <small>C++11</small>	Unordered Map (class template )
<b>unordered_multimap</b> <small>C++11</small>	Unordered Multimap (class template )

# STL: CONTAINER

- ▶ Tutti i contenitori hanno funzioni comuni; ogni container può avere delle funzioni specifiche.
- ▶ Es.
- ▶ `empty()`: restituisce true se il container non contiene elementi  
`size()`: restituisce il numero di elementi presenti correntemente nel container  
`max_size()`: restituisce il numero massimo di elementi per un container
- ▶ [solo per contenitori sequenziali ed associativi]  
`begin()`: restituisce un iteratore che punta al primo elemento del container  
`end()`: restituisce un iteratore che punta all'ultimo elemento del container  
`erase()`: elimina uno o più elementi del container

# STL: CONTAINER

- ▶ In C e C++ si accede agli elementi di un array tramite puntatore
- ▶ Per accedere agli elementi di un container si usano gli **iteratori**.
- ▶ Classe degli iteratori progettata per essere utilizzati in modo generico su qualsiasi container.  
Per ora pensiamoli come dei puntatori "intelligenti".
- ▶ Infine, STL contiene degli **algoritmi** per operare sugli oggetti contenuti dentro un container.  
Anche qui, algoritmi sono implementati in modo indipendente dal container.

# STL: CONTAINER

```
▶ #include <iostream>
  #include <list>
  #include <algorithm>

▶ using namespace std;

▶ int main() {
    list<int> contenitore;
    int x;

▶     do {
        cin >> x;
        contenitore.push_back( x );
    } while (x != 0);

▶     contenitore.sort();

▶     for( list<int>::iterator i = contenitore.begin(); i != contenitore.end(); i++ )

▶         cout << *i << endl;

▶     return 0;

▶ }
```

# STL: CONTAINER

- ▶ Un **contenitore sequenziale** è un container a cui si può accedere sequenzialmente per prelevarne gli elementi
- ▶ Es: vector, list, deque, array, forward\_list  
(occhio: std::array non è il "solito" array di C!)
- ▶ **std::vector** rappresenta array di dimensione non fissata: possono crescere di dimensione qualora si aggiungano elementi oltre la capacità.
- ▶ Come array, hanno operatore []
- ▶ Efficienti nell'accesso a i-esimo elemento, relativamente efficienti nell'aggiunta di un nuovo elemento alla fine
- ▶ Ogni altra operazione di inserimento (inizio o all'interno del vector) è inefficiente.



# STL: CONTAINER

- ▶ Ok, ma quanto è inefficiente?
- ▶ Definisco `std::vector<int> v;`

```
for (int i = 0; i < N; i++)  
    v.push_back(i);
```


```
for (int i = 0; i < N; i++)  
    v.insert(v.begin(), i);
```

# STL: CONTAINER

- ▶ Ok, ma quanto è inefficiente?
- ▶ Definisco `std::vector<int> v;`

```
for (int i = 0; i < N; i++)  
    v.push_back(i);
```

```
for (int i = 0; i < N; i++)  
    v.insert(v.begin(), i);
```

	push_front	push_back
N = 1000	0.01	0
N = 10000	0.89	0.01
N = 100000	117.98	0.04
N = 1000000	Hours	0.31
N = 10000000	Months	3.16

# STL: CONTAINER

- ▶ **std::list** sono liste double-linked e rappresentano la soluzione al problema dell'inserimento, eliminazione o spostamento, poiché queste operazioni vengono compiute in tempo lineare.
- ▶ Al contrario degli array, NON hanno operatore []
- ▶ Es. Per accedere a sesto elemento della lista bisogna scorrerla a partire dal primo o ultimo elemento!
- ▶ Come fare?
- ▶ Nota: guardare su [cppreference](#) gli altri container sequenziali

# STL: CONTAINER

- ▶ I contenitori sequenziali memorizzano dati in modo che sia rispettato un ordine.
- ▶ Al contrario, i **container associativi** non hanno idea di cosa sia l'ordine: progettati per fornire accesso diretto in memorizzazione e recupero dati tramite **chiavi di ricerca** (sistema chiave – valore)
- ▶ Attualmente, STL supporta otto tipi di contenitori associativi:

<b>Associative containers:</b>	
<b>set</b>	Set (class template )
<b>multiset</b>	Multiple-key set (class template )
<b>map</b>	Map (class template )
<b>multimap</b>	Multiple-key map (class template )
<b>Unordered associative containers:</b>	
<b>unordered_set</b> <small>C++11</small>	Unordered Set (class template )
<b>unordered_multiset</b> <small>C++11</small>	Unordered Multiset (class template )
<b>unordered_map</b> <small>C++11</small>	Unordered Map (class template )
<b>unordered_multimap</b> <small>C++11</small>	Unordered Multimap (class template )

- ▶ Nei primi 4 le chiavi sono mantenute ordinate e attraversamento container avviene secondo l'ordinamento previsto.

# STL: CONTAINER

- ▶ **std::set** è contenitore che modella un insieme.
- ▶ Fornisce operazioni per manipolare valori le cui chiavi sono i valori stessi.
- ▶ Non sono ammessi valori duplicati (ma in **std::multiset** sì)
- ▶ **std::map** e **std::multimap**, gli oggetti memorizzati sono oggetti pair, cioè coppie chiave-valore
- ▶ Differenza tra map e multimap è che nel secondo, ad una chiave posso associare più valori
- ▶ Operatore [ ] ?

# STL: CONTAINER

- ▶ Cosa posso fare con i `std::set`? Es: definisco un set di stringhe:
- ▶ `std::set<std::string> sds;`
- ▶ Posso aggiungere un elemento a `sds` con `sds.insert( "Nuova stringa" );`
- ▶ Come per tutti i container della STL, `std::set` supporta gli iteratori, quindi `sds.begin()` e `sds.end()` restituiscono puntatori a primo e ultimo elemento del set
- ▶ Di default, elementi di set sono memorizzati in ordine ascendente e l'insieme è mantenuto ordinato (la chiamata ad `insert()` piazza il nuovo elemento nel posto giusto).
- ▶ Domande:
  - come avviene confronto? (chiaro per valori numerici, ma per stringhe?)
  - come faccio a cambiare ordine?

# STL: CONTAINER

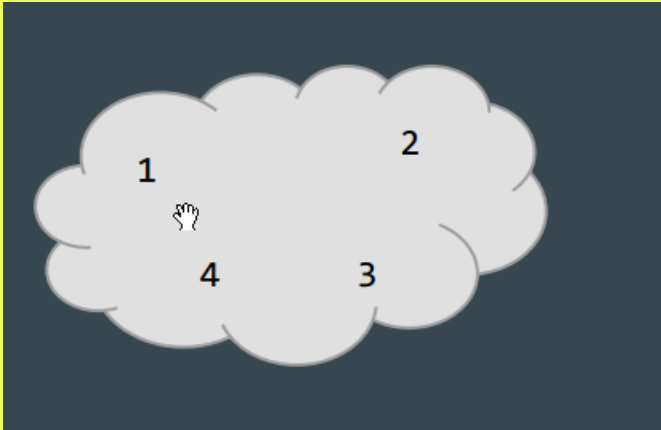
- ▶ Punto di forza dei contenitori associativi è ricerca di valori:
- ▶ `sds.find( "stringa cercata" )` restituisce un iterator che punta all'elemento cercato oppure `sds.end()` qualora l'elemento non fosse presente nell'insieme
- ▶ Struttura di memorizzazione di container associativi è *red-black tree*: in questa struttura, il tempo di ricerca medio è proporzionale a  $\log N$ , con  $N = \text{\#elementi in set}$ .
- ▶ Confronta con `vector::find()` che ha tempo di ricerca lineare
- ▶ Svantaggio: struttura "grossa" e insertion richiede tree rebalancing.
- ▶ Stesse considerazioni valgono per `std::map` e `std::multimap`
- ▶ Nota: elementi si `std::set` sono immutable!

# C++: STL: ITERATOR



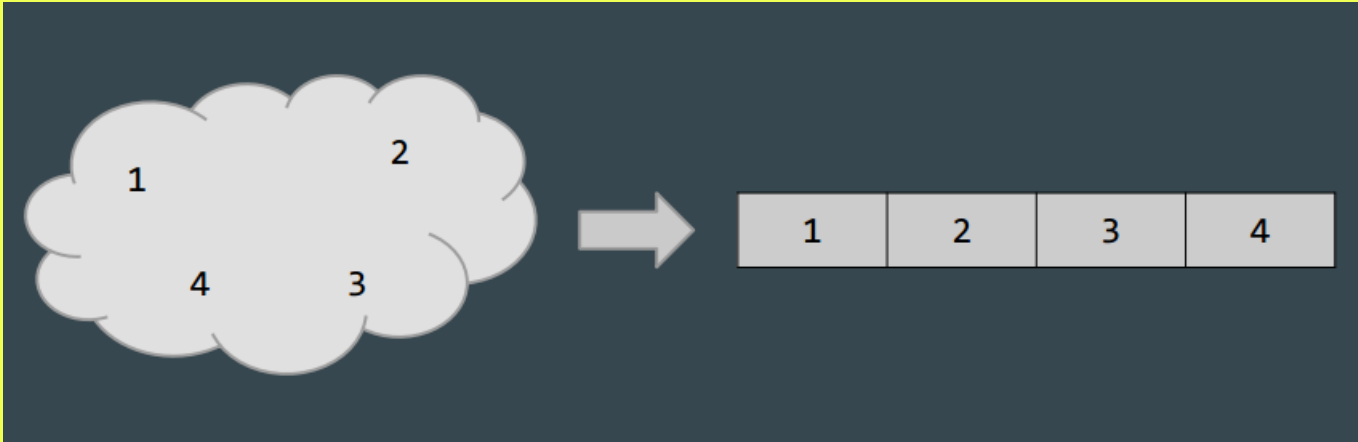
# STL: ITERATOR

- ▶ Oggetti che permettono di iterare su un container a prescindere che questo sia ordinato o meno
- ▶ Immaginiamo di avere in `std::set` di interi: `std::set<int> mySet;`



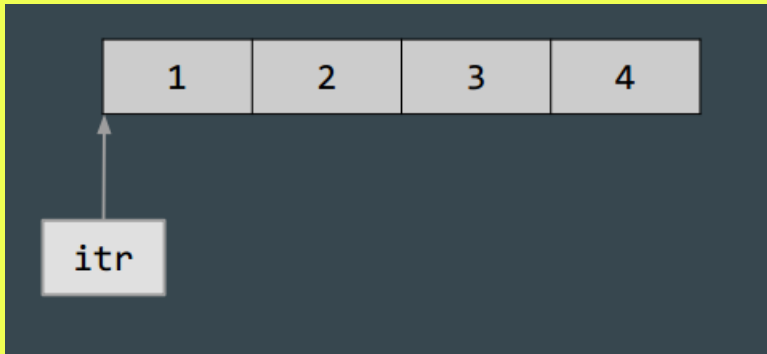
# STL: ITERATOR

- ▶ Oggetti che permettono di iterare su un container a prescindere che questo sia ordinato o meno
- ▶ Immaginiamo di avere in `std::set` di interi: `std::set<int> mySet;`
- ▶ Iterator permettono di vederlo in forma lineare



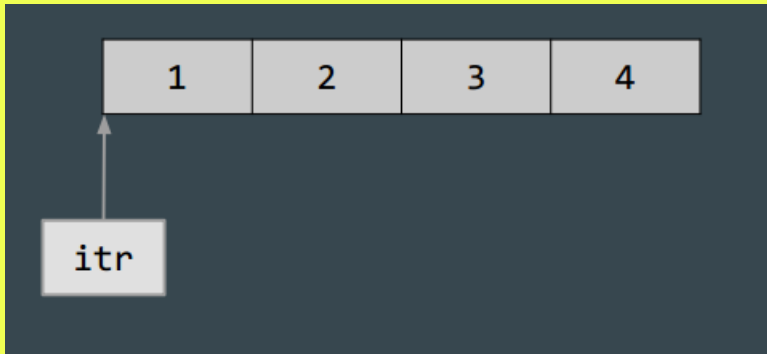
# STL: ITERATOR

- ▶ Oggetti che permettono di iterare su un container a prescindere che questo sia ordinato o meno
- ▶ Immaginiamo di avere in `std::set` di interi: `std::set<int> mySet;`
- ▶ Posso costruire un iteratore che punta al primo elemento dell'insieme  
`std::set<int>::iterator itr = mySet.begin();`



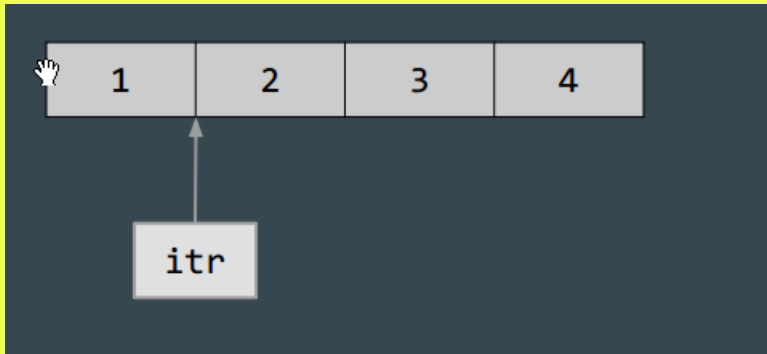
# STL: ITERATOR

- ▶ Oggetti che permettono di iterare su un container a prescindere che questo sia ordinato o meno
- ▶ Immaginiamo di avere in `std::set` di interi `std::set<int> mySet;`
- ▶ Posso costruire un iteratore che punta al primo elemento dell'insieme  
`std::set<int>::iterator itr = mySet.begin();`
- ▶ Ed ottenere il valore di un iteratore tramite l'operatore di dereferenziazione `*`  
`std::cout << *itr << std::endl;`      `// Stampa 1`



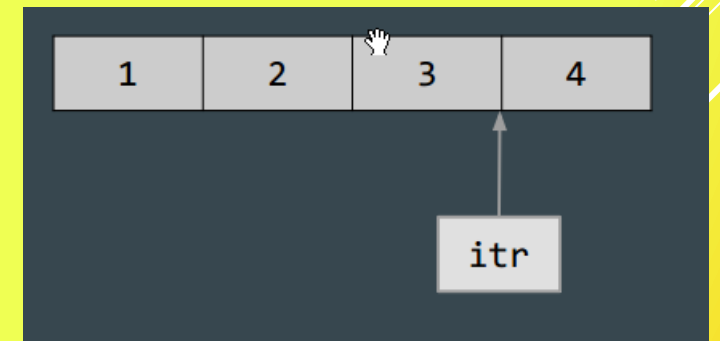
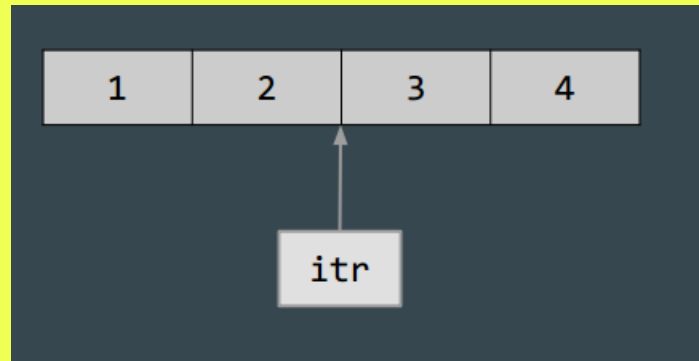
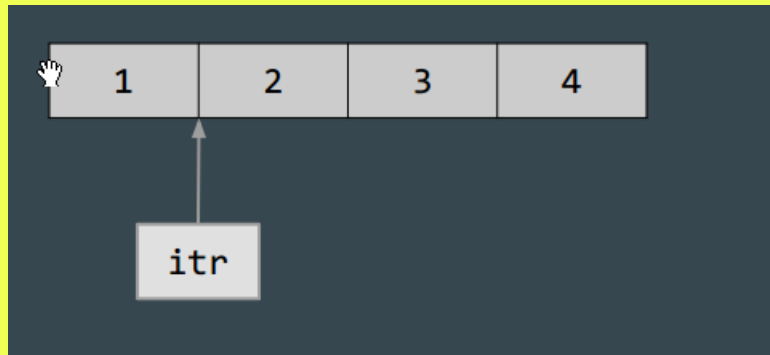
# STL: ITERATOR

- ▶ Oggetti che permettono di iterare su un container a prescindere che questo sia ordinato o meno
- ▶ Immaginiamo di avere in `std::set` di interi `std::set<int> mySet;`
- ▶ Per avanzare all'elemento successivo, uso l'operatore di incremento unario `++itr;`



# STL: ITERATOR

- ▶ Oggetti che permettono di iterare su un container a prescindere che questo sia ordinato o meno
- ▶ Immaginiamo di avere in `std::set` di interi `std::set<int> mySet;`
- ▶ Per avanzare all'elemento successivo, uso l'operatore di incremento unario `++itr;`
- ▶ E continuare ad incrementare quanto voglio



# STL: ITERATOR

- ▶ Quattro operazioni fondamentali:
- ▶ - **Creazione** di un iterator
- ▶ - **Dereferenziazione** di un iteratore per ottenere il valore (o oggetto) a cui sta attualmente puntando
- ▶ - **Avanzamento** di una o più posizioni
- ▶ - **Comparazione** due iteratori, in particolare con quello restituito da `.end()`

# STL: ITERATOR

- ▶ Iterator possono scorrere un determinato range di un container:
- ▶ Cosa fa questo codice?

```
▶ std::set<int>::iterator i = mySet.lower_bound(12);  
  std::set<int>::iterator j = mySet.upper_bound(42);  
  while (i != j) {  
      std::cout << *i << std::endl;  
      i++;  
  }
```



# STL: ITERATOR

- ▶ Iterator possono scorrere un determinato range di un container:
- ▶ Cosa fa questo codice?

```
▶ std::set<int>::iterator i = mySet.lower_bound(12);  
  std::set<int>::iterator j = mySet.upper_bound(42);  
  while (i != j) {  
      std::cout << *i << std::endl;  
      i++;  
  }
```

- ▶ Nota:

	$[a, b]$	$[a, b)$	$(a, b]$	$(a, b)$
begin	lower_bound(a)	lower_bound(a)	upper_bound(a)	upper_bound(a)
end	upper_bound(b)	lower_bound(b)	upper_bound(b)	lower_bound(b)