1 Standard Template Library (STL)

La Standard Template Library (STL) è il cuore della libreria standard del C++, permette ai programmatori l'utilizzo di algoritmi e strutture dati allo stato dell'arte, senza preoccuparsi della loro implementazione. Tutti i componenti della STL sono generici, possono quindi essere utilizzati come elementi di tipo arbitrario.
La STL porta alcuni vantaggi, tra cui:

- **Disponibilità di componenti generali**: Non offre soluzioni per un solo problema specifico, ma componenti generici che vanno bene per tutto.
- **Alto livello di astrazione**: Concentrarsi sul "cosa" vuoi fare, ignorando il "come" il computer lo gestisce.
- **Portabilità del codice**: Poiché la STL è uno standard internazionale del C++, il codice che scrivi usando queste librerie funzionerà ovunque ci sia un compilatore C++.
- **Non dover re-implementare ogni cosa from scratch**: Significa applicare il principio del riutilizzo del codice. Invece di scrivere manualmente le funzionalità di base, il programmatore utilizza i componenti già pronti della libreria.

Lo stato dell' arte

Con l'espressione "Stato dell'arte" si intende l'insieme degli algoritmi, delle tecnologie e delle metodologie che, ad oggi, offrono le prestazioni migliori in termini di efficienza, sicurezza e affidabilità.

Inoltre mette a disposizione:

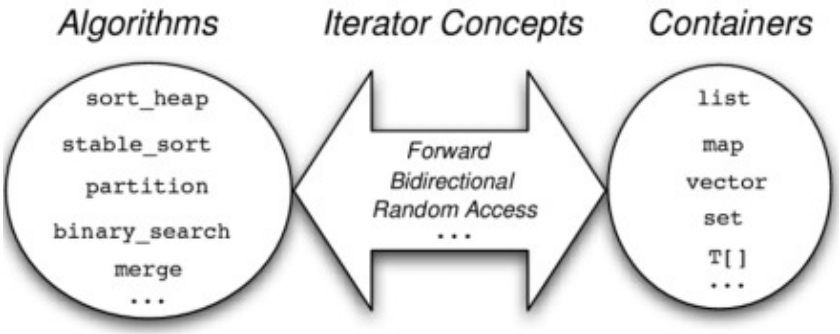
- **Varie strutture dati**: array dinamici, liste, alberi binari, ecc.
- **Vari algoritmi**: per ricerca, ordinamento, ecc.
- **Il tipo stringa**.
- **Classi per la gestione dell'I/O**.
- **Classi numeriche**.

La STL è basata sulla cooperazione di diversi componenti:

- **Contenitori (Container)**: sono usati per la gestione di oggetti di un dato tipo. Possono essere implementati come array, liste, etc.
- **Iteratori (Iterator)**: sono usati per visitare gli elementi di collezioni di oggetti (container o sottoinsiemi di essi).
- **Algoritmi (Algorithm)**: sono usati per processare elementi di collezioni (ricerca, ordinamento, modifica, etc.).

1.1 Container

La progettazione della STL è basata sulla separazione tra dati e operazioni. I dati sono gestiti dalle **classi container**, mentre le operazioni sono definite da **algoritmi**. Gli iteratori sono il "collante" tra container e algoritmi.



STL fornisce diversi tipi di container per venire incontro a diverse necessità da parte del programmatore. Ognuno di essi ha vantaggi e svantaggi, la scelta deve essere effettuata a seconda delle operazioni che si vogliono effettuare. Esistono due tipi di container:

- **Sequenziali (Sequence container)**: vector, deque e list.
- **Associativi (Associative container)**: set, multiset, map e multimap.

1.2 Sequence container (Contenitori Sequenziali)

I Sequence container, contengono una collezione ordinata di elementi di un unico tipo. I principali sono:

- **vector**: archivia gli elementi in una disposizione lineare e consente l'accesso casuale a qualsiasi elemento.
- **list**: viene archiviata come elenco collegato bidirezionale di elementi in disposizione lineare. Consente inserimenti ed eliminazioni efficienti in qualsiasi posizione all'interno della sequenza.
- **deque**: si comporta come un vector ma è preferibile nelle implementazioni delle code.

1.2.1 Vector

Il vector è considerato il container di "default" del C++. Gli elementi al suo interno sono gestiti come in un **array dinamico**, ovvero un array la cui dimensione può cambiare durante l'esecuzione del programma. Le sue caratteristiche principali sono:

+ **Accesso casuale (Random Access)**: È possibile accedere direttamente a qualsiasi elemento tramite il suo indice, senza dover scorrere gli elementi precedenti.
+ **Inserimento/Rimozione in coda**: L'aggiunta o la rimozione di elementi alla fine del vettore è un'operazione molto veloce.
+ **Inserimento nel mezzo**: L'inserimento o la rimozione in posizioni diverse dalla coda è meno efficiente, in quanto richiede lo spostamento (shift) di tutti gli elementi successivi per fare spazio a quello nuovo o per riempire il vuoto.

- **Lentezza posizionale**: Potenzialmente lento se le operazioni di inserimento/cancellazione avvengono spesso in testa (inizio) o in posizioni casuali.
- **Costo di copia**: Se il tipo di dato contenuto è complesso, riordinare gli elementi (spostarli) può essere costoso.
- **Invalidazione di puntatori e iteratori**: Ogni operazione che modifica la capacità del vettore (es. un inserimento che fa scattare un ridimensionamento automatico) può far sì che il vettore venga spostato in un nuovo blocco di memoria. Di conseguenza, tutti i puntatori o iteratori che puntavano ai vecchi elementi diventano non validi (dangling pointers) e usarli causa errori nel programma.

Metodi principali:

Modificatori (Modifiers):
- `assign()`: Assegna nuovi valori al vettore, sostituendo completamente quelli vecchi.
- `push_back()`: Aggiunge un nuovo elemento alla fine del vettore.
- `pop_back()`: Rimuove l'ultimo elemento del vettore (riducendo la dimensione di 1).
- `insert()`: Inserisce nuovi elementi prima di una posizione specifica (indicata tramite un iteratore).
- `erase()`: Rimuove elementi da una specifica posizione o da un intervallo.
- `swap()`: Scambia il contenuto di due vettori dello stesso tipo. È un'operazione molto efficiente anche se le dimensioni sono diverse.
- `clear()`: Rimuove **tutti** gli elementi dal vettore.

Iteratori (Iterators):
- `begin()`: Restituisce un iteratore che punta al **primo** elemento del vettore.
- `end()`: Restituisce un iteratore che punta all'elemento teorico **successivo all'ultimo** (past-the-end). Non punta all'ultimo dato valido, ma segna la fine del container.

Gestione della Capacità (Capacity):
- `size()`: Restituisce il numero effettivo di elementi attualmente presenti nel vettore.
- `max_size()`: Restituisce il numero massimo teorico di elementi che il vettore può contenere.
- `capacity()`: Restituisce la dimensione dello spazio di memoria attualmente **allocato**. Indica il numero elementi il vettore può contenere prima di dover chiedere altra memoria al sistema.
- `resize(n)`: Ridimensiona il contenitore affinché contenga esattamente 'n' elementi. Se 'n' è minore dell'attuale size, gli elementi in eccesso vengono distrutti.
- `empty()`: Restituisce `true` se il container è vuoto (ovvero se la size è 0).
- `shrink_to_fit()`: Riduce la capacità (memoria allocata) per farla coincidere con la size (elementi effettivi), liberando la memoria inutilizzata.

Listing 1: Esempio di utilizzo dei metodi di Capacità

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

int main() {
// Dichiarazione di un vettore di interi vuoto.
vector<int> g1;

// Ciclo di riempimento: aggiunge gli interi da 1 a 5.
for (int i = 1; i <= 5; i++)
    // push_back(): Metodo per l'inserimento efficiente in coda.
    g1.push_back(i);

// Accesso casuale O(1): usa l'operatore [] per accedere al terzo
   elemento (valore 3).
cout << "\nReference operator [g] : g1[2] = " << g1[2];

cout << "Output of begin and end: ";
// Iterazione: usa gli iteratori begin() e end() per stampare il
   contenuto (1 2 3 4 5).
for (auto i = g1.begin(); i != g1.end(); ++i)
    cout << *i << " ";

// size(): Numero attuale di elementi (5).
cout << "\nSize : " << g1.size();

// capacity(): Memoria allocata, sara >= size().
cout << "\nCapacity : " << g1.capacity();

// max_size(): Dimensione massima teorica.
cout << "\nMax_Size : " << g1.max_size();

// resize(4): Ridimensiona il vettore a 4 elementi. L'ultimo (5) viene
   eliminato.
g1.resize(4);

cout << "\nSize : " << g1.size(); // Nuova dimensione (4)

// empty(): Controllo se il vettore e' vuoto (restituisce false).
if (g1.empty() == false)
    cout << "\nVector is not empty";
else
    cout << "\nVector is empty";

return 0;
}
```

1.2.2 List

Una lista è implementata come una lista di elementi doppiamente collegata, quindi ogni elemento ha il suo segmento di memoria e puntatori sia al predecessore che al successore. La lista non ha random access per cui per accedere ad un ipotetico decimo elemento, bisogna scorrere i primi nove elementi della catena.

Confronto con Vector

+ **Inserimento e cancellazione in $O(1)$:** Una volta trovata la posizione, l'operazione è rapidissima, in quanto si modificano solo i puntatori (link), senza spostare i dati.
+ **Ideale per oggetti pesanti:** Non essendoci costi di copia o spostamento degli elementi in memoria, è la scelta migliore per memorizzare oggetti di grandi dimensioni.
− **Accesso casuale in $O(N)$:** Per accedere all'elemento N, è necessario scorrere N nodi a partire dalla testa o dalla coda. Non supporta l'operatore `[]`.
− **Traversamento lento (Bad Memory Locality):** I nodi sono sparsi in memoria. La CPU non riesce a sfruttare la cache, rendendo lo scorrimento sequenziale meno efficiente rispetto al `vector`.
− **Maggiore consumo di memoria (Overhead):** Ogni nodo richiede memoria aggiuntiva per memorizzare i due puntatori (prev e next).

Metodi principali:

- `front()`: Ritorna il valore del primo elemento della lista.
- `back()`: Ritorna il valore dell' ultimo elemento della lista.
- `push_front()`: Aggiunge un nuovo elemento all' inizio della lista.
- `push_back()`: Aggiunge un nuovo elemento alla fine della lista.
- `pop_front()`: Rimuove il primo elemento della lista e riduce la grandezza di 1.
- `pop_back()`: Rimuove l'ultimo elemento della lista e riduce la grandezza di 1.
- `insert()`: Aggiunge un nuovo elemento prima di quello nella posizione specificata.
- `size()`: Ritorna il numero di elementi nella lista.
- `begin()`: Ritorna un iteratore che punta al primo elemento della lista.
- `begin()`: Ritorna un iteratore che punta al teorico ultimo elemento della lista.

Listing 2: Esempio di utilizzo dei metodi delle list

```
#include <iostream>
#include <list>

using namespace std;

int main() {
list<int> gqlist{12, 45, 8, 6};

// push_front(33): Inserisce l'elemento 33 all'inizio della lista (testa)
   .
gqlist.push_front(33);
```

```
// push_back(55): Inserisce l'elemento 55 alla fine della lista (coda).
gqlist.push_back(55);

// Output finale: 33 12 45 8 6 55
for (auto i : gqlist) {
    cout << i << " ";
}
return 0;
}
```

1.2.3 Deque

Il termine deque è un'abbreviazione per "double-ended queue". E un array dinamico implementato in maniera tale da rendere veloci inserimento e cancellazione sia in testa che in coda.

Confronto con Vector e List

+ **Accesso casuale in $O(1)$:** Accesso diretto ad ogni elemento tramite indice (come il `vector`), sebbene con un minimo overhead.
+ **Inserimento e cancellazione su entrambe le estremità ($O(1)$):** A differenza del `vector`, supporta `push_front()` e `pop_front()` in tempo costante.
+ **Non invalida puntatori in coda/testa:** Le operazioni di inserimento sui bordi non invalidano i riferimenti agli elementi esistenti.
− **Lentezza negli inserimenti/rimozioni centrali ($O(N)$):** Come il `vector`, le operazioni che richiedono lo spostamento dei dati sono costose.
− **Costo di copia:** Potenziale lentezza se il tipo di dato ha un elevato costo di copia durante il riordino degli elementi.
− **Non completamente cache-friendly:** Non essendo totalmente contiguo in memoria, è meno efficiente del `vector` nel sfruttare la cache della CPU.

Listing 3: Esempio di utilizzo dei metodi della deque

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

// Funzione helper per stampare il contenuto del deque.
void showdq(deque<int> g)
{
deque<int>::iterator it;
// Scorrimento sequenziale tramite iteratore begin() ed end().
for (it = g.begin(); it != g.end(); ++it)
    cout << '\t' << *it;
cout << '\n';
}
```

```
int main()
{
deque<int> gquiz; // Dichiarazione di un deque vuoto.

// Operazioni di Inserimento in testa e coda (O(1)).
gquiz.push_back(10);  // Deque: {..., 10}
gquiz.push_front(20); // Deque: {20, 10, ...}
gquiz.push_back(30);  // Deque: {20, 10, 30}
gquiz.push_front(15); // Deque: {15, 20, 10, 30}
//
cout << "The deque gquiz is : ";
showdq(gquiz);

// size(): Stampa la dimensione attuale (4)
cout << "\ngquiz.size() : " << gquiz.size();
// max_size(): Stampa la dimensione massima teorica.
cout << "\ngquiz.max_size() : " << gquiz.max_size();

// at(2): Accesso casuale all'elemento in indice 2 (valore 10).
cout << "\ngquiz.at(2) : " << gquiz.at(2);
// front(): Elemento in testa (15).
cout << "\ngquiz.front() : " << gquiz.front();
// back(): Elemento in coda (30).
cout << "\ngquiz.back() : " << gquiz.back();

return 0;
}
```

1.2.4 Altri container

Dati i container fondamentali, STL offre anche supporto specifico per le seguenti strutture:
Metodi principali:

- `Stack`: container in cui gli elementi sono gestiti con politica LIFO.
- `Queue`: container in cui gli elementi sono gestiti con politica FIFO.
- `Priority Queue`: la coda con priorità.
- `Associative container (Contenitori Associativi)`: supportano efficientemente interrogazioni circa la presenza e richiese di estrazione di un elemento. I principali sono: set, multiset, map e multimap.

1.2.5 Associative container

Sono collezioni raggruppate nelle quali la posizione corrente di un elemento dipende dal suo valore e da un dato criterio di ordinamento. L'ordine di inserimento non è tenuto in conto. Il criterio di ordinamento ha la forma di una funzione che compara un valore od una

determinata chiave. Sono tipicamente implementati come alberi binari.
Nella STL, sono predefiniti i seguenti associative container:

- `Set`: collezione nella quale gli elementi sono ordinati per valore. Ogni elemento può occorrere una sola volta, non sono ammessi duplicati.
- `Multiset`: come il Set, ma con la differenza che i duplicati sono ammessi.
- `Map`: contiene elementi che sono coppie chiave/valore. Ogni elemento ha quindi una chiave (sulla base della quale è effettuato l'ordinamento) ed un valore. Non sono ammessi duplicati nelle chiavi.
- `Multimap`: come le map, ma sono ammesse chiavi duplicate.

1.3 Iteratori

Un iteratore è un oggetto che può iterare tra gli elementi di un container (o parte di esso). Fornisce un metodo generale per accedere in successione a ciascun elemento di un container. Un iteratore rappresenta una certa posizione all'interno del container, ad esempio, se iter è un iteratore di un container, allora con ++iter fa avanzare l'iteratore avanti in modo che punti al successivo elemento del container, mentre con *iter restituisce il valore dell'elemento puntato.

Operazioni Fondamentali:

- `*`: **Dereferenziazione**. Restituisce l'elemento (*il valore*) puntato dalla posizione corrente dell'iteratore.
- `->`: **Accesso a Membri**. Permette di accedere ai membri di una struttura o classe puntata dall'iteratore, senza dover dereferenziare esplicitamente.
- `++` / `--`: **Avanzamento/Arretramento**. Muove l'iteratore all'elemento successivo (++) o precedente (--). *(Attenzione: non tutti gli iteratori supportano --)*.
- `==` / `!=`: **Confronto**. Controlla se due iteratori puntano alla **stessa posizione** all'interno del container.
- `=`: **Assegnazione**. Assegna la posizione di un iteratore a un altro.

Ogni classe container nella Standard Template Library (STL) del C++ fornisce le stesse funzioni membro di base per l'utilizzo degli iteratori, definendo così un'interfaccia uniforme per l'accesso ai dati. Queste funzioni sono essenziali per permettere a tutti gli algoritmi STL di lavorare con qualsiasi tipo di container, indipendentemente dalla sua implementazione interna

cosa sono e funzioni membro di base?

Le funzioni membro di base sono i metodi standard che ogni classe container deve implementare per garantire la compatibilità con gli algoritmi della Standard Template Library (STL).
Queste funzioni definiscono l'interfaccia minima per l'utilizzo degli iteratori, permettendo così agli algoritmi di lavorare con qualsiasi container senza conoscerne i dettagli implementativi interni.

Le due funzioni membro fondamentali sono begin() ed end(), che definiscono il range di elementi su cui un algoritmo può operare:

- **begin()**: Restituisce un iteratore che punta all'inizio degli elementi validi del container,

ovvero alla posizione del primo elemento (se esiste).

- **end()**: Restituisce l'iteratore alla fine logica degli elementi, noto come past-the-end iterator. Questo iteratore punta alla posizione teorica immediatamente successiva all'ultimo elemento; non rappresenta un elemento valido e non deve mai essere dereferenziato.

Questo modello standardizzato offre due vantaggi cruciali per la programmazione generica in C++:

- **Semplice Criterio di Terminazione**: Il modello [begin(), end()) offre un criterio di terminazione universale e semplice per tutti i cicli di visita. Si può continuare a scorrere gli elementi finché l'iteratore corrente non raggiunge (non è uguale a) end().
- **Gestione dei Contenitori Vuoti**: I contenitori vuoti non richiedono trattamenti particolari o controlli espliciti. Quando un container è vuoto, begin() è direttamente uguale a end(). Di conseguenza, il ciclo di visita non viene eseguito nemmeno una volta, garantendo che il codice non tenti mai di accedere a dati inesistenti.

Listing 4: Inserzione di elementi in un set e stampa usando gli iteratori

```
#include <iostream>
#include <set>

using namespace std;

int main() {
// Definizione: crea un container 'set' che memorizza valori di tipo int.
// Un set mantiene gli elementi unici e ordinati automaticamente.
set<int> coll;

// coll.insert(): Metodo usato per aggiungere nuovi elementi.
coll.insert(3);
coll.insert(1);
coll.insert(5);
coll.insert(4);

// Tentativo di inserire il valore 1 una seconda volta.
// In un set, l'inserimento non avviene perche' gli elementi devono
   essere unici.
coll.insert(1);
coll.insert(6);
coll.insert(2);

// Stampa di tutti gli elementi usando gli iteratori.
// L'iterazione avviene in ordine crescente (1, 2, 3, 4, 5, 6) grazie
   alla natura del set.
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << " ";
}

cout << endl;
return 0;
}
```

Il set è composto da interi: gli elementi saranno in ordine ascendente (usa $<$). È possibile anche ordinarli in maniera diversa, ad esempio in ordine discendente, con una struttura del tipo: $set < int, greater < int >>$ **coll**
NB: lo spazio è volutamente lasciato altrimenti » viene interpretato come operatore shift!
Il nuovo elemento è inserito col metodo insert automaticamente, push back o push front non sono ammessi.

Listing 5: Esempio di utilizzo di std::multimap con chiavi duplicate

```
#include <iostream>
#include <map> // Necessario per multimap
#include <string>
using namespace std;

// Alias per semplificare il tipo: multimap<chiave: int, valore: string>
typedef multimap<int,string> IntStringMap;

int main() {
IntStringMap coll; // Dichiarazione del container multimap.

// Inserimento di elementi: si usano coppie (chiave, valore).
coll.insert(make_pair(5, "tagged"));
coll.insert(make_pair(1, "this"));
coll.insert(make_pair(4, "of"));
coll.insert(make_pair(6, "strings"));

// Inserimento con chiave duplicata (4): questo e' permesso in multimap.
coll.insert(make_pair(4, "of"));

coll.insert(make_pair(3, "multimap"));

// Iterazione e stampa dei soli valori.
// Il multimap garantisce l'ordinamento automatico in base alla chiave.
for(IntStringMap::iterator pos = coll.begin(); pos != coll.end(); ++pos)
   {
    // pos->second: L'iteratore accede alla coppia; '.second' e' il
       valore (stringa).
    // L'output finale riflettera' l'ordine delle chiavi: 1, 3, 4, 4, 5,
       6.
    cout << pos->second << " ";
}

cout << endl;
return 0;
}
```

Listing 6: Map come array associativo

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



int main() {

// Definisce un alias: chiave=stringa, valore=float.
typedef map<string,float> StringFloatMap;

StringFloatMap coll;

// Inserimento tramite l'operatore di array associativo [].
// Se la chiave non esiste, la crea; altrimenti, ne aggiorna il valore.
coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;

// Iterazione su tutti gli elementi (in ordine alfabetico di chiave).
for (StringFloatMap::iterator pos = coll.begin(); pos != coll.end(); ++
   pos) {
    // pos->first e' la chiave (stringa); pos->second e' il valore (float
       ).
    cout << "key: " << pos->first << " \\ "
            << "value: " << pos->second << endl;
}
return 0;
}
```

Gli iteratori possono avere altre funzionalità, oltre quelle di base. Le funzionalità addizionali dipendono dal tipo di container. Gli iteratori si dividono in due categorie:

- **Iteratori bidirezionali**: Possono agire in due direzioni, in avanti, con l'operatore di incremento o, indietro con l'operatore di decremento. In questa tipologia ricadono gli iteratori per le classi dei container list, set, multiset, map, multimap.
- **Iteratori ad accesso casuale**: Hanno gli operatori necessari a poter definire una "aritmetica degli iteratori" (analoga a quella dei puntatori). In questa tipologia ricadono gli iteratori per le classi dei container vector e deque.

NB: se si vuole scrivere codice generico (indipendente dal container) meglio usare solo gli operatori degli iteratori bidirezionali

1.4 Algoritmi della STL

La STL fornisce una serie di algoritmi per processare elementi in collezioni che offrono servizi di base quali ricerca, ordinamento, copia, ecc...
Gli algoritmi non sono funzioni membro delle classi container, ma funzioni globali che operano tramite iteratori. Il vantaggio è che sono implementati una volta per tutte indipendentemente dal container utilizzato.

Listing 7: Esempio di utilizzo degli algoritmi STL

```
#include <iostream>
#include <vector>
#include <algorithm> // Necessario per tutti gli algoritmi STL
using namespace std;

int main() {
vector<int> coll;
// Inserimento elementi in ordine arbitrario nel vettore.
coll.push_back(2);
coll.push_back(5);
coll.push_back(1);
coll.push_back(6);
coll.push_back(4);
coll.push_back(3);

// 1. Ricerca del Minimo e Massimo:
// min_element e max_element restituiscono un iteratore all'elemento
   trovato.
auto pos_min = min_element(coll.begin(), coll.end());
cout << "min: " << *pos_min << endl; // Stampa il valore minimo (1)

auto pos_max = max_element(coll.begin(), coll.end());
cout << "max: " << *pos_max << endl; // Stampa il valore massimo (6)

// 2. Ordinamento (Sort):
```

```
// L'algoritmo sort riordina gli elementi nell'intervallo [begin(), end()
   ).
sort(coll.begin(), coll.end()); // Risultato in coll: 1 2 3 4 5 6

// 3. Ricerca specifica (Find):
// Trova la prima occorrenza del valore 3 e restituisce l'iteratore alla
   posizione.
auto pos = find(coll.begin(), coll.end(), 3);

// 4. Inversione (Reverse):
// Inverte l'ordine degli elementi dall'elemento trovato (pos) fino alla
   fine.
reverse(pos, coll.end()); // Risultato in coll: 1 2 3 6 5 4

// Stampa tutti gli elementi del vettore dopo le modifiche.
for (pos = coll.begin(); pos != coll.end(); ++pos)
    cout << *pos << " ";

cout << endl;
return 0;
}
```

1.4.1 Intervalli

Tutti gli algoritmi lavorano con uno o più intervalli (range) di elementi. Bisogna passare come argomento l'inizio e la fine di un intervallo, anziché l'intera collezione come un unico elemento. E compito dell'utente assicurarsi che l'intervallo sia valido. Ogni algoritmo processa intervalli chiusi a sinistra (come quelli usati dagli iteratori).

Listing 8: Algoritmo for_each()

```
#include <iostream>
#include <vector>
#include <algorithm> // Necessario per for_each
using namespace std;

// Funzione predicato: definita per stampare ogni elemento ricevuto in
   input.
void print (int elem) {
cout << elem << ' ';
}

int main() {
vector<int> coll;
// Inserimento elementi da 1 a 9 nel vettore.
for (int i=1; i<=9; ++i) {
    coll.push_back(i);
}
```

```
// for_each: Applica la funzione 'print' a tutti gli elementi nel range [
   begin, end).
// range: coll.begin(), coll.end()
// operation: print
for_each (coll.begin(), coll.end(), print);

cout << endl;
return 0;
}
```

1.4.2 Predicati

Un predicato è un tipo speciale di funzione ausiliaria per gli algoritmi, restituiscono un valore booleano. Sono spesso usati come criteri di discriminazione nelle procedure di ordinamento e di ricerca e possono essere di due tipi: **unari** o **binari**. Inoltre devono restituire sempre lo stesso risultato dato lo stesso input.

- **Predicati unari**: Controllano specifiche proprietà di un singolo argomento. Esempio: funzione isPrime.
- **Predicati binari**: Usati per confrontare specifiche proprietà di due argomenti.

Necessario quando, ad esempio, voglio ordinare una collezione di elementi per cui non è definito l'operatore **<**.

Listing 9: Algoritmo find_if(): uso di un predicato per la ricerca

```
#include <iostream>
#include <list>
#include <algorithm> // Necessario per find_if
#include <cstdlib> // Necessario per la funzione abs()
using namespace std;

// PREDICATO: Funzione che definisce il criterio di ricerca.
// Deve restituire un valore booleano (true o false).
bool isPrime (int number) {
int num = abs(number); // Considera solo il segno positivo
if (num <= 1) return false; // 0 e 1 non sono primi

// Controlla l'esistenza di un divisore > 1.
for (int divisor = num/2; divisor > 1; --divisor) {
    if (num % divisor == 0) return false;
}
// Se il ciclo finisce senza trovare divisori, il numero e' primo.
return true;
}

int main() {
list<int> coll;
// Inserisce elementi da 24 a 30.
```

```
for (int i=24; i<=30; ++i)
    coll.push_back(i);

// find_if: Cerca il primo elemento nel range [begin, end) per cui la
   funzione predicato (isPrime) restituisce true.
list<int>::iterator pos =
    find_if(coll.begin(), coll.end(), isPrime);

// Il valore trovato e' 29.
if (pos != coll.end()) {
    cout << *pos << " is first prime number found" << endl;
} else {
    cout << "no prime number found" << endl;
}
return 0;
}
```