Standard Template Library

Serafino Cicerone

Dipartimento di Ingegneria Elettrica
Università degli Studi dell'Aquila
I-67040 Monteluco di Roio, L'Aquila, Italy

cicerone@ing.univaq.it

Sommario

- Il presente documento rappresenta una dispensa didattica. Il suo scopo è quello di fornire allo studente una introduzione alla Standard Template Library del C++.
  La dispensa può essere considerata come materiale didattico integrativo per il corso di *Programmazione ad Oggetti*.
- Il materiale presente in questa dispensa proviene principalmente dal seguente volume:
  "C++ Tecniche Avanzate di Programmazione, H.M. Deitel , P.J. Deitel. Apogeo, 2001.
- La versione della presente dispensa è aggiornata all'Anno Accademico 2003/04.

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

La genericità è l'ultimo dei principi fondamentali analizzato nella dispensa *Tecnologia ad Oggetti*. Illustriamo nuovamente il concetto attraverso un breve esempio.

Supponiamo di dover realizzare un modulo software per la gestione della lista dei nostri personali contatti telefonici. A tale scopo, decidiamo di progettare e realizzare due classi:

- `ContattoTelefonico`, per poter creare oggetti che rappresentano le persone che sono presenti sulla nostra rubrica telefonica;
- `ElencoTelefonico`, per poter gestire un numero arbitrario di oggetti appartenenti alla classe `ContattoTelefonico`.

In particolare, la classe `ElencoTelefonico` offre servizi per effettuare le classiche operazioni di gestione: *inserimento*, *cancellazione*, *ricerca* e *modifica*.

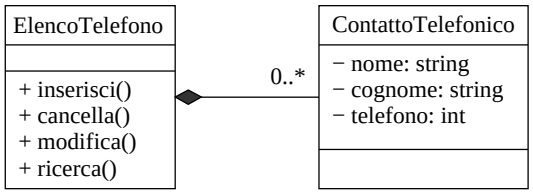

Figura 1: Relazione tra le classi `ElencoTelefonico` e `ContattoTelefonico`.

Supponiamo anche che la classe `ElencoTelefonico` faccia uso di strutture dati complicate (per gli attributi) ed algoritmi molto efficienti (per le operazioni). In quasta situazione, se, successivamente, fosse necessario dover gestire (tramite le stesse operazioni) anche le liste dei CD e dei libri personali, si potrebbe recuperare il codice prodotto per la classe `ElencoTelefonico` nel seguente modo:

1. copiare la classe `ElencoTelefonico` nella classe `ElencoCD`, sostituendo le occorrenze di oggetti `ContattoTelefonico` con oggetti di tipo `CD`
2. copiare la classe `ElencoTelefonico` nella classe `ElencoLibri`, sostituendo le occorrenze di oggetti `ContattoTelefonico` con oggetti di tipo `Libro`

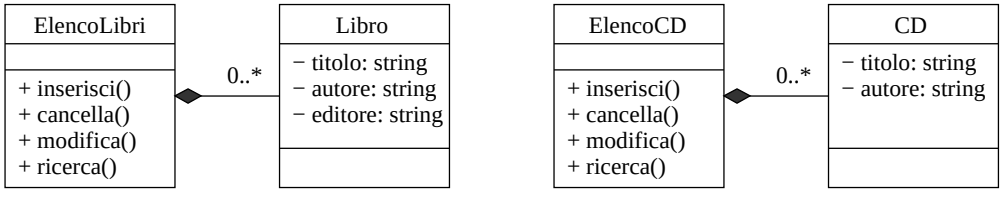

Figura 2: Relazione tra le classi `ElencoLibro` e `Libro` e tra le classi `ElencoCD` e `CD`.

Questo riuso di codice potrebbe aumentare notevolmente la nostra produttività ma richiederebbe il mantenimento di tre copie di codice quasi identico. In un caso come questo, la genericità fornisce un notevole aiuto. È possibile progettare una classe parametrica `Elenco<>` (ovvero, si potrebbe lasciare indefinita la classe che specifica gli oggetti interni; questa classe viene dunque considerata come un parametro da istanziare solo in fase di compilazione/esecuzione), e poi istanziare il parametro a seconda delle necessità. A questo punto potremmo semplicemente istanziare il parametro di

`Elenco<>` per ottenere classi specifiche come `Elenco<ContattoTelefonico>`, `Elenco<CD>` ed anche `Elenco<Libro>`. Questa soluzione eviterebbe il mantenimento di più copie dello stesso software.

Riassumendo, la genericità permette di definire classi parametriche; una classe parametrica usa classi interne come parametro. Le classi parametriche offrono i vantaggi del codice donato senza il lavoro aggiuntivo dovuto alla manutenzione replicata. Questo esempio evidenzia alcuni vantaggi della genericità nell'ambito della programmazione orientata agli oggetti:

1. riuso del software,
2. semplicità di manutenzione,
3. comprensibilità del codice.

Il linguaggio C++ (uno dei linguaggi di riferimento del corso) possiede i costrutti linguistici per poter creare classi parametriche; nel C++, le classi parametriche sono definite *template*. Considerando di nuovo l'esempio precedente, possiamo affermare che il C++ ci consente di progettare il template `Elenco<>`. Ma quest'esempio ci permette di puntualizzare anche un altro aspetto: vista l'assoluta generalità della classe `Elenco<>` (una classe contenitore, parametrica rispetto agli oggetti contenuti, che offre come servizi le classiche operazioni di gestione: inserimento, cancellazione, ricerca e modifica), per caso il linguaggio già possiede in libreria questo tipo di template? La risposta a questa domanda è affermativa, poiché il C++ mette a disposizione la Standard Template Library (d'ora in avanti definita brevemente STL).

La STL è stata sviluppata da Alexander Stepanov e Meng Lee presso la Hewlett-Packard. Essa ha origine dalle loro ricerche nel campo della programmazione generica (*generic programming*), con il contributo significativo di David Musser. La STL è ormai inclusa nello standard ANSI/ISO, e per questo implementata dai principali compilatori. La libreria contiene diversi tipi di contenitori: ogni contenitore gestisce gli oggetti contenuti secondo le politiche dei classici tipi di dato astratto, quali liste, pile, code, code di priorità e così via. La STL viene utilizzata in grande scala; grazie ad essa non solo possiamo risparmiare una considerevole quantità di tempo e di sforzi, ma l'uso di componenti riutilizzabili garantisce come risultato programmi di migliore qualità.

Efficienza 1 *Per ogni particolare applicazione possono essere appropriati diversi tipi di container della STL. Scegliete quelli che si adottano meglio al vostro caso e che permettono di raggiungere un livello ottimale di efficienza (per esempio, un buon bilancio tra velocità e dimensioni del programma). L'efficienza è stato un obiettivo cruciale nella progettazione della STL*

Efficienza 2 *Le funzionalità della libreria standard sono implementate per operare con efficienza su una vasta gamma di applicazioni. Per le applicazioni in cui sono necessarie prestazioni eccezionalmente elevate, può essere necessario scrivere un'implementazione personale di tali funzionalità.*

Ingegneria del Software 1 *L'approccio STL consente di scrivere programmi generali in modo tale che il codice non dipenda dal particolare container utilizzato. Questo principio è alla base della cosiddetta programmazione generica.*

In C e nel C++ di base si accede agli elementi di un array tramite i puntatori. Usando la STL, invece, si accede agli elementi dei container con oggetti iteratore che, in un certo senso, sono assimilabili a puntatori ma si comportano in modo più "intelligente". Le classi iteratore sono progettate per essere utilizzate in modo generico su qualsiasi container. I container incapsulano

alcune operazioni primitive, ma gli algoritmi della STL sono implementati in modo indipendente dai container.

Questa dispensa vuole essere una semplice e breve introduzione ai tre componenti principali della STL: i *container*, gli *iteratori* e gli *algoritmi*. Per la presentazione dei concetti fondamentali della STL viene usato un approccio "live-code", cioè la spiegazione corredata da un numero consistente di esempi reali. I container della STL includono le strutture dati più utilizzate e più importanti ed ognuna di esse è definita come template, per cui è possibile farle contenere i tipi di dato definiti nelle vostre applicazioni. Nei corsi di studio precedenti sono state studiate e create strutture dati come liste concatenate, code, pile e alberi; gli elementi di queste strutture erano collegati tramite puntatori. Ma il codice che fa uso dei puntatori è complesso e la più piccola omissione o disattenzione può comportare serie violazioni di accesso o gravi errori di perdita di memoria. Se si vogliono poi implementare altre strutture dati, come deque, code di priorità, insiemi e mappe, sarebbe necessario impiegare una notevole quantità di lavoro.

Ingegneria del Software 2 *Evitate di inventare la ruota ogni volta: servitevi dei componenti riutilizzabili della libreria standard. La STL include la maggior parte delle strutture dati più utilizzate come contenitori e fornisce molti tra gli algoritmi più comuni per accedere e elaborare i dati di tali container.*

Collaudo 1 *Se utilizziamo le strutture dati che si basano sui puntatori, dobbiamo provvedere noi stessi al debugging per verificare che le nostre strutture dati, le classi e gli algoritmi funzionino correttamente. È facile commettere errori quando si opera ad un livello così basso: perdite di memoria e violazioni di accesso divengono errori tipici. Per la maggior parte dei programmatori, e per la maggior parte delle applicazioni, le strutture dati della STL sono sufficienti. Utilizzando il codice della STL risparmierete una considerevole quantità di tempo di debugging e di messa a punto. L'unico punto debole di questo approccio riguarda i progetti di grandi dimensioni, perché il tempo di compilazione dei template può essere considerevole.*

Ogni container della STL ha funzioni membro associate ed alcune delle funzionalità che esse realizzano sono comuni a tutti i container, mentre altre sono tipiche di alcuni particolari container. Illustreremo la maggior parte delle funzionalità comuni con i template di classe `vector`, `list` e `deque`, mentre introdurremo le funzionalità specifiche di alcuni container negli esempi relativi a tutti gli altri container della STL. Il seguente programma può essere considerato come esempio introduttivo ai componenti principali della STL: contenitori, iteratori ed algoritmi generici:

```
 1 // STL: esempio introduttivo
 2 #include <iostream>
 3 #include <stdlib.h>
 4 #include <list>      // per usare il contenitore "list" delle STL
 5 #include <algorithm> // per usare gli algoritmi generici (validi
 6                      // per tutti i contenitori della STL)
 7
 8 using namespace std;
 9
10 int main()
11 {
12   list<int> contenitore;  // creazione di un contenitore lista di interi
13
```

```
14   int x;   // variabile usata per leggere interi da input
15
16   cout << "Inserisci sequenza di interi (0 per terminare) :\n" ;
17
18   // ---- ciclo lettura da input ed inserimento nel contenitore
19   do {
20        cin >> x;
21        contenitore.push_back(x);   // inserisco valore corrente
22                                    // alla fine del contenitore
23      } while ( x != 0);
24
25   contenitore.sort();  // uso metodo sort() del contenitore list<>
26
27   ostream_iterator<int> output(cout, " "); // creazione iteratore di output
28
29   // ---- dichirazione di 2 iteratori per list<int>:
30   //      i punta al primo elemento del contenitore
31   //      j punta all'elemento successivo all'ultimo elemento del contenitore
32   list<int>::iterator i = contenitore.begin(), j = contenitore.end();
33
34   cout << "Sequenza ordinata: \n";
35   copy(i, j, output );  // stampa in output la sequenza tramite
36                         // algoritmo generico della STL
37   return 0;
38 }
```

ESECUZIONE:

```
Inserisci sequenza di interi (0 per terminare) :
2 45 80 9 -34 12 5 0
Sequenza ordinata:
-34 0 2 5 9 12 45 80
```

Le linee 4 e 5 mostrano i file da includere per poter usare il contenitore `list<>` e gli algoritmi generici della STL; la linea 12 crea un oggetto di tipo `list<int>` (dunque, a tempo di compilazione viene specificata la natura degli oggetti contenuti nel container); la linea 21 mostra l'uso di un metodo del template `list<>` per l'inserimento di nuovi oggetti, mentre la linea 25 mostra un metodo per l'ordinamento. Gli iteratori vengono introdotti nelle linea 32, e vengono passati all'algoritmo generico `copy()` per poter permettere a quest'ultimo di operare sulle strutture dati interne del container.

Nelle seguenti sezioni vengono introdotti i container, gli iteratori e gli algoritmi generici.

1.1 Introduzione ai container

I tipi di container presenti nella STL sono elencati nella Tabella 1. I container si suddividono in tre grandi categorie: *container sequenziali*, *container associativi* e *adattatori*. I primi due tipi sono detti anche container di prima classe. Ci sono altri quattro tipi di container che sono considerati dei "quasi container": gli *array* in stile C, le *string*, i *bitset* preposti alla manutenzione di valori

flag binari e i *valarray* che effettuano operazioni matematiche vettoriali ad alta velocità (questa classe è stata ottimizzata per eseguire i calcoli in modo molto efficiente e non è così flessibile come i container di prima classe). Questi quattro tipi sono "quasi container" perché prevedono funzionalità simili a quelle dei container di prima classe, ma non ne supportano tutte le caratteristiche.

Classe container della STL	**Descrizione**
container sequenziali	
`vector`	inserimenti/eliminazioni rapidi in coda; accesso diretto a qualsiasi elemento
`deque`	inserimenti/eliminazioni rapidi in testa o in coda; accesso diretto a qualsiasi elemento
`list`	lista a doppio concatenamento; inserimenti ed eliminazioni rapidi ovunque
container associativi	
`set`	ricerca rapida; non sono consentiti duplicati
`multiset`	ricerca rapida; sono consentiti duplicati
`map`	contiene coppie `(chiave,valore)`; mapping uno-a-uno tra le chiavi ed i valori (non è consentito duplicare chiavi); ricerca rapida di una chiave
`multimap`	contiene coppie `(chiave,valore)`; mapping uno-a-molti tra le chiavi ed i valori (è consentito duplicare chiavi); ricerca rapida di una chiave
adattatori di container	
`stack`	l'ultimo inserito è il primo estratto: last-in-first-out (LIFO)
`deque`	il primo inserito è il primo estratto: first-in-first-out (FIFO)
`priority_queue`	l'elemento di priorità più alta è sempre il primo elemento estratto

Tabella 1: Le classi container della STL

La STL è stata progettata in modo che i container forniscano funzionalità simili. Ci sono molte operazioni generiche, come la funzione `size()`, che si applicano a tutti i container, e ci sono altre operazioni che si applicano a sottoinsiemi di container simili. Questo fatto incoraggia l'estensibilità della STL con nuove classi. Le funzioni comuni[1] a tutti i container della STL sono presentate in Tabella 2.

I file di intestazione di ogni container della STL sono riportati in Tabella 3. IMPORTANTE: per utilizzare i container su alcuni compilatori sarà necessario premettere l'istruzione `using namespace std;` su altri sarà necessario anteporre l'operatore di scope `std::` ad ogni nome di template e ad ogni nome di algoritmo della STL. Per maggiori informazioni consultate la documentazione del vostro compilatore.

La Tabella 4 mostra le definizioni di tipo dei container di prima classe che servono a creare sinonimi per i tipi di dato. Queste definizioni sono utilizzate nelle dichiarazioni generiche delle variabili, dei parametri di funzione e dei valori restituiti da esse. Per esempio, `value_type` in ogni container è sempre una typedef che rappresenta il tipo di valore memorizzato nel container.

Efficienza 3 *La STL non utilizza generalmente l'ereditarietà e le funzioni virtuali, favorendo la programmazione generica con i template per ottenere prestazioni ottimali in fase di esecuzione.*

Portabilità 1 *La STL diventerà certamente la metodologia favorita per lavorare sui container. Utilizzando la STL il vostro codice sarà più portabile.*

[1] Gli overloading di `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==` e `operator!=` non sono disponibili per le `priority_queue`

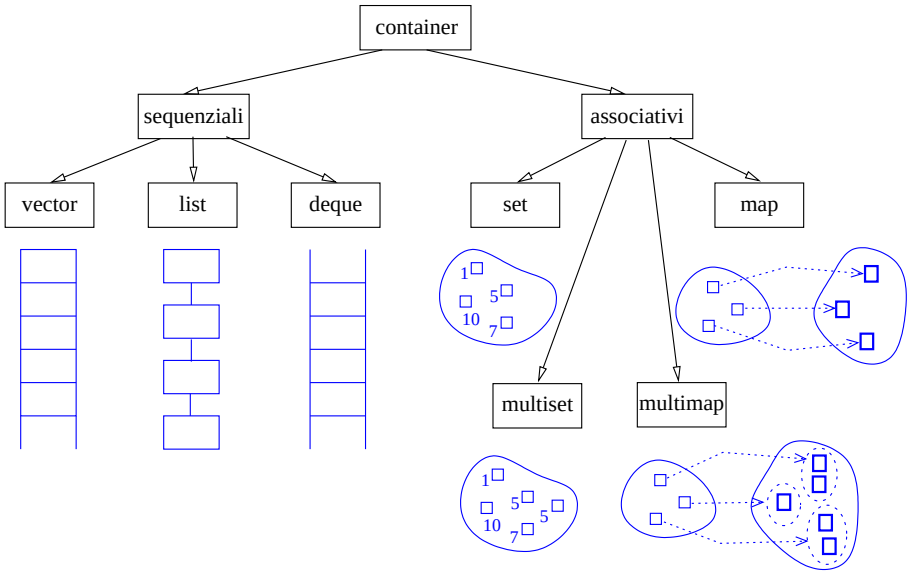

Figura 3: Rapresentazione schematica dei vari tipi di containers.

Efficienza 4 *Studiate i componenti della STL. Scegliere il container più adatto al vostro problema può ottimizzare l'efficienza e minimizzare la memoria richiesta dal programma.*

Ingegneria del Software 3 IMPORTANTE*: se decidete di utilizzare un container della STL, è importante che vi assicuriate che il tipo di elemento da memorizzare supporti un insieme minimo di funzionalità. Quando si inserisce un elemento in un container, viene effettuata una copia di tale elemento. Perciò il tipo dell'elemento dovrebbe avere un proprio costruttore di copie e un proprio operatore di assegnamento (ciò serve soltanto se la copia di default membro a membro non effettua correttamente l'operazione di copia su quel tipo di elemento). Sono necessari anche gli operatori di uguaglianza (`==`) e di minore (`<`).*

1.2 Introduzione agli iteratori

Gli iteratori hanno molte caratteristiche in comune con i puntatori e servono ad accedere agli elementi dei container di prima classe (e ad altri scopi che vedremo tra breve). Gli iteratori contengono informazioni di stato sensibili ai container su cui operano, per cui sono implementati nel modo appropriato per ciascun tipo di container. Tuttavia alcune operazioni degli iteratori hanno lo stesso significato su tutti i container. Per esempio, l'operatore di risoluzione del riferimento (`*`) dereferenzia un iteratore in modo da poter utilizzare l'elemento a cui punta. L'operazione `++` su un iteratore restituisce un iteratore all'elemento successivo del container (un pò come l'incremento di un puntatore in un array fa si che esso punti all'elemento successivo dell'array).

I container di prima classe della STL forniscono le funzioni membro `begin()` e `end()`. La funzione `begin()` restituisce un iteratore che punta al primo elemento del container mentre la funzione `end()` restituisce un iteratore che punta al primo elemento dopo la fine del container (un elemento che non esiste!). Se l'iteratore `iter` punta a un particolare elemento, `++iter` punta all'elemento successivo e `*iter` riferisce l'elemento a cui punta `iter`.

A fronte delle seguenti linee di codice:

```
vector<int> v;
```

Funzioni membro comuni a tutti i container della STL	**Descrizione**
costruttore di default	Costruttore per l'inizializzazione di default del container. Di norma, ogni container ha diversi costruttori che forniscono una varietà di metodi di inizializzazione del container
costruttore di copia	Costruttore che inizializza il container come copia di un container esistente dello stesso tipo
distruttore	Funzione per la distruzione del container
`empty`	Restituisce true se il container non contiene elementi, false altrimenti
`max_size`	Restituisce il numero massimo di elementi per un container
`size`	Restituisce il numero di elementi presenti correntemente in un container
`operator=`	Assegna un container a un altro
`operator<`	Restituisce true se il primo container è minore del secondo, false altrimenti
`operator<=`	Restituisce true se il primo container è minore o uguale al secondo, false altrimenti
`operator>`	Restituisce true se il primo container è maggiore del secondo, false altrimenti
`operator>=`	Restituisce true se il primo container è maggiore o uguale al secondo, false altrimenti
`operator==`	Restituisce true se il primo container è uguale al secondo, false altrimenti
`operator!=`	Restituisce true se il primo container non è uguale al secondo, false altrimenti
`swap`	Scambia gli elementi dei due container
Funzioni dei soli container di prima classe	
`begin`	Le due versioni di questa funzione restituiscono un `iterator` o un `const_iterator` che riferisce il primo elemento del container
`end`	Le due versioni di questa funzione restituiscono un `iterator` o un `const_iterator` che riferisce la posizione successiva dopo la fine del container
`rbegin`	Le due versioni di questa funzione restituiscono un `iterator` o un `const_iterator` che riferisce l'ultimo elemento del container
`rend`	Le due versioni di questa funzione restituiscono un `iterator` o un `const_iterator` che riferisce la posizione prima del primo elemento del container
`erase`	Elimina uno o più elementi del container
`clear`	Elimina tutti gli elementi del container

Tabella 2: Le funzioni comuni a tutti i container della STL.

```
... // vengono aggiunti alcuni elementi a v tramite push_back()
vector::iterator i1 = v.begin();
vector::iterator i2 = v.end();
```

vengono creati gli iteratori `i1` e `i2` come da Figura 4. Notate che tramite la dereferenziazione di `i1`, `*i1` rappresenta l'oggetto nella prima posizione del contenitore sequenziale.

L'utilizzo di oggetti di tipo `iterator` è consentito per riferire elementi di un container che può essere modificato, mentre nel caso dell'accesso a container non modificabili si può utilizzare un oggetto di tipo `const_iterator`.

È possibile associare degli oggetti iteratori anche ai flussi di dati. Il programma in `codice1.cpp` mostra l'input dall'unità standard utilizzando un `istream_iterator` e l'output sull'unità standard di output utilizzando un `ostream_iterator`. Il programma effettua l'input di due interi dalla tastiera e ne visualizza la somma.

File di intestazione dei container della STL	
`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	contiene `queue` e `priority_queue`
`<stack>`	
`<map>`	contiene `map` e `multimap`
`<set>`	contiene `set` e `multiset`
`<bitset>`	

Tabella 3: File di intestazione per i container della STL

typedef	Descrizione
`value_type`	Il tipo di elemento memorizzato nel container
`reference`	Un riferimento al tipo di elemento memorizzato nel container
`const_reference`	Un riferimento costante al tipo di elemento memorizzato nel container. Tale riferimento si può utilizzare soltanto per leggere gli elementi del container e per effettuare operazioni const
`pointer`	Un puntatore al tipo di elemento memorizzato nel container
`iterator`	Un iteratore che punta al tipo di elemento memorizzato nel container
`const_iterator`	Un iteratore costante che punta al tipo di elemento memorizzato nel container e che può essere utilizzato soltanto per leggere gli elementi
`reverse_iterator`	Un iteratore inverso che punta al tipo di elemento memorizzato nel container. Questo tipo di iteratore serve a iterare al contrario attraverso un container
`const_reverse_iterator`	Un iteratore inverso costante al tipo di elemento memorizzato nel container e che può essere utilizzato soltanto per leggere gli elementi. Questo tipo di iteratore serve a iterare al contrario attraverso un container
`difference_type`	Il tipo di risultato che deriva dalla sottrazione di due iteratori, che riferiscono lo stesso container[a]
`size_type`	Il tipo utilizzato per contare gli elementi in un container e per l'indicizzazione in un container sequenziale[b]

[a] `operator-` non è definita per gli iteratori di liste e per i container associativi

[b] Non è possibile utilizzare l'operatore di indicizzazione su di una lista

Tabella 4: Definizioni di tipo comuni dei container di prima classe.

```
 1 // codice1.cpp
 2 // Gestione input e output con iteratori.
 3 #include <iostream>
 4 #include <iterator>
 5
 6 using namespace std;
 7
 8 int main()
 9 {
10   cout << "Immetti due interi: ";
11
12   istream_iterator< int > inputInt( cin );
13   int number1, number2;
14
```

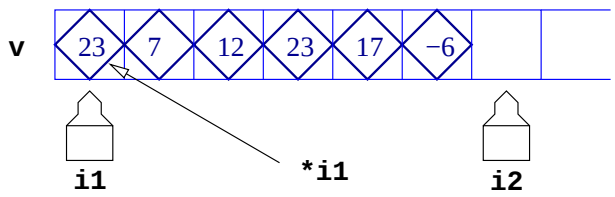

Figura 4: Creazione di iteratori tramite le istruzioni `vector<int>::iterator i1=v.begin()` e `vector<int>::iterator i2=v.end()`. La dereferenziazione `*i1` restituisce il primo oggetto del vector `v`.

```
15   number1 = *inputInt; // legge primo int da input standard
16   ++inputInt;  // sposta l'iteratore sul valore successivo in input
17   number2 = *inputInt; // legge il successivo int dall'input standard
18
19   cout << "La somma e': ";
20
21   ostream_iterator< int > outputInt( cout );
22
23   *outputInt = number1 + number2;  // output su cout
24   cout << endl;
25   return 0;
26 }
```

La linea 12:

```
istream_iterator< int > inputInt( cin );
```

crea un `istream_iterator` in grado di effettuare l'input di valori `int` in maniera sicura per il tipo di dato dall'oggetto di input standard `cin`. La linea 15:

```
number1 = *inputInt; // legge primo int da input standard
```

dereferenzia l'iteratore `inputInt` in modo da leggere il primo intero da `cin` e assegnarlo a `number1`. Osservate l'uso dell'operazione di dereferenziazione `*` per ottenere il valore dallo stream associato a `inputInt`: è un'operazione che assomiglia alla dereferenziazione di un puntatore. La linea 16:

```
++inputInt; // sposta l'iteratore sul successivo valore in input
```

posiziona l'iteratore `inputInt` sul successivo valore dello stream di input. La linea 17:

```
number2 = *inputInt; // legge succ.  int dall'input standard
```

effettua l'input dell'intero successivo da `inputInt` e lo assegna a `number2`. La linea 21:

```
ostream_iterator< int > outputInt( cout );
```

crea un `ostream_iterator` in grado di effettuare l'output di valori `int` sull'oggetto di output standard `cout`. La linea 23:

```
*outputInt = number1 + number2; // output su cout
```

effettua l'output di un intero su `cout` assegnando a `*outputInt` la somma di `number1` e `number2`. Osservate l'uso dell'operatore di dereferenziazione `*` per utilizzare `*outputInt` come *lvalue* nell'istruzione di assegnamento. Se volete inviare in output un altro valore con `outputInt`, dovete incrementare l'iteratore con `++` (come preincremento o postincremento).

Collaudo 2 *L'operatore di dereferenziazione* `*` *utilizzato su un* `const_iterator` *restituisce un riferimento* `const` *all'elemento del container, inibendo l'utilizzo di funzioni membro non* `const`*.*

Errore Tipico 1 *Se tentate di dereferenziare un iteratore posizionato al di fuori del suo container commettete un errore logico in fase di esecuzione. In particolare, l'iteratore restituito da* `end()` *non può essere dereferenziato.*

Errore Tipico 2 *Se tentate di creare un iteratore non* `const` *per un container* `const` *commettete un errore di sintassi.*

La Tabella 5 mostra le categorie di iteratori utilizzati dalla STL. Ciascuna di esse fornisce un insieme specifico di funzionalità.

Categoria	**Descrizione**
input	Utilizzato per leggere un elemento da un container. Un iteratore di input si può muovere solo in avanti (cioè dall'inizio del container alla fine) di un elemento alla volta. Gli iteratori di input supportano solo accessi sequenziali ad un passaggio: lo stesso iteratore di input non può essere utilizzato per ripassare una seconda volta su una sequenza
output	Utilizzato per scrivere un elemento in un container. Un iteratore di output si può muovere solo in avanti di un elemento alla volta. Gli iteratori di output supportano solo accessi sequenziali ad un passaggio: lo stesso iteratore di output non può essere utilizzato per ripassare una seconda volta su una sequenza
forward	Combina le funzionalità degli iteratori di input e di output, e mantiene la propria posizione nel container (come informazione di stato)
bidirezionale	Combina le funzionalità di un iteratore forvard con la possibilità di muoversi indietro (cioè dalla fine del container all'inizio). Gli iteratori bidirezionali consentono l'accesso sequenziale nelle due direzioni e passaggi multipli
ad accesso casuale	Combina le funzionalità di un iteratore bidirezionale con la possibilità di accedere direttamente a qualsiasi elemento del container, ovvero di saltare in avanti e all'indietro di un numero arbitrario di elementi

Tabella 5: Le categorie di iteratori

La Figura 5 illustra la gerarchia delle categorie di iteratori. Percorrendo la gerarchia dall'alto in basso, ogni categoria supporta tutte le funzionalità delle categorie sopra di essa. Quindi i tipi di iteratore più specifici sono in cima alla gerarchia, mentre il più generale è in fondo a essa. Vogliamo sottolineare che *non* si tratta di una gerarchia di ereditarietà.

La categoria di iteratore supportata da un container determina se tale container può essere utilizzato con gli specifici algoritmi della STL. I container che supportano gli iteratori ad accesso casuale possono essere utilizzati con tutti gli algoritmi della STL. Come vedremo, i puntatori in array possono sostituire gli iteratori nella maggior parte degli algoritmi della STL, inclusi quelli che richiedono gli iteratori ad accesso casuale. La Tabella 6 mostra la categoria di iteratore supportata da ogni container della STL. Notate che solo `vector`, `deque`, `list`, `set`, `multiset`, `map` e `multimap` (cioè i container di prima classe) si possono attraversare con gli iteratori.

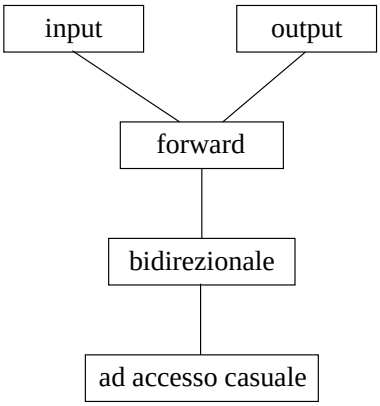

Figura 5: La gerarchia delle categorie di iteratori.

Ingegneria del Software 4 *Se l'iteratore più specifico porta a prestazioni accettabili, conviene utilizzare questo tipo di iteratore, per produrre componenti il più possibile riutilizzabili.*

Container	**Tipo di operatore supportato**
Container sequenziali	
`vector`	ad accesso casuale
`deque`	ad accesso casuale
`list`	bidirezionale
Container associativi	
`set`	bidirezionale
`multiset`	bidirezionale
`map`	bidirezionale
`multimap`	bidirezionale
Adattatori al container	
`stack`	nessun iteratore supportato
`queue`	nessun iteratore supportato
`priority_queue`	nessun iteratore supportato

Tabella 6: Tipo di iteratore supportato da ogni container della STL

La Tabella 7 mostra le typedef predefinite per i tipi di iteratori che si trovano nelle definizioni di classe dei container della STL. Non tutte le typedef sono definite su ogni container. Utilizziamo le versioni `const` degli iteratori per l'attraversamento di container a sola lettura e gli iteratori inversi per l'attraversamento dei container in direzione inversa.

Collaudo 3 *Le operazioni effettuate su un* `const_iterator` *restituiscono riferimenti* `const` *per evitare modifiche sugli elementi del container che sono manipolati. Utilizzate i* `const_iterator` *anziché gli* `iterator` *dove è appropriato.*

La Tabella 8 mostra le operazioni che si possono effettuare su ciascun tipo di iteratore. Notate che le operazioni di un tipo includono tutte le operazioni che precedono quel tipo nella figura. Notare inoltre che per gli iteratori di input e di output non è possibile salvare l'iteratore e utilizzare il valore salvato in un secondo tempo.

Typedef predefinite per i tipi di operatore	Direzione di ++	Funzionalità
`iterator`	avanti	lettura/scrittura
`const_iterator`	avanti	lettura
`reverse_iterator`	indietro	lettura/scrittura
`const_reverse_iterator`	indietro	lettura

Tabella 7: Typedef predefinite per gli iteratori

1.3 Introduzione agli algoritmi

Un aspetto cruciale della STL sta nel fatto che fornisce algoritmi che possono essere utilizzati in modo generale su diversi tipi di container. Gli algoritmi della STL sono comunemente utilizzati nella manipolazione dei container. L'inserimento, l'eliminazione, la ricerca, l'ordinamento e altre operazioni sono appropriate per alcuni o per tutti i container della STL.

La STL include all'incirca 70 algoritmi standard e per alcuni di essi forniremo degli esempi. Gli algoritmi operano sugli elementi dei container solo indirettamente tramite gli iteratori. Molti algoritmi operano su sequenze di elementi definite da coppie di iteratori, in cui il primo iteratore punta al primo elemento della sequenza e il secondo punta all'elemento successivo all'ultimo della sequenza.

La funzione membro `begin()` restituisce un iteratore al primo elemento di un container; `end()` restituisce un iteratore alla prima posizione dopo l'ultimo elemento del container. Gli algoritmi restituiscono spesso iteratori.

Un algoritmo come `find()`, per esempio, trova un elemento e restituisce un iteratore a tale elemento. Se l'elemento cercato non viene trovato, `find()` restituisce l'iteratore `end()`; testare se il risultato di `find()` è uguale a `end()` è un tipico modo per verificare se la ricerca ha avuto esito positivo. L'algoritmo `find()` può essere utilizzato con tutti i container della STL.

Ingegneria del Software 5 *La STL è stata implementata in modo conciso. Finora i progettisti di classi avrebbero associato gli algoritmi ai container, rendendo i primi funzioni membro dei secondi. Nella STL l'approccio è diverso. Gli algoritmi sono separati dai container e operano sugli elementi dei container in modo indiretto tramite gli iteratori. Questa separazione semplifica la scrittura di algoritmi generici, applicabili a molte altre classi container.*

Gli algoritmi della STL creano ancora un'altra opportunità per il riutilizzo del software. Sfruttando questi algoritmi di uso comune si risparmia tempo e fatica.

Se un algoritmo utilizza gli iteratori meno generali, può essere utilizzato anche con i container che supportano gli iteratori più generali. Alcuni algoritmi hanno bisogno di iteratori più generali, come per esempio l'ordinamento, che ha bisogno di iteratori ad accesso casuale.

Ingegneria del Software 6 *La STL è estensibile. È semplice aggiungere nuovi algoritmi senza dover modificare in alcun modo i container della STL.*

Ingegneria del Software 7 *Gli algoritmi della STL possono operare sui container della STL, sulle string C++ e sugli array basati su puntatori in stile C.*

Portabilità 2 *Dato che gli algoritmi della STL operano sui container solo indirettamente (tramite gli iteratori) si può utilizzare spesso lo stesso algoritmo su molti container diversi.*

La Tabella 10 mostra alcuni algoritmi che non modificano il container su cui sono applicati, mentre la Tabella 11 mostra gli algoritmi numerici che possono essere usati includendo il file di intestazione `<numeric>`.

Operazioni sull'iteratore	Descrizione
Tutti gli iteratori	
`++p`	preincrementa un iteratore
`p++`	postincrementa un iteratore
Iteratori di input	
`*p`	dereferenzia un iteratore per utilizzare il risultato come rvalue
`p = p1`	assegna un iteratore a un altro
`p == p1`	verifica se due iteratori sono uguali
`p != p1`	verifica se due iteratori sono diversi
Iteratori di output	
`*p`	dereferenzia un iteratore per utilizzare il risultato come lvalue
`p = p1`	assegna un iteratore a un altro
Iteratori forward	gli iteratori forward hanno tutte le funzionalità degli iteratori di input e di output
Iteratori bidirezionali	
`--p`	predecrementa un iteratore
`p--`	postdecrementa un iteratore
Iteratori ad accesso casuale	
`p += i`	incrementa l'iteratore `p` di `i` posizioni
`p -= i`	decrementa l'iteratore `p` di `i` posizioni
`p + i`	dà come risultato un iteratore posizionato in `p` incrementato di `i` posizioni
`p - i`	dà come risultato un iteratore posizionato in `p` decrementato di `i` posizioni
`p[ i ]`	restituisce un riferimento all'elemento che si scosta da `p` di `i` posizioni
`p < p1`	restituisce true se l'iteratore `p` è minore dell'iteratore `p1` (l'iteratore `p` si trova prima di `p1` nel container); altrimenti false
`p <= p1`	restituisce true se l'iteratore `p` è minore o uguale all'iteratore `p1` (l'iteratore `p` si trova prima o alla stessa posizione di `p1` nel container); altrimenti false
`p > p1`	restituisce true se l'iteratore `p` è maggiore dell'iteratore `p1` (l'iteratore `p` si trova dopo `p1` nel container); altrimenti false
`p >= p1`	restituisce true se l'iteratore `p` è maggiore o uguale all'iteratore `p1` (l'iteratore `p` si trova alla stessa posizione di `p1` o dopo di esso nel container); altrimenti false

Tabella 8: Operazioni per ogni tipo di iteratore

2 I container sequenziali

La STL fornisce tre container sequenziali: `vector`, `list` e `deque`. Le classi `vector` e `deque` si basano entrambe sugli array. La classe `list` implementa una lista concatenata.

Uno dei container più utilizzati della classe STL è `vector`. Un `vector` può cambiare dimensioni in modo dinamico. A differenza degli array in stile C e del C++, i `vector` possono essere assegnati tra di loro. Come negli array del C, gli indici di un `vector` non sono sottoposti a controlli di validità, ma questa funzionalità è accessibile tramite la funzione membro `at()`.

Efficienza 5 *L'inserimento di un elemento in coda a un* `vector` *è efficiente. Il* `vector` *cresce semplicemente se è necessario, in modo da contenere il nuovo elemento. È invece costoso l'inserimento (e allo stesso modo l'eliminazione) di un elemento in mezzo a un* `vector`*, perché deve essere spostata l'intera porzione del* `vector` *che segue l'elemento inserito (o eliminato): gli elementi di un* `vector`*, infatti, occupano celle di memoria contigue, proprio come gli array standard del C e del C++.*

Algoritmi di modifica del contenuto di container		
`copy()`	`remove()`	`reverse_copy()`
`copy_backward()`	`remove_copy()`	`rotate()`
`fill()`	`remove_copy_if()`	`rotate_copy()`
`fill_in()`	`remove_if()`	`stable_partition()`
`generate()`	`replace()`	`swap()`
`generate_n()`	`replace_copy ()`	`swap_ranges()`
`iter_swap()`	`replace_copy_if()`	`transform()`
`partition()`	`replace_if()`	`unique()`
`random_shuffle()`	`reverse()`	`unique_copy()`

Tabella 9: Algoritmi di modifica del contenuto di container

Algoritmi che non modificano il contenuto dei container		
`adjacent_find()`	`equal()`	`mismatch()`
`count()`	`find()`	`search()`
`count_if()`	`for_each()`	`search_n()`

Tabella 10: Algoritmi che non modificano il contenuto dei container

La Tabella 2 presenta le operazioni comuni a tutti i container della STL. Oltre a esse, ogni container fornisce di norma diverse altre funzionalità. Molte di queste funzionalità sono comuni a parecchi container, ma non è detto che siano ugualmente efficienti su ognuno di essi. Spesso occorre scegliere il container più appropriato per la propria applicazione.

Efficienza 6 *Le applicazioni che effettuano frequenti operazioni di inserimento ed eliminazione a entrambi i capi di un container usano di norma un* `deque` *anziché un* `vector`*. Anche se possiamo inserire ed eliminare elementi in testa e in coda sia in un* `vector` *che in un* `deque`*. La classe* `deque` *è più efficiente di* `vector` *per gli inserimenti e le eliminazioni in testa al container.*

Efficienza 7 *Le applicazioni che effettuano frequenti operazioni di inserimento ed eliminazione in mezzo o a alle estremità di un container utilizzano di norma una* `list`*, per la sua implementazione efficiente di queste operazioni in qualsiasi punto della struttura dati.*

Oltre alle operazioni comune descritte nella Tabella 2, i container sequenziali ne hanno in comune molte altre: `front()` che restituisce un riferimento al primo elemento del container, `back()` che restituisce un riferimento all'ultimo elemento del container, `push_back()` che inserisce un nuovo elemento alla fine del container e `pop_back()` che elimina l'ultimo elemento del container.

2.1 Il container sequenziale `vector`

La classe `vector` fornisce una struttura dati che occupa posizioni di memoria contigue. Ciò permette un accesso efficiente e diretto a qualsiasi elemento di un vector tramite l'operatore di indicizzazione `[]` utilizzato allo stesso modo degli array del C e del C++. La classe `vector` viene utilizzata di norma se i dati del container devono essere ordinati e devono essere accessibili semplicemente con un indice. Quando la memoria di un `vector` si esaurisce, il `vector` alloca automaticamente un'area di memoria contigua più grande, copia gli elementi originali nella nuova area e dealloca la vecchia area.

Efficienza 8 *Scegliete il container* `vector` *se desiderate prestazioni ottimali per l'accesso casuale.*

Algoritmi numerici
accumulate()
inner_product()
partial_sum()
adjacent_difference()

Tabella 11: Algoritmi numerici

Efficienza 9 *Gli oggetti della classe* `vector` *forniscono un rapido accesso indicizzato tramite l'overloading dell'operatore* `[]`*, perché sono memorizzati in posizioni di memoria contigue, proprio come gli array standard del C e del C+ +.*

Efficienza 10 *È più rapido inserire molti elementi in una volta sola che un elemento alla volta.*

Un aspetto importante di ogni container è il tipo di iteratore che supporta. Questo determina quali sono gli algoritmi che si possono applicare al container. Un vector supporta iteratori ad accesso casuale, su cui si possono applicare tutte le operazioni nella Tabella 9, dunque, tutti gli algoritmi della STL possono operare su di esso. Gli iteratori di un vector sono implementati di norma come puntatori a elementi del vector. Ogni algoritmo della STL che prende argomenti iteratore richiede che tali iteratori forniscono un livello minimo di funzionalità. Se un algoritmo richiede un iteratore forward, per esempio, esso può operare su qualsiasi container che fornisca iteratori forward, bidirezionali o ad accesso casuale. Se il container supporta le funzionalità minime degli iteratori, l'algoritmo può operare sul container.

Il programma in `codice2.cpp` illustra diverse funzioni del template di classe `vector`. Molte di esse sono disponibili per tutti i container di prima classe della STL. Per poter utilizzare la classe vector occorre includere il file di intestazione `<vector>`.

```
 1 // codice2.cpp
 2 // Test del template vector<>
 3 #include <iostream>
 4 #include <vector>
 5
 6 using namespace std;
 7
 8 template < class T >
 9 void printVector( const vector< T > &vec );
10
11 int main()
12 {
13    const int SIZE = 6;
14    int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15    vector< int > v;
16
17    cout << "La size iniziale di v e': " << v.size()
18         << "\nLa capacita' iniziale di v e': " << v.capacity();
19    v.push_back( 2 );  // il metodo push_back() e' in
20    v.push_back( 3 );  // ogni container sequenziale
21    v.push_back( 4 );
```

```
22    cout << "\nLa size di v e': " << v.size()
23         << "\ nLa capacita' di v e': " << v.capacity();
24    cout << "\nContenuto dell'array a usando notazione puntatore: ";
25
26    for ( int *ptr = a; ptr != a + SIZE; ++ptr )
27       cout << *ptr << ' ';
28
29    cout << "\ nContenuto del vettore v usando notazione iteratore: ";
30    printVector( v );
31
32    cout << "\nContenuto Reversed del vettore v: ";
33
34    vector< int >::reverse_iterator p2;
35
36    for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
37       cout << *p2 << ' ';
38    cout << endl;
39    return 0;
40 }
41
42 template < class T >
43 void printVector( const vector< T > &vec )
44 {
45    vector< T >::const_iterator p1;
46
47   for ( p1 = vec.begin(); p1 != vec.end(); p1++ )
48       cout << *p1 << ' ';
49 }
```

Esecuzione:

```
La size iniziale di v e' 0
La capacita' iniziale di v e' 0
La size di v e' 3
La capacita' di v e' 4
Contenuto dell'array a usando notazione puntatore: 1 2 3 4 5 6
Contenuto del vettore v usando notazione iteratore: 2 3 4
Contenuto Reversed del vettore v: 4 3 2
```

La linea 15

```
vector< int > v
```

dichiara l'istanza `v` della classe `vector` che memorizza valori `int`. Quando viene istanziato questo oggetto, viene creato un `vector` vuoto di dimensione 0 (corrispondente al numero di elementi memorizzati nel `vector`) e di capacità 0 (corrispondente al numero di elementi che possono essere memorizzati senza allocare ulteriore memoria per il `vector`).

Le linee 17 e 18

```
cout << "La size iniziale di v e': " << v.size()
     << "\nLa capacita' iniziale di v e': " << v.capacity();
```

mostrano le funzioni `size()` e `capacity()`, che all'inizio restituiscono entrambe 0 per `v`. La funzione `size()`, disponibile per ogni container, restituisce il numero di elementi memorizzati correntemente nel container. La funzione `capacity()` restituisce il numero di elementi che possono essere memorizzati nel `vector` prima che debba ridimensionarsi dinamicamente per accogliere ulteriori elementi.

Le linee 19-21:

```
v.push_back( 2 );  // il metodo push_back() e' in
v.push_back( 3 );  // ogni container sequenziale
v.push_back( 4 );
```

utilizzano la funzione `push_back()`, disponibile per tutti i container sequenziali, per aggiungere un elemento alla fine del `vector` (cioè nella successiva posizione disponibile). Se si aggiunge un elemento a un `vector` pieno, il `vector` si accresce automaticamente: in alcune implementazioni della STL il `vector` raddoppia le sue dimensioni automaticamente.

Efficienza 11 *Le implementazioni che raddoppiano il* `vector` *quando serve ulteriore spazio possono in realtà sprecare molto spazio. Per esempio, un* `vector` *pieno di 1.000.000 di elementi passa automaticamente a 2.000.000 elementi quando se ne deve aggiungere anche uno soltanto. Ciò lascia ben 999.999 elementi inutilizzati. I programmatori possono utilizzare* `resize()` *per controllare meglio la gestione dello spazio.*

Le linee 22 e 23 utilizzano `size()` e `capacity()` per mostrare come cambia la dimensione e la capacità del `vector` dopo le operazioni di `push_back()`. La funzione `size()` restituisce 3, il numero di elementi aggiunti al `vector`. La funzione `capacity()` restituisce 4, indicando che possiamo aggiungere un altro elemento senza allocare nuova memoria per il `vector`. Quando abbiamo aggiunto il primo elemento, la dimensione di `v` era diventata 1 e la capacità ugualmente 1. Quando abbiamo aggiunto il secondo elemento, la dimensione di `v` era diventata 2, così come la capacità. Al terzo elemento, la dimensione di `v` diventa 3 e la capacità 4. Se aggiungiamo altri due elementi, la dimensione di v sarà 5 e la capacità 8. La capacità si raddoppia ogni volta che lo spazio totale allocato per il `vector` si riempie e viene aggiunto un altro elemento.

Le linee 26 e 27 mostrano come inviare in output il contenuto di un array utilizzando i puntatori e l'aritmetica dei puntatori. La linea 30 chiama la funzione `printVector()` per visualizzare il contenuto di un `vector` tramite gli iteratori. La definizione del template di funzione `printVector()` inizia alla linea 43. Tale funzione riceve come argomento un riferimento costante a un `vector`. La linea 45

```
vector< T >::const_iterator p1;
```

dichiara un `const_iterator` di nome `p1` che itera attraverso il `vector` e ne visualizza il contenuto. Un `const_iterator` consente al programma di leggere gli elementi del `vector`, ma non ne consente la modifica. Il costrutto `for` alle linee 47 e 48

```
for ( p1 = vec.begin(); p1 != vec.end(); p1++ )
    cout << *p1 << ' ';
```

inizializza `p1` utilizzando la funzione `begin()` membro di `vector`, la quale restituisce un `const_iterator` al primo elemento del `vector` (c'è un'altra versione di `begin()` che restituisce un iterator che può essere utilizzato per i container non const). Il ciclo continua finché `p1` non oltrepassa la fine del `vector`. Questa situazione viene determinata confrontando `p1` con il risultato di `vec.end()`, che restituisce un `const_iterator` (come per `begin()`, c'è un'altra versione di `end()` che restituisce un iteratore) indicante la posizione che si trova dopo l'ultimo elemento del `vector`. Se `p1` è uguale a questo valore, vuol dire che è stata raggiunta la fine del `vector`. Le funzioni `begin()` e `end()` sono disponibili per tutti i container di prima classe. Il corpo del ciclo dereferenzia l'iteratore `p1` per ottenere il valore presente nell'elemento corrente del `vector`. L'espressione `p1++` posiziona l'iteratore sull'elemento successivo del `vector`.

Collaudo 4 *Soltanto gli iteratori ad accesso casuale supportano l'operatore* `<`. *È meglio utilizzare* `!=` *e* `end()` *per determinare se si è raggiunta la fine di un container.*

La linea 34

```
vector< int >::reverse_iterator p2;
```

dichiara un `reverse_iterator` che può essere utilizzato per iterare attraverso un `vector` all'indietro. Tutti i container di prima classe supportano questo tipo di iteratore.

Le linee 36 e 37

```
for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
   cout << *p2 << ' ';
```

utilizzano un costrutto for simile a quello della funzione `printVector()` per iterare attraverso il `vector`. In questo ciclo, le funzioni `rbegin()` (ovvero l'iteratore per il punto di partenza dell'iterazione all'indietro) e `rend()` (ovvero l'iteratore per il punto finale dell'iterazione all'indietro) delineano l'intervallo di elementi da inviare in output al contrario. Come per `begin()` e `end()`, `rbegin()` e `rend()` possono restituire un `const_reverse_iterator` o un `reverse_iterator` a seconda se il container sia costante o meno.

Il programma `codice3.cpp` illustra le funzioni che consentono di recuperare e manipolare gli elementi di un `vector`. La linea 14

```
vector< int > v( a, a + SIZE );
```

utilizza un overloading del costruttore di vector che prende come argomenti due iteratori. Ricordate che si possono utilizzare puntatori ad array come iteratori. Questa istruzione crea il vector di interi `v` e lo inizializza con il contenuto dell'array di interi `a` dalla posizione di `a` fino alla posizione `a+SIZE` esclusa.

```
1 // codice3.cpp
2 // Test delle funzioni di manipolazione degli elementi
3 // del template di classe vector
4 #include <iostream>
5 #include <vector>
6 #include <algorithm>
7
8 using namespace std;
9
```

```
10 int main()
11 {
12    const int SIZE = 6;
13    int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
14    vector< int > v( a, a + SIZE );
15    ostream_iterator< int > output( cout, " " );
16    cout << "Il vector v contiene: ";
17    copy( v.begin(), v.end(), output );
18
19    cout << "\nPrimo elemento di v: " << v.front()
20         << "\nUltimo elemento di v: " << v.back();
21
22    v[ 0 ] = 7;         // imposta a 7 il primo elemento
23    v.at( 2 ) = 10;     // imposta l'elemento di posizione 2 a 10
24    v.insert( v.begin() + 1, 22 );  // inserisce 22 al 2 elemento
25    cout << "\nContenuto di v dopo il cambiamento: ";
26    copy( v.begin(), v.end(), output );
27
28    try {
29       v.at( 100 ) = 777;   // accesso ad un elemento non valido
30    }
31    catch ( out_of_range e ) {
32       cout << "\nEccezione: " << e.what();
33    }
34
35    v.erase( v.begin() );
36    cout << "\ nContenuto di v dopo l'uso di erase(): ";
37    copy( v.begin(), v.end(), output );
38    v.erase( v.begin(), v.end() );
39    cout << "\nDopo erase(), il vector v "
40         << ( v.empty() ? "e'" : "non e'" ) << " vuoto";
41
42    v.insert( v.begin(), a, a + SIZE );
43    cout << "\ nContenuto di v prima di clear(): ";
44    copy( v.begin(), v.end(), output );
45    v.clear(); // clear() usa erase() per svuotare una  collezione
46    cout << "\nDopo clear(), il vector v "
47         << ( v.empty() ? "e'" : "non e'" ) << " vuoto";
48
49    cout << endl;
50    return 0;
51 }
```

Esecuzione:

```
Il vector v contiene: 1 2 3 4 5 6
Primo elemento di v: 1
Ultimo elemento di v: 6
Contenuto di v dopo il cambiamento: 7 22 2 10 4 5 6
```

```
Eccezione: invalid vector<T> subscript
Contenuto di v dopo l'uso di erase(): 22 2 10 4 5 6
Dopo erase(), il vector v e'  vuoto
Contenuto di v prima di clear(): 1 2 3 4 5 6
Dopo clear(), il vector v e'  vuoto
```

La linea 15

```
ostream_iterator< int > output( cout,   );
```

dichiara un `ostream_iterator` di nome `output` che può essere utilizzato per visualizzare interi separati da uno spazio tramite `cout`. Un `ostream_iterator` costituisce un meccanismo di `output` sicuro per i tipi di dato, perché invia in output soltanto i valori di tipo `int` o di tipo compatibile con `int`. Il primo argomento del costruttore specifica lo stream di output e il secondo argomento è una stringa che specifica i caratteri da utilizzare come separatori per i valori inviati in output, in questo caso uno spazio. Utilizzeremo `ostream_iterator` per visualizzare il contenuto del `vector` in questo esempio.

La linea 17

```
copy( v.begin(), v.end(), output );
```

utilizza l'algoritmo `copy()` della STL per inviare l'intero contenuto del vector `v` sull'output standard. L'algoritmo `copy()` copia ogni elemento del container `v` a partire dalla posizione specificata dall'iteratore passato come primo argomento fino alla posizione specificata dall'iteratore passato come secondo argomento: quest'ultima posizione è esclusa. Il primo e il secondo argomento devono soddisfare i requisiti degli iteratori di input, cioè degli iteratori tramite i quali si possono leggere valori da un container. Gli elementi sono copiati alla posizione specificata dall'iteratore di output (cioè un iteratore tramite il quale si può memorizzare o inviare in output un valore) specificato come ultimo argomento. In questo caso, l'iteratore di output è un `ostream_iterator` utilizzato con `cout`, per cui gli elementi sono copiati sull'output standard. Per utilizzare gli algoritmi della STL, dovete includere il file di intestazione `<algorithm>`.

Le linee 19 e 20 utilizzano le funzioni `front()` e `back()` (disponibili per tutti i container sequenziali) che determinano rispettivamente il primo e l'ultimo elemento del `vector`.

Errore Tipico 3 *I risultati delle funzioni `front()` e `back()` sono indefiniti su un oggetto `vector` vuoto.*

Le linee 22 e 23

```
v[ 0 ] = 7;         // imposta a 7 il primo elemento
v.at( 2 ) = 10;     // imposta l'elemento di posizione 2 a 10
```

illustrano due modi per indirizzare un `vector` utilizzando gli indici che si possono anche utilizzare con i container `deque`. La linea 22 utilizza l'operatore di indicizzazione in overloading che restituisce un riferimento al valore della posizione specificata o un riferimento costante a tale valore, a seconda se il container sia costante o meno. La funzione `at()` effettua la stessa operazione effettuando, però, il controllo di validità dell'indice. La funzione `at()` controlla prima che il valore fornitole come argomento si trovi nei limiti del `vector`. Se così non è, `at()` lancia un'eccezione `out_of_bounds` (come mostrano le linee 28-33). La Tabella 16 mostra alcuni tipi di eccezione della STL.

Tipi di eccezione	Descrizione
`out_of_range`	Indica che l'indice non è valido perché è fuori dai limiti del vector: questa eccezione viene tipicamente lanciata dalla funzione `at()`
`invalid_argument`	Indica che una funzione ha ricevuto un argomento non valido
`length_error`	Indica un tentativo di creare un container troppo lungo, e vale anche per le string, ecc.
`bad_alloc`	Indica che il tentativo di allocare memoria di new (o di un allocatore) non è andato a buon fine perché non c'è memoria sufficiente

Tabella 12: Tipi di eccezione della STL

La linea 24

```
v.insert( v.begin() + 1, 22 ); // inserisce 22 al 2 elemento
```

utilizza una delle tre funzioni `insert()` disponibili per tutti i container sequenziali. Questa istruzione inserisce il valore 22 prima dell'elemento che si trova alla posizione specificata dall'iteratore passato come primo argomento. In questo esempio, l'iteratore punta al secondo elemento del `vector`, per cui 22 viene inserito nel secondo elemento e il secondo elemento originario diventa il terzo del `vector`. Le altre versioni di `insert()` consentono di inserire copie multiple dello stesso valore a partire da una particolare posizione del container o di inserire un intervallo di valori provenienti da un altro container (o da un array) a partire da una particolare posizione del container origine.

Le linee 35 e 38

```
v.erase( v.begin() );
v.erase( v.begin(), v.end() );
```

utilizzano le due funzioni `erase()` disponibili per tutti i container di prima classe. La linea 35 indica che l'elemento che si trova alla posizione specificata dall'argomento iteratore deve essere eliminato dal container (in questo esempio l'elemento iniziale del `vector`). La linea 38 specifica che devono essere eliminati dal container tutti gli elementi nell'intervallo che va dalla posizione specificata dal primo argomento fino a quella del secondo argomento (esclusa). In questo esempio, dal `vector` sono eliminati tutti gli elementi. La linea 40 utilizza la funzione `empty()` (disponibili per tutti i container e gli adattatori) per avere la conferma che il `vector` sia vuoto.

Errore Tipico 4 *Se si elimina un elemento che contiene un puntatore a un oggetto allocato dinamicamente non si elimina anche tale oggetto.*

La linea 42

```
v.insert( v.begin(), a, a + SIZE );
```

utilizza la versione di `insert()` che indica di inserire nel vector una sequenza di valori compresi tra la posizione iniziale specificata dal secondo argomento e quella finale specificata dal terzo: i valori possono provenire anche da un altro container, ma in questo caso provengono dall'array di interi `a`. Ricordate che la posizione finale specifica la posizione della sequenza dopo l'ultimo elemento da inserire: la copia non include anche quest'ultima posizione.

Infine, la linea 45

```
v.clear();
```

utilizza la finzione `clear()` (disponibili per tutti i container di prima classe) per svuotare il `vector`. Questa funzione chiama in realtà la versione di `erase()` utilizzata nella linea 38 per effettuare l'operazione.

Nota: Ci sono altre funzioni di cui non abbiamo parlato che sono comuni a tutti i container o a tutti i container sequenziali. Ne parleremo nelle prossime sezioni. Parleremo anche delle diverse funzioni specifiche di ciascun container.

2.2 Il container sequenziale `list`

Il container sequenziale `list` fornisce un'implementazione efficiente delle operazioni di inserimento ed eliminazione in qualsiasi posizione del container. Se la maggior parte di queste operazione si verifica agli estremi del container, conviene però utilizzare l'implementazione più efficiente di `deque` (sezione 2.3). La classe `list` è implementata come lista concatenata doppia, vale a dire ogni nodo della `list` contiene un puntatore al nodo precedente e uno al nodo successivo. Ciò consente alla classe `list` di supportare gli iteratori bidirezionali, per poter attraversare il container nei due sensi. Ogni algoritmo che richiede iteratori di input, output, forward o bidirezionali può operare su una `list`. Molte delle funzioni membro di `list` manipolano gli elementi del container come un insieme ordinato di elementi. Oltre alle funzioni membro dei container della STL in Tabella 2 e alle funzioni membro di tutti i container sequenziali discussi nella sezione 5, la classe `list` fornisce altre funzioni membro: `splice()`, `push_front()`, `remove()`, `inique()`, `merge()`, `reverse()` e `sort()`. Il programma `codice4.cpp` illustra diverse caratteristiche della classe `list`. Ricordate che molte delle funzioni presentate in `codice2.cpp` e `codice3.cpp` possono essere utilizzate con la classe `list`. Per utilizzare la classe `list` occorre includere il file di intestazione `<list>`.

```
 1 // codice4.cpp
 2 // Test della classe list
 3 #include <iostream>
 4 #include <list>
 5 #include <algorithm>
 6
 7 using namespace std;
 8
 9 template < class T >
10 void printList( const list< T > &listRef );
11
12 int main()
13 {
14    const int SIZE = 4;
15    int a[ SIZE ] = { 2, 6, 4, 8 };
16    list< int > values, otherValues;
17
18    values.push_front( 1 );
19    values.push_front( 2 );
20    values.push_back( 4 );
21    values.push_back( 3 );
22
23    cout << "values contiene: ";
```

```
24    printList( values );
25    values.sort();
26    cout << "\nvalues dopo sort() contiene: ";
27    printList( values );
28
29    otherValues.insert( otherValues.begin(), a, a + SIZE );
30    cout << "\notherValues contiene: ";
31    printList( otherValues );
32    values.splice( values.end(), otherValues );
33    cout << "\nDopo splice() values contiene: ";
34    printList( values );
35
36    values.sort();
37    cout << "\nvalues contiene: ";
38    printList( values );
39    otherValues.insert( otherValues.begin(), a, a + SIZE );
40    otherValues.sort();
41    cout << "\notherValues contiene: ";
42    printList( otherValues );
43    values.merge( otherValues );
44    cout << "\nDopo merge():\n   values contiene: ";
45    printList( values );
46    cout << "\n   otherValues contiene: ";
47    printList( otherValues );
48
49    values.pop_front();
50    values.pop_back();   // per tutti i container sequenziali
51    cout << "\nDopo pop_front() e pop_back() values contiene:\n";
52    printList( values );
53
54    values.unique();
55    cout << "\nDopo unique() values contiene: ";
56    printList( values );
57
58    // il metodo swap() e' disponibile in tutti i container
59    values.swap( otherValues );
60    cout << "\nDopo swap:\n values contiene: ";
61    printList( values );
62    cout << "\n   otherValues contiene: ";
63    printList( otherValues );
64
65    values.assign( otherValues.begin(), otherValues.end() );
66    cout << "\nDopo assign() values contiene: ";
67    printList( values );
68
69    values.merge( otherValues );
70    cout << "\nvalues contiene: ";
```

```
71    printList( values );
72    values.remove( 4 );
73    cout << "\nDopo remove( 4 ) values contiene: ";
74    printList( values );
75    cout << endl;
76    return 0;
77 }
78
79 template < class T >
80 void printList( const list< T > &listRef )
81 {
82    if ( listRef.empty() )
83       cout << "La list e' vuota";
84    else {
85       ostream_iterator< T > output( cout, " " );
86       copy( listRef.begin(), listRef.end(), output );
87    }
88 }
```

Esecuzione:

```
values contiene: 2 1 4 3
values dopo sort() contiene: 1 2 3 4
otherValues contiene: 2 6 4 8
Dopo splice() values contiene: 1 2 3 4 2 6 4 8
values contiene: 1 2 2 3 4 4 6 8
otherValues contiene: 2 6 4 8
Dopo merge():
    values contiene: 1 2 2 2 3 4 4 4 6 6 8 8
    otherValues contiene: La list e' vuota
Dopo pop_front() e pop_back() values contiene: 2 3 4 6 8
Dopo swap(): values contiene: La list e' vuota
    otherValues contiene: 2 3 4 6 8
Dopo assign() values contiene: 2 3 4 6 8
values contiene: 2 2 3 3 4 4 6 6 8 8
Dopo remove( 4 ) values contiene: 2 2 3 3 6 6 8 8
```

La linea 16

```
list< int > values, otherValues;
```

istanzia due oggetti `list` in grado di memorizzare interi. Le linee 18 e 19 utilizzano la funzione `push_front()` per inserire degli interi all'inizio di `values`. La funzione `push_front()` è specifica delle classi `list` e `deque` (e non di `vector`). Le linee 20 e 21 utilizzano la funzione `push_back()` per inserire degli interi alla fine di `values`. Ricordate che la funzione `push_back()` è comune a tutti i container sequenziali.

La linea 25

```
values.sort();
```

utilizza la funzione `sort()` membro di `list` per porre gli elementi della list in ordine crescente[2]. C'è una seconda versione di `sort()` che consente di specificare una funzione predicativa binaria che prende due argomenti (i valori nella `list`), effettua un confronto e restituisce un valore `bool` che indica il risultato. Questa funzione determina l'ordine in cui sono ordinati gli elementi della `list`. Questa versione può risultare particolarmente utile per una `list` che memorizza puntatori anziché valori[3].

La linea 32

```
values.splice( values.end(), otherValues );
```

utilizza la funzione `splice()` per eliminare gli elementi in `otherValues` e inserirli in `values` prima della posizione specificata dall'iteratore passato come primo argomento. Ci sono altre due versioni di questa funzione. La funzione `splice()` con tre argomenti consente di eliminare un elemento dal container specificato come secondo argomento dalla posizione specificata dall'iteratore passato come terzo argomento. La funzione `splice()` con quattro argomenti utilizza gli ultimi due argomenti per specificare l'intervallo di posizioni da togliere dal container specificato come secondo argomento e da porre alla posizione specificata dal primo argomento.

Dopo aver inserito altri elementi in `otherValues` e aver ordinato `values` e `otherValues`, la linea 43

```
values.merge( otherValues );
```

utilizza la funzione `merge()` membro di `list` per rimuovere tutti gli elementi di `otherValues` e per inserirli in sequenza ordinata in `values`. Entrambe le `list` devono essere ordinate con lo stesso criterio prima che venga effettuata questa operazione. Una seconda versione di `merge()` consente di fornire una funzione predicativa che prende due argomenti (i valori nella `list`) e restituisce un valore `bool`. La funzione predicativa specifica il criterio di ordinamento utilizzato da `merge()`.

La linea 49 utilizza la funzione `pop_front()` per eliminare il primo elemento nella `list`. La linea 50 utilizza la funzione `pop_back()` (disponibile per tutti i container sequenziali) per eliminare l'ultimo elemento della `list`.

La linea 54

```
values.unique();
```

utilizza la funzione `unique()` per eliminare gli elementi duplicati presenti nella `list`. La `list` dovrebbe essere ordinata (in modo che tutti i duplicati siano contigui) prima che sia effettuata questa operazione, per garantire che siano eliminati tutti i duplicati. Una seconda versione di `unique()` consente di fornire una funzione predicativa che prende due argomenti (i valori nella `list`) e restituisce un valore `bool`. La funzione predicativa specifica se due elementi sono uguali.

La linea 59

```
values.swap( otherValues );
```

utilizza la funzione `swap()` (disponibile per tutti i container) per scambiare il contenuto di `values` con quello di `otherVa1ues`.

La linea 65

[2]Questa operazione è differente da `sort()` negli algoritmi della STL.

[3]In `codice12.cpp` mostriamo una funzione predicativa unaria. Una funzione di questo tipo prende un solo argomento, effettua un confronto utilizzando tale argomento e restituisce un valore bool che indica il risultato.

```
values.assign( otherValues.begin(), otherValues.end() );
```

utilizza la funzione `assign()` per sostituire il contenuto di `values` con quello di `otherValues` nell'intervallo di elementi specificato dai due argomenti iteratore. Una seconda versione di `assign()` sostituisce il contenuto originario con copie dei valori specificati nel secondo argomento. Il primo argomento della funzione specifica il numero di copie.

La linea 72

```
values.remove( 4 );
```

utilizza la funzione `remove()` per eliminare tutte le copie del valore 4 dalla `list`.

2.3 Il container sequenziale `deque`

La classe `deque` fornisce diversi dei pregi di un `vector` e di una `list` in un unico container. Il termine `deque` è una forma abbreviata per "double-ended queue", coda a due estremi. La classe `deque` è implementata per fornire un accesso indicizzato efficiente per la lettura e la modifica dei suoi elementi, un po' come un `vector`. La classe `deque` è anche implementata per effettuare operazioni di inserimento ed eliminazione efficienti per entrambi gli estremi, come una `list` (anche se una `list` può anche effettuare inserimenti/eliminazioni efficienti nelle posizioni intermedie). La classe `deque` supporta gli iteratori ad accesso casuale, per cui i `deque` si possono utilizzare con tutti gli algoritmi della STL. Uno degli usi più comuni di un `deque` consiste nel mantenere una coda di elementi FIFO (il primo inserito è anche il primo estratto).

È prevista l'allocazione di ulteriore spazio per un `deque` ad entrambi gli estremi in blocchi di memoria che sono mantenuti tipicamente da array di puntatori. Dato che la disposizione in memoria di un `deque` non è contigua, i suoi iteratori devono essere più "intelligenti" dei puntatori che attraversano i `vector` o gli array.

Efficienza 12 *Quando viene allocato un blocco di memoria per un* ***`deque`****, in diverse implementazioni tale blocco non viene deallocato finché il* ***`deque`*** *non viene distrutto. Ciò ottimizza le operazioni di un* ***`deque`*** *rispetto alle implementazioni in cui viene ripetutamente allocato, deallocato e riallocato. Ma ciò significa anche che una* ***`deque`*** *utilizzerà la memoria in modo meno efficiente rispetto, ad esempio, ad un* ***`vector`****.*

Efficienza 13 *Inserimenti/eliminazioni nelle posizioni intermedie di un* ***`deque`*** *sono ottimizzati per minimizzare il numero di elementi copiati, mantenendo la parvenza che gli elementi siano contigui.*

La classe `deque` fornisce le stesse operazioni di base della classe `vector`, e vi aggiunge le funzioni membro `push_front()` e `pop_front()` rispettivamente per l'inserimento e l'eliminazione all'inizio del `deque`.

In `codice5.cpp` vengono mostrate le caratteristiche della classe `deque`. Ricordate che molte delle funzioni presentate in `codice2.cpp`, `codice3.cpp` e `codice4.cpp` possono essere utilizzate con la classe `deque`. Per utilizzare la classe `deque` bisogna includere il file di intestazione `<deque>`.

La linea 11

```
deque< double > values;
```

istanzia un `deque` che può memorizzare valori double. Le linee 14-16 utilizzano le funzioni `push_front()` e `push_back()` per inserire degli elementi all'inizio e alla fine del `deque`. Ricordate che `push_back()` è disponibile per tutti i container sequenziali, mentre `push_front()` è disponibile soltanto per le classi `list` e `deque`.

```
 1 // codice5.cpp
 2 // Test della classe deque
 3 #include <iostream>
 4 #include <deque>
 5 #include <algorithm>
 6
 7 using namespace std;
 8
 9 int main()
10 {
11   deque< double > values;
12   ostream_iterator< double > output( cout, " " );
13
14   values.push_front( 2.2 );
15   values.push_front( 3.5 );
16   values.push_back( 1.1 );
17
18   cout << "values contiene: ";
19
20   for ( int i = 0; i < values.size(); ++i )
21      cout << values[ i ] << ' ';
22
23   values.pop_front();
24   cout << "\nDopo pop_front() values contiene: ";
25   copy ( values.begin(), values.end(), output );
26
27   values[ 1 ] = 5.4;
28   cout << "\nDopo values[ 1 ] = 5.4, values contiene: ";
29   copy ( values.begin(), values.end(), output );
30   cout << endl;
31   return 0;
32 }
```

Esecuzione:

```
values contiene: 3.5 2.2 1.1
Dopo pop_front() values contiene: 2.2 1.1
Dopo values[ 1 ] = 5.4, values contiene: 2.2 5.4
```

Il costrutto `for` alla linea 20

```
for ( int i = 0; i < values.size(); ++i )
   cout << values[ i ] << ' ';
```

utilizza l'operatore di indicizzazione per recuperare il valore di ogni elemento del `deque` e inviarlo in output. Notare l'uso della funzione `size()` nella condizione, per evitare di accedere a un elemento che si trovi oltre i limiti del `deque`. La linea 23 utilizza la funzione `pop_front()` per eliminare il primo elemento del `deque`. Ricordate che la funzione `pop_front()` è disponibile soltanto per le classi `list` e `deque` (non per la classe `vector`).

La linea 27

```
values[ 1 ] = 5.4;
```

utilizza l'operatore di indicizzazione per creare un lvalue. Ciò consente di assegnare direttamente i valori a ogni elemento del `deque`.

3 I container associativi

I container associativi della STL sono stati progettati per fornire un accesso diretto nelle operazioni di memorizzazione e di recupero di elementi tramite chiavi di ricerca. I quattro container associativi sono `multiset`, `set`, `multimap` e `map`. In ogni container le chiavi sono mantenute ordinate. L'attraversamento di un container associativo avviene secondo l'ordinamento previsto su tale container. Le classi `multiset` e `set` forniscono operazioni per la manipolazione di insiemi di valori in cui i valori sono le chiavi, ovvero non c'è separazione tra valori e chiavi associate. La principale differenza tra un `multiset` e un `set` è che un `multiset` permette chiavi duplicate mentre un `set` no. Le classi `multimap` e `map` forniscono operazioni per la manipolazione dei valori associati alle chiavi. La principale differenza tra un `multimap` e un `map` è che un `multimap` permette chiavi duplicate con valori associati e un `map` permette soltanto chiavi univoche. Oltre alle funzioni membro comuni a tutti i container presentati in Tabella 2, tutti i container associativi supportano anche molte altre funzioni membro, tra cui `find()`, `lower_bound()`, `upper_bound()` e `count()`. Nelle sezioni seguenti mostriamo degli esempi per ogni container associativo e per le funzioni membro comuni.

3.1 Il container associativo `multiset`

Il container associativo `multiset` (multinsieme) serve per la memorizzazione e il recupero veloci di dati permettendo la presenza di chiavi duplicate. L'ordinamento degli elementi è determinato da un oggetto funzione comparatrice. Per esempio, in un multinsieme di interi gli elementi possono essere posti in ordine crescente ordinando le chiavi per mezzo dell'oggetto funzione `less<int>`[4]. Il tipo di dato delle chiavi in tutti i container associativi deve supportare il confronto in maniera corretta sulla base dell'oggetto funzione comparatrice specificato: le chiavi ordinate con `less<int>` devono supportare il confronto con la funzione `operator<`. Se le chiavi utilizzate nei container associativi sono tipi di dati definiti dal programmatore, essi devono fornire gli operatori appropriati di confronto (vedi la nota Ingegneria del Software 3). I multiset supportano iteratori bidirezionali, ma non ad accesso casuale.

Efficienza 14 *Per motivi di efficienza i* `multiset` *e i* `set` *sono implementati tipicamente con i cosiddetti* red-black trees *(alberi rosso-nero) un tipo particolare di alberi di binari di ricerca che tendono ad essere bilanciati, minimizzando cosi tempi medi di ricerca.*

Il programma in `codice6.cpp` mostra un `multiset` di interi che si trovano in ordine crescente. Per utilizzare la classe `multiset` dovete includere il file `<set>`. I container `multiset` e `set` forniscono le stesse funzioni membro.

```
1 // codice6.cpp
2 // Test della classe multiset
```

[4] Vedi sezione 7

```
 3 #include <iostream>
 4 #include <set>
 5 #include <algorithm>
 6
 7 using namespace std;
 8
 9 int main()
10 {
11    const int SIZE = 10;
12    int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
13    typedef multiset< int, less< int > > ims;
14    ims intMultiset;    // ims sta per "integer multiset"
15    ostream_iterator< int > output( cout, " " );
16
17    cout << "Ora ci sono " << intMultiset.count( 15 )
18         << " occorrenze di 15 nel multiset\n";
19    intMultiset.insert( 15 );
20    intMultiset.insert( 15 );
21    cout << "Dopo inserts(), ci sono "
22         << intMultiset.count( 15 )
23         << " occorrenze di 15 nel multiset\n";
24
25    ims::const_iterator result;
26
27    result = intMultiset.find( 15 );  // find rende un iteratore
28
29    if ( result != intMultiset.end() ) // se l'iteratore non e' alla fine
30       cout << "Trovato il valore 15\n";     // e' stato trovato 15
31
32    result = intMultiset.find( 20 );
33
34    if ( result == intMultiset.end() )    // sara' quindi vero
35       cout << "Non trovato il valore 20\n"; // 20 non trovato
36
37    intMultiset.insert( a, a + SIZE ); // aggiunge array a al multiset
38    cout << "Dopo insert(), intMultiset contiene:\n";
39    copy( intMultiset.begin(), intMultiset.end(), output );
40
41    cout << "\nLower bound di 22: "
42         << *( intMultiset.lower_bound( 22 ) );
43    cout << "\nUpper bound di 22: "
44         << *( intMultiset.upper_bound( 22 ) );
45
46    pair< ims::const_iterator, ims::const_iterator > p;
47
48    p = intMultiset.equal_range( 22 );
49    cout << "\nUsando equal_range( 22 )"
```

```
50         << "\n   Lower bound: " << *( p.first )
51         << "\n   Upper bound: " << *( p.second );
52    cout << endl;
53    return 0;
54 }
```

ESECUZIONE:

```
Ora ci sono 0 occorrenze di 15 nel multiset
Dopo inserts(), ci sono 2 occorrenze di 15 nel multiset
Trovato il valore 15
Non trovato il valore 20
Dopo insert(), intMultiset contiene: 1 7 9 13 15 15 18 22 22 30 85 100
Lower bound di 22: 22
Upper bound di 22: 30
Usando equal_range( 22 ):
    Lower bound: 22
    Upper bound: 30
```

Le linee 13 e 14

```
typedef multiset< int, less< int > > ims;
ims intMultiset;    // ims sta per "integer multiset"
```

creano con typedef un nuovo tipo di dato corrispondente a un multinsieme di interi posti in ordine crescente utilizzando l'oggetto funzione `less<int>`. Questo nuovo tipo viene poi utilizzato per stanziare un oggetto `multiset` di interi, `intMultiset`.

Buona Abitudine 1 *Utilizzate typedef per semplificare il codice che contiene nomi lunghi (come `multiset`).*

L'istruzione di output di linea 17

```
cout << "Ora ci sono " << intMultiset.count( 15 )
     << " occorrenze di 15 nel multiset\n";
```

utilizza la funzione `count()` (disponibile per tutti i container associativi) per contare il numero di occorrenze del valore 15 presenti correntemente nel `multiset`.

Le linee 19 e 20

```
intMultiset.insert( 15 );
intMultiset.insert( 15 );
```

utilizzano una delle tre versioni di `insert()` per aggiungere il valore 15 al `multiset`. Una seconda versione di `insert()` prende come argomenti un iteratore e un valore e inizia la ricerca del punto di inserimento dalla posizione specificata dall'iteratore. Una terza versione di `insert()` prende come argomenti due iteratori che specificano un intervallo di valori provenienti da un altro container da aggiungere al `multiset`.

La linea 27

```
result = intMultiset.find( 15 ); // find rende un iteratore
```

utilizza la funzione `find()` (disponibile per tutti i container associativi) per localizzare il valore 15 nel `multiset`. La funzione `find()` restituisce un `iterator` o un `const_iterator` che punta alla prima posizione in cui è stato trovato il valore cercato. Se il valore non viene trovato, `find()` restituisce un `iterator` o un `const_iterator` uguale al valore restituito dalla funzione `end()`.

La linea 37

```
intMultiset.insert( a, a + SIZE ); // aggiunge l'array a al multiset
```

utilizza la funzione `insert()` per inserire gli elementi dell'array `a` nel `multiset`. Alla linea 39, l'algoritmo `copy()` copia gli elementi del `multiset` sull'output standard. Notate che gli elementi sono visualizzati in ordine crescente.

Le linee 41-44

```
cout << "\nLower bound di 22: "
     << *( intMultiset.lower_bound( 22 ) );
cout << "\nUpper bound di 22: "
     << *( intMultiset.upper_bound( 22 ) );
```

utilizzano le funzioni disponibili in tutti i container associativi `lower_bound()` (che sta per limite inferiore) e `upper_bound()` (che sta per limite superiore) per determinare la posizione della prima occorrenza del valore 22 nel `multiset` e la posizione dell'elemento che si trova dopo l'ultima occorrenza di 22 nel `multiset`. Entrambe le funzioni restituiscono degli `iterator` o `const_iterator` che puntano alle posizioni appropriate o l'iteratore restituito da `end()` se il valore non è presente nel `multiset`.

La linea 46

```
pair< ims::const_iterator, ims::const_iterator > p;
```

crea un'istanza della classe `pair` di nome `p`. Gli oggetti della classe `pair` servono a manipolare coppie di valori. In questo esempio, il contenuto di un `pair` sono due `const_iterator` per il nostro `multiset` di interi. Lo scopo di `p` è memorizzare il valore restituito dalla funzione `equal_range()` di `multiset`, che restituisce un `pair` contenente i risultati delle operazioni `lower_bound()` e `upper_bound()`. Il tipo `pair` contiene due dati membro public di nome `first` e `second`.

La linea 48

```
p = intMultiset.equal_range( 22 );
```

utilizza la lunzione `equal_range()` per determinare il lower bound e l'upper bound di 22 nel `multiset`. La linee 50 e 51 utilizzano `p.first()` e `p.second()` rispettivamente per accedere a `lower_bound` e `upper_bound`. Abbiamo dereferenziato gli iteratori per visualizzare i valori presenti alle posizioni restituite da `equal_range()`.

3.2 Il container associativo `set`

Il container associativo `set` è utile per la memorizzazione e il recupero efficiente di chiavi univoche. L'implementazione di un `set` è identica a quella di un `multiset` eccetto il fatto che un `set` deve avere chiavi uniche, perciò se si tenta di inserire un duplicato di una chiave in un `set`, tale duplicato

viene ignorato. Infatti, `set` è l'implementazione del concetto matematico di insieme, ed è questo il comportamento degli insiemi in matematica, per cui non identificheremo questo comportamento come errore di programmazione. Un `set` supporta iteratori bidirezionali, ma non ad accesso casuale. Il programma in `codice7.cpp` mostra un `set` di `double`. Per utilizzare la classe `set` bisogna includere il file di intestazione `<set>`.

```
 1 // codice7.cpp
 2 // Test della classe set
 3 #include <iostream>
 4 #include <set>
 5 #include <algorithm>
 6
 7 using namespace std;
 8
 9 int main()
10 {
11    typedef set< double, less< double > > double_set;
12    const int SIZE = 5;
13    double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
14    double_set doubleSet( a, a + SIZE );
15    ostream_iterator< double > output( cout, " " );
16
17    cout << "doubleSet contiene: ";
18    copy( doubleSet.begin(), doubleSet.end(), output );
19
20    pair< double_set::const_iterator, bool > p;
21
22    p = doubleSet.insert( 13.8 ); // valore non presente nel set
23    cout << '\n' << *( p.first )
24        << ( p.second ? " e'" : " non e'" ) << " inserito";
25    cout << "\ndoubleSet contiene: ";
26    copy( doubleSet.begin(), doubleSet.end(), output );
27
28    p = doubleSet.insert( 9.5 );  // valore gia' nel set
29    cout << '\n' << *( p.first )
30         << ( p.second ? " e'" : " non e'" ) << " inserito";
31    cout << "\ndoubleSet contiene: ";
32    copy( doubleSet.begin(), doubleSet.end(), output );
33
34    cout << endl;
35    return 0;
36 }
```

Esecuzione:

```
DoubleSet contiene: 2.1 3.7 4.2 9.5
13.8 e' inserito
DoubleSet contiene: 2.1 3.7 4.2 9.5 13.8
9.5 non e' inserito
```

```
DoubleSet contiene: 2.1 3.7 4.2 9.5 13.8
```

La linea 11

```
typedef set< double, less< double > > double_set;
```

utilizza typedef per creare un nuovo tipo di dato per un `set` di `double` posti in ordine crescente utilizzando l'oggetto funzione `less<double>`. La linea 14

```
double_set doubleSet( a, a + SIZE );
```

utilizza il nuovo tipo `double_set` per istanziare l'oggetto `doubleSet`. Il costruttore prende gli elementi nell'array `a` tra `a` e `a + SIZE` (ovvero tutto l'array) e li inserisce nel `set`. La linea 18 utilizza l'algoritmo `copy()` per visualizzare il contenuto dell'insieme. Osservate che il valore 2.1, che compare due volte nell'array `a`, compare soltanto una volta in `doubleSet`. Ciò accade perché il container `set` non permette chiavi duplicate.

La linea 20

```
pair< double_set::const_iterator, bool > p;
```

dichiara una coppia (in inglese pair) che consiste in un `const_iterator` per un `double_set` e un valore `bool`. Questo oggetto memorizza il risultato di una chiamata alla funzione `insert()` di `set`.

La linea 22

```
p = doubleSet.insert( 13.8 ); // valore non presente nel set
```

utilizza la funzione `insert()` per porre nell'insieme il valore 13.8. L'oggetto `pair` restituito, `p`, contiene un iteratore `p.first()` che punta al valore 13.8 nel `set` e un valore `bool` che è true se il valore è stato effettivamente inserito e false in caso contrario (era già presente nel `set`).

3.3 Il container associativo `multimap`

Il container associativo `multimap` si utilizza per la memorizzazione e il recupero efficienti di valori associati ad una chiave (che prendono anche il nome di coppie chiave/valore). Molti dei metodi utilizzati con i `multiset` e i `set` sono comuni anche a `multimap` e `map`. Gli elementi dei `multimap` e dei `map` sono coppie di chiavi e valori, anziché valori individuali. Quando si effettua un inserimento in un `multimap` o in un `map`, si utilizza un oggetto `pair` che contiene la chiave e il valore. L'ordinamento delle chiavi è determinato dalla funzione comparatrice. Per esempio, in un `multimap` che utilizza gli interi come tipo di chiave, le chiavi possono essere poste in ordine crescente tramite l'oggetto funzione comparatrice `less<int>`. In un `multimap` sono permesse chiavi duplicate, per cui a una singola chiave si possono associare più valori. Questo è un esempio di relazione uno-a-molti. Per esempio, in un sistema di elaborazione delle transazioni su carta di credito, un conto relativo a una sola carta di credito può essere associato a molte transazioni; in un università, uno studente può iscriversi a molti corsi e un professore può insegnare a molti studenti; nelle forze armate, un grado (come "soldato semplice") si riferisce a molte persone. Un `multimap` supporta iteratori bidirezionali, ma non ad accesso casuale. Come per i `multiset` e i `set`, i `multimap` sono tipicamente implementati con alberi di ricerca binaria rosso/nero, in cui i nodi dell'albero sono le coppie chiave/valore. Il programma in `codice8.cpp` mostra l'utilizzo del container associativo `multimap`. Per utilizzare la classe `multimap` bisogna includere il file di intestazione `<map>`.

Efficienza 15 *Un* `multimap` *è implementato per localizzare in modo efficiente tutti i valori associati a una data chiave.*

```
 1 // codice8.cpp
 2 // Test della classe multimap
 3 #include <iostream>
 4 #include <map>
 5
 6 using namespace std;
 7
 8 int main()
 9 {
10    typedef multimap< int, double, less< int > > mmid;
11    mmid pairs;
12
13    cout << "Ora ci sono " << pairs.count( 15 )
14         << " coppie con chiave 15 nel multimap\n";
15    pairs.insert( mmid::value_type( 15, 2.7 ) );
16    pairs.insert( mmid::value_type( 15, 99.3 ) );
17    cout << "Dopo insert(), ci sono "
18         << pairs.count( 15 )
19         << " coppie con chiave 15\n";
20    pairs.insert( mmid::value_type( 30, 111.11 ) );
21    pairs.insert( mmid::value_type( 10, 22.22 ) );
22    pairs.insert( mmid::value_type( 25, 33.333 ) );
23    pairs.insert( mmid::value_type( 20, 9.345 ) );
24    pairs.insert( mmid::value_type( 5, 77.54 ) );
25    cout << "La multimap contiene:\nKey\tValue\n";
26
27    for ( mmid::const_iterator iter = pairs.begin();
28          iter != pairs.end(); ++iter )
29       cout << iter->first << '\t'
30            << iter->second << '\n';
31
32   cout << endl;
33   return 0;
34 }
```

Esecuzione:

```
Ora ci sono 0 coppie con chiave 15 nel multimap
Dopo insert(), ci sono 2 coppie con chiave 15
La multimap contiene:
Key   Value
5     77.54
10    22.22
15    2.7
15    99.3
20    9.345
```

```
25    33.333
30    111.11
```

La linea 10

```
typedef multimap< int, double, less< int > > mmid;
```

utilizza `typedef` per definire un nuovo tipo `multimap` in cui il tipo della chiave è `int`, il tipo del valore associato è `double` e gli elementi sono posti in ordine crescente. La linea 11 utilizza il nuovo tipo per istanziare un `multimap` di nome `pairs`.

L'istruzione di linea 13

```
cout << "Ora ci sono " << pairs.count( 15 )
     << " coppie con chiave 15 nel multimap\n";
```

utilizza la funzione `count()` per determinare il numero di coppie chiave/valore per la chiave 15.

La linea 15

```
pairs.insert( mmid::value_type( 15, 2.7 ) );
```

utilizza la funzione `insert()` per aggiungere una nuova coppia chiave/valore al `multimap`. L' espressione `mmid::value_type( 15, 2.7 )` crea un oggetto `pair` in cui `first` è la chiave (15) di tipo `int` e `second` è il valore (2.7) di tipo `double`. Il tipo `mmid::value_type` è definito alla linea 10 come parte del typedef per il `multimap`.

Il costrutto `for` di linea 27 visualizza il contenuto del `multimap`, incluse le chiavi e i valori. Le linee 29 e 30

```
cout << iter->first << '\t'
     << iter->second << '\n';
```

utilizzano il `const_iterator` di nome `iter` per accedere ai membri del `pair` in ogni elemento del `multimap`. Osservate nell'output che le chiavi sono in ordine crescente.

3.4 Il container associativo `map`

Il container associativo `map` (mappa) è utilizzato per la memorizzazione e il recupero veloci di chiavi uniche e di valori associati. Una mappa non permette chiavi duplicate, per cui si può associare un solo valore a ogni chiave. Si tratta della cosiddetta relazione uno-a-uno. Per esempio, un'azienda che utilizza numeri identificativi unici per i dipendenti come 100, 200 e 300 può porli in un container `map` che associa il numero del dipendente con il suo numero telefonico interno (ad esempio 4321, 4115 e 5217, rispettivamente). Con un `map` si specifica la chiave e si ottiene velocemente il dato associato. Un `map` è chiamato comunemente array associativo: fornendo la chiave nell'operatore di indicizzazione `[ ]` di `map`, si localizza il valore associato alla chiave nella mappa. Le operazioni di inserimento/eliminazione possono essere effettuate ovunque in un `map`.

Il programma in `codice9.cpp` mostra l'uso del container associativo `map`. Questo programma utilizza le stesse caratteristiche di quello in `codice8.cpp` eccetto l'operatore di indicizzazione. Per utilizzare la classe `map` bisogna includere il file di intestazione `<map>`. Le linee 29 e 30

```
pairs[ 25 ] = 9999.99;   // cambia il valore esistente per 25
pairs[ 46 ] = 8765.43;   // inserisce un nuovo valore per 46
```

utilizzano l'operatore di indicizzazione della classe `map`. Se l'indice è una chiave già presente nel `map`, l'operatore restituisce un riferimento al valore associato. Altrimenti l'operatore inserisce la chiave nel `map` e restituisce un riferimento che può essere utilizzata per associare un valore a tale chiave. La linea 29 sostituisce il valore per la chiave 25 (che valeva precedentemente 33.333 come specifica la linea 17) con il nuovo valore di 9999.99. La linea 30 inserisce una nuova coppia chiave/valore nel `map`: si dice che si crea un'associazione.

```
1 codice9.cpp
2   Test della classe map
3   #include <iostream>
4   #include <map>
5
6   using namespace std;
7
8   int main()
9   {
10    typedef map< int, double, less< int > > mid;
11    mid pairs;
12
13    pairs.insert( mid::value_type( 15, 2.7 ) );
14    pairs.insert( mid::value_type( 36, 111.11 ) )
15    pairs.insert( mid::value_type( 5, 1010.1 ) );
16    pairs.insert( mid::value_type( 16, 22.22 ) );
17    pairs.insert( mid::value_type( 25, 33.333 ) );
18    pairs.insert( mid::value_type( 5, 77.54 ) ); // duplicato ignorato
19    pairs.insert( mid::value_type( 26, 9.345 ) );
20    pairs.insert( mid::value_type( 15, 99.3 ) ); // duplicato ignorato
21    cout << "pairs contiene:\nKey\tvalue\n";
22
23    mid::const_iterator iter;
24
25    for ( iter = pairs.begin(); iter pairs.end(); ++iter )
26       cout << iter->first '\t'
27            << iter->second '\n';
28
29    pairs[ 25 ] = 9999.99;   // cambia il valore esistente per 25
30    pairs[ 46 ] = 8765.43;   // inserisce un nuovo valore per 46
31    cout << "\nDopo pairs[ ], pairs contiene:"
32         << "\nKey\tvalue\n";
33
34    for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
35       cout << iter->first '\t'
36            << iter->second '\n';
37
38    cout << endl;
39    return 0;
40  }
```

Esecuzione:

```
pairs contiene:
Key   Value
5     1010.1
10    22.22
15    2.7
20    9.345
25    33.333
30    111.11

Dopo pairs[ ], pairs contiene:
Key   Value
5     1010.1
10    22.22
15    2.7
20    9.345
25    9999.99
30    111.11
40    8765.43
```

4 Adattatori di container

La STL fornisce tre adattatori di container: `stack`, `queue` e `priority_queue`. Gli adattatori non sono container di prima classe: essi non forniscono l'implementazione effettiva di una struttura dati in cui memorizzare elementi, perché non supportano gli iteratori. Il questa dispensa non analizziamo gli adattatori di container.

5 Gli algoritmi

Fino all'avvento della STL, le librerie di container e di algoritmi sviluppate da rivenditori diversi erano sostanzialmente incompatibili. Le prime librerie di container utilizzavano generalmente l'ereditarietà e il polimorfismo, con l'iperutilizzo di risorse tipico delle chiamate di funzioni virtuali. Le prime librerie includevano gli algoritmi nelle classi container come comportamenti di tali classi. La STL separa gli algoritmi dai container. Ciò semplifica notevolmente l'aggiunta di nuovi algoritmi. La STL è stata implementata con un'attenzione particolare all'efficienza. Perciò evita lo spreco di risorse tipico delle chiamate di funzioni virtuali. Con la STL, si può accedere agli elementi dei container tramite gli iteratori.

Ingegneria del Software 8 *Gli algoritmi della STL non dipendono dai dettagli di implementazione dei container su cui operano. Se gli iteratori del container (o dell'array) soddisfano i requisiti dell'algoritmo, esso può funzionare allo stesso modo su un array in vecchio stile C, basato sui puntatori, e su un container della STL (e su qualsiasi struttura dati definita dall'utente).*

Ingegneria del Software 9 *È possibile aggiungere facilmente nuovi algoritmi all' STL senza dover modificare le classi container.*

5.1 `fill`, `fill_n`, `generate` e `generate_n`

Il programma in `codice10.cpp` mostra le funzioni della STL `fill()`, `fill_n()`, `generate()` e `generate_n()`. Le funzioni `fill()` e `fill_n()` impostano un intervallo di elementi di un container a un valore specifico. Le funzioni `generate()` e `generate_n()` utilizzano una funzione generatore per creare valori per un intervallo di elementi di un container. La funzione generatore non prende argomenti e restituisce un valore che può essere posto in un elemento di un container.

```
1   // codice10.cpp
2   // Esempio dei metodi della STL
3   // fill(), fill_n(), generate() e generate_n().
4   #include <iostream>
5   #include <algorithm>
6   #include <vector>
7
8   using namespace std;
9
10  char nextLetter();
11
12  int main()
13  {
14     vector< char > chars( 10 );
15     ostream_iterator< char > output( cout, " ");
16
17     fill( chars.begin(), chars.end(),'5' );
18     cout << "Il vettore chars dopo fill() con 5:\n";
19     copy( chars.begin(), chars.end(), output );
28
21     fill_n( chars.begin(), 5,  'A'  );
22     cout << "Il vettore chars dopo fill_n() di cinque elementi"
23              uguali ad A:\n";
24     copy( chars.begin(), chars.end(), output );
25
26     generate( chars.begin(), chars.end(), nextLetter );
27     cout << "Il vettore chars dopo generate() lettere A-J:\n";
28     copy( chars.begin(), chars.end(), output );
29
30     generate_n( chars.begin(), 5, nextLetter );
31     cout << "Il vettore chars dopo generate_n() K-O per i"
32              primi cinque elementi:\n";
33     copy(chars.begin(), chars.end(), output );
34
35     cout << endl;
36     return 0;
37  }
38
39  char nextLetter()
48  {
```

```
41     static char letter = 'A';
42     return letter++;
43  }
```

Esecuzione:

```
Il vettore chars dopo fill() con 5:
5 5 5 5 5 5 5 5 5 5
Il vettore chars dopo fill_n() di cinque elementi uguali ad A:
A A A A A 5 5 5 5 5
Il vettore chars dopo generate() lettere A-J:
A B C D E F G H I J
Il vettore chars dopo generate_n() K-O per i primi cinque elementi:
K L M N O F G H I J
```

La linea 17

```
fill( chars.begin(), chars.end(),'5' );
```

utilizza la funzione `fill()` per porre il carattere '5', in ogni elemento del vettore chars da `chars.begin()` a `chars.end()` escluso. Notate che gli iteratori forniti come primo e secondo argomento devono essere perlomeno iteratori forward (cioè possono essere utilizzati per accedere sequenzialmente ad un container in avanti).

La linea 21

```
fill_n( chars.begin(), 5, 'A' );
```

utilizza la funzione `fill_n()` per porre il carattere `'A'` nei primi cinque elementi del vettore `chars`. L'iteratore fornito come primo argomento deve essere (almeno) un iteratore di output (cioè può essere utilizzato per l'output su un container in avanti). Il secondo argomento specifica il numero di elementi da impostare. Il terzo argomento specifica il valore da porre in ogni elemento.

La linea 26

```
generate( chars.begin(), chars.end(), nextLetter );
```

utilizza la funzione `generate()` per porre il risultato di una chiamata alla funzione generatore `nextLetter` in ogni elemento del vettore `chars` da `chars.begin()` a `chars.end()` escluso. Gli iteratori forniti come primo e secondo argomento devono essere (almeno) iteratori forward. La funzione `nextLetter` (definita alla linea 39) inizia con il carattere `'A'` mantenuto in una variabile locale static. L'istruzione di linea 42

```
return letter++;
```

restituisce il valore corrente di `letter` a ogni chiamata di `nextLetter`, quindi incrementa il valore di `letter`.

La linea 30

```
generate_n( chars.begin(), 5, nextLetter );
```

utilizza la funzione `generate_n()` per porre il risultato di una chiamata alla funzione generatore `nextLetter` in cinque elementi del vettore `chars` a partire da `chars.begin()`. L'iteratore fornito come primo argomento deve essere (almeno) un iteratore di output.

5.2 `equal`, `mismatch` e `lexicographical_compare`

Il programma in `codice11.cpp` mostra il confronto di sequenze di valori tramite le funzioni della STL `equal()`, `mismatch()` e `lexicographical_compare()`.

```
1   // codice11.cpp
2   // Esempio di utilizzo delle funzioni equal(),
3   // mismatch(), lexicographical_compare().
4   #include <iostream>
5   #include <algorithm>
6   #include <vector>
7
8   using namespace std;
9
10  int main()
11  {
12     const int SIZE = 10;
13     int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14     int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
15     vector< int > v1( a1, a1 + SIZE ),
16                   v2( a1, a1 + SIZE ),
17                   v3( a2, a2 + SIZE );
18     ostream_iterator< int > output( cout, " ");
19
28     cout << "Il vettore v1 contiene:  ";
21     copy( v1.begin(), v1.end(), output );
22     cout << "\nIl vettore v2 contiene: ";
23     copy( v2.begin(), v2.end(), output );
24     cout << "\nIl vettore v3 contiene: ";
25     copy( v3.begin(), v3.end(), output );
26
27     bool result = equal( v1.begin(), v1.end(), v2.begin() );
28     cout << "\nIl vettore v1 " << ( result ? "e'" : "non e'" )
29          << " uguale al vettore v2.\n";
38
31     result = equal( v1.begin(), v1.end(), v3.begin() );
32     cout << "Il vettore v1 " <<   ( result ? "e'" :  "non e'" )
33          << " uguale al vettore v3.\n";
34
35     pair< vector< int >::iterator,
36           vector< int >::iterator > location;
37     location  mismatch( v1.begin(), v1.end(), v3.begin() );
38     cout << "C'e' un mismatch tra v1 e v3 alla"
39          << "locazione  " << ( location.first - v1.begin() )
48          << ", dove v1 contiene "  << *location.first
41          << " e v3 contiene " << *location.second
42          << "\n";
43
```

```
44     char c1[ SIZE ] = "HELLO", c2[ SIZE ] = "BYE BYE";
45
46     result = lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
48     cout << c1
49          << ( result ? " e' minore di " : " e' maggiore di " )
58          << c2;
51
52     cout << endl;
53     return 0;
54  }
```

ESECUZIONE:

```
Il vettore v1 contiene: 1 2 3 4 5 6 7 8 9 10
Il vettore v2 contiene: 1 2 3 4 5 6 7 8 9 10
Il vettore v2 contiene: 1 2 3 4 1000 6 7 8 9 10
Il vettore v1 e' uguale al vettore v2
Il vettore v1 non e' uguale al vettore v3
C'e' un mismatch tra v1 e v3 alla locazione 4, dove v1 contiene 5 e v3 contiene 1000
HELLO e' maggiore di BYE BYE
```

La linea 27

```
bool result = equal( v1.begin(), v1.end(), v2.begin() );
```

utilizza la funzione `equal()` per verificare l'uguaglianza di due sequenze di valori. Ogni sequenza non deve contenere necessariamente lo stesso numero di elementi: `equal()` restituisce false se le sequenze non sono della stessa lunghezza. La funzione `operator==` effettua il confronto degli elementi. In questo esempio sono confrontati gli elementi in vector `v1` a partire da `v1.begin()` fino a `v1.end()` escluso e gli elementi in vector `v2` a partire da `v2.begin()` (in questo esempio `v1` e `v2` sono uguali). I tre argomenti devono essere (almeno) iteratori di input (cioè possono essere utilizzati nell'input da una sequenza in avanti). La linea 31 utilizza la funzione `equal()` per confrontare i vector `v1` e `v3`, che non risultano uguali.

C'è un'altra versione della funzione `equal()` che prende una funzione predicativa binaria come quarto parametro. La funzione predicativa riceve due elementi da confrontare e restituisce un valore bool che indica se gli elementi sono uguali. Ciò è utile nelle sequenze di puntatori a valori anziché di valori effettivi, perché è possibile definire un confronto su ciò a cui puntano i puntatori anziché sugli stessi puntatori (ovvero sugli indirizzi che contengono).

Le linee 35-37

```
pair< vector< int >::iterator,
      vector< int >::iterator > location;
location  mismatch( v1.begin(), v1.end(), v3.begin() );
```

istanziano un pair di iteratori di nome `location` per un vector di interi. Questo oggetto memorizza il risultato della chiamata a `mismatch()` alla linea 37. La funzione `mismatch()` confronta due sequenze di valori e restituisce un pair di iteratori che indica la posizione in ogni sequenza degli elementi diversi. Se tutti gli elementi sono uguali, i due iteratori in `location` sono uguali all'ultimo iteratore di ogni sequenza. I tre argomenti devono essere (almeno) iteratori di input.

Per determinare la posizione effettiva della disuguaglianza nei vector di questo esempio, viene utilizzata l'espressione di linea 39 `location.first - v1. begin()`. Il risultato di questo calcolo è il numero di elementi che separano gli iteratori. Ciò corrisponde all'elemento `number` in questo esempio, perché il confronto viene effettuato dall'inizio di ciascun vector.

Come per la funzione `equal()`, c'è un'altra versione di `mismatch()` che prende una funzione predicativa binaria come quarto parametro.

Le linea 46

```
result = lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
```

utilizza la funzione `lexicographical_compare()` per confrontare il contenuto di due array di caratteri. I quattro argomenti di questa funzione devono essere (almeno) iteratori di input.

Come sapete, i puntatori in array sono iteratori ad accesso casuale. I primi due argomenti iteratore specificano l'intervallo di posizioni nella prima sequenza. Gli ultimi due argomenti iteratore specificano l'intervallo di posizioni nella seconda sequenza. Nell'attraversamento della sequenza tramite gli iteratori, se un elemento della prima sequenza è minore dell'elemento corrispondente nella seconda sequenza, la funzione restituisce true. Se l'elemento nella prima sequenza è maggiore o uguale all'elemento nella seconda sequenza, la funzione restituisce false. Questa funzione si può anche utilizzare per porre le sequenze in ordine lessicografico. Sequenze del genere conterranno tipicamente stringhe.

5.3 `remove`, `remove_if`, `remove_copy` e `remove_copy_if`

Il programma in `codice12.cpp` mostra l'eliminazione da una sequenza tramite le funzioni della STL `remove()`, `remove_if()`, `remove_copy()` e `remove_copy_if()`.

```
 1 // codice12.cpp
 2 // Esempio di utilizzo delle funzioni remove(), remove_if()
 3 // remove_copy() and remove_copy_if()
 4 #include <iostream>
 5 #include <algorithm>
 6 #include <vector>
 7
 8 using namespace std;
 9
10 bool greater9( int );
11
12 int main()
13 {
14    const int SIZE = 10;
15    int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
16    ostream_iterator< int > output( cout, " " );
17
18    // Rimuove 10 da v
19    vector< int > v( a, a + SIZE );
20    vector< int >::iterator newLastElement;
21    cout << "Il vettore v prima di rimuovere tutti i 10:\n";
22    copy( v.begin(), v.end(), output );
```

```
23    newLastElement = remove( v.begin(), v.end(), 10 );
24    cout << "\nIl vettore v dopo aver rimosso tutti i 10:\n";
25    copy( v.begin(), newLastElement, output );
26
27    // Copia da v2 a c, remuove i 10
28    vector< int > v2( a, a + SIZE );
29    vector< int > c( SIZE, 0 );
30    cout << "\n\nIl vettore v2 prima di rimuovere tutti i 10:"
31         << "e copiare:\n";
32    copy( v2.begin(), v2.end(), output );
33    remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
34    cout << "\nIl vettore c dopo aver elim. tutti i 10 da v2:\n";
35    copy( c.begin(), c.end(), output );
36
37    // Rimuove gli elementi maggiori di 9 da v3
38    vector< int > v3( a, a + SIZE );
39    cout << "\n\nIl vettore v3 prima di rimuovere tutti gli"
40         << "\nelementi maggiori di 9:\n";
41    copy( v3.begin(), v3.end(), output );
42    newLastElement =
43       remove_if( v3.begin(), v3.end(), greater9 );
44    cout << "\nIl vettore v3 dopo aver rimosso tutti gli elementi"
45         << "\nmaggiori di 9:\n";
46    copy( v3.begin(), newLastElement, output );
47
48    // Copia degli elementi da v4 a c2,
49    // con rimozione degli elementi maggiori di 9
50    vector< int > v4( a, a + SIZE );
51    vector< int > c2( SIZE, 0 );
52    cout << "\n\nIl vettore v4 prima della rimozione di tutti gli"
53         << "\n elementi maggiori di 9 e copia:\n";
54    copy( v4.begin(), v4.end(), output );
55    remove_copy_if( v4.begin(), v4.end(),
56                        c2.begin(), greater9 );
57    cout << "\nIl vettore c2 dopo la rimozione di tutti gli "
58         << "\nelementi maggiori di 9 da v4:\n";
59    copy( c2.begin(), c2.end(), output );
60
61    cout << endl;
62    return 0;
63 }
64
65 bool greater9( int x )
66 {
67    return x > 9;
68 }
```

Esecuzione:

```
Il vettore v prima di rimuovere tutti i 10:
10 2 10 4 16 6 14 8 12 10
Il vettore v dopo aver rimosso tutti i 10:
2 4 16 6 14 8 12

Il vettore v2 prima di rimuovere tutti i 10 e copiare:
10 2 10 4 16 6 14 8 12 10
Il vettore c dopo aver elim. tutti i 10 da v2:
2 4 16 6 14 8 12 0 0 0

Il vettore v3 prima di rimuovere tutti gli elementi maggiori di 9:
10 2 10 4 16 6 14 8 12 10
Il vettore v3 dopo aver rimosso tutti gli elementi maggiori di 9:
2 4 6 8

Il vettore v4 prima della rimozione di tutti gli elementi maggiori di 9 e copia:
10 2 10 4 16 6 14 8 12 10
Il vettore c2 dopo la rimozione di tutti gli elementi maggiori di 9 da v4:
2 4 6 8 0 0 0 0 0 0
```

La linea 23

```
newLastElement = remove( v.begin(), v.end(), 10 );
```

utilizza la funzione `remove()` per eliminare dal vettore `v` tutti gli elementi con valore 10 nell'intervallo da `v.begin()` a `v.end()` escluso. I primi due argomenti devono essere iterarori forward, in modo che l'algoritmo possa modificare gli elementi nella sequenza. Questa funzione non modifica il numero di elementi nel vector né distrugge gli elementi eliminati, ma sposta tutti gli elementi non eliminati verso l'inizio del vector. La funzione restituisce un iteratore posizionato dopo l'ultimo elemento del vector che non è stato eliminato. Gli elementi dalla posizione dell'iteratore alla fine del vector hanno valori "indefiniti" (in questo esempio, ogni posizione indefinita ha valore 0). La linea 33

```
remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
```

utilizza la funzione `remove_copy()` per copiare dal vector `v2` tutti gli elementi che non hanno valore 10 nell'intervallo da `v2.begin()` a `v2.end()` escluso. Gli elementi sono posti nel vector `c` a partire dalla posizione `c.begin()`. Gli iteratori forniti come primi due argomenti devono essere iteratori di input. L'iteratore fornito come terzo argomento deve essere un iteratore di output, in modo che gli elementi copiati possano essere inseriti nella posizione della copia. Questa funzione restituisce un iteratore posizionato dopo l'ultimo elemento copiato nel vector `c`. Notate alla linea 29 l'uso del costruttore del vector che riceve il numero di elementi del vector ed il valore iniziale di tali elementi.

Le linee 42 e 43

```
  newLastElement =
remove_if( v3.begin(), v3.end(), greater9 );
```

utilizzano la funzione `remove_if()` per eliminare dal vector `v3` tutti gli elementi dell'intervallo da `v3.begin()` a `v3.end()` escluso per cui la funzione predicativa unaria definita dall'utente `greater9` restituisce true. La funzione `greater9` è definita alla linea 64 e restituisce true se il valore passatole è maggiore di 9 e restituisce false altrimenti. Gli iteratori forniti come primi due argomenti devono essere iteratori forward, in modo che l'algoritmo possa modificare gli elementi nella sequenza. Questa funzione non modifica il numero di elementi nel vector, ma sposta verso l'inizio del vector tutti gli elementi che non sono stati eliminati. Questa funzione restituisce un iteratore posizionato dopo l'ultimo elemento nel vector che non è stato eliminato. Tutti gli elementi a partire dalla posizione iteratore fino alla fine del vector hanno valore indefinito.

Le linee 55 e 56

```
remove_copy_if( v4.begin(), v4.end(),
                c2.begin(), greater9 );
```

utilizzano la funzione `remove_copy_if()` per copiare dal vector `v4` tutti gli elementi dell'intervallo da `v4.begin()` a `v4.end()` escluso per cui la funzione predicativa unaria `greater9` restituisce true. Gli elementi sono posti nel vector `c2` a partire dalla posizione `c2.begin()`. Gli iteratori forniti come primi due argomenti devono essere iteratori di input. L'iteratore fornito come terzo argomento deve essere un iteratore di output, in modo che l'elemento da copiare sia inserito nella posizione di copia. Questa funzione restituisce un iteratore posizionato dopo l'ultimo elemento copiato nel vector `c2`.

5.4 `replace`, `replace_if`, `replace_copy` e `replace_copy_if`

Il programma in `codice13.cpp` mostra la sostituzione di valori da una sequenza tramite le funzioni della STL `replace()`, `replace_if()`, `replace_copy()` e `replace_copy_if()`.

```
 1 // codice13.cpp
 2 // Esempio di utilizzo delle funzioni replace(), replace_if()
 3 // replace_copy() and replace_copy_if()
 4 #include <iostream>
 5 #include <algorithm>
 6 #include <vector>
 7
 8 using namespace std;
 9
10 bool greater9( int );
11
12 int main()
13 {
14   const int SIZE = 10;
15   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
16   ostream_iterator< int > output( cout, " " );
17
18   // Sostituisce i 10 in v1 con 100
19   vector< int > v1( a, a + SIZE );
20   cout << "Il vettore v1 prima della sostituzione di tutti i 10:\n";
21   copy( v1.begin(), v1.end(), output );
22   replace( v1.begin(), v1.end(), 10, 100 );
```

```
23   cout << "\nIl vettore v1 dopo la sostituzione di tutti i 10 con 100:\n";
24   copy( v1.begin(), v1.end(), output );
25
26   // Copia da v2 a c1, sostituendo i 10 con 100
27   vector< int > v2( a, a + SIZE );
28   vector< int > c1( SIZE );
29   cout << "\n\nIl vettore v2 prima della sostitituzione di "
30        << "tutti i 10 e della copia:\n";
31   copy( v2.begin(), v2.end(), output );
32   replace_copy( v2.begin(), v2.end(),
33                       c1.begin(), 10, 100 );
34   cout << "\nIl vettore c1 dopo la sostituzione di tutti i 10 in v2:\n";
35   copy( c1.begin(), c1.end(), output );
36
37   // Sostituzione dei valori maggiori di 9 in v3 con 100
38   vector< int > v3( a, a + SIZE );
39   cout << "\n\nIl vettore v3 prima della sostituzione dei valori"
40        << " maggiori di 9:\n";
41   copy( v3.begin(), v3.end(), output );
42   replace_if( v3.begin(), v3.end(), greater9, 100 );
43   cout << "\nIl vettore v3 dopo la sostituzione dei valori "
44        << "\nmaggiori di 9 con 100:\n";
45   copy( v3.begin(), v3.end(), output );
46
47   // Copia da v4 a c2, sostituendo gli elementi maggiori di 9 con 100
48   vector< int > v4( a, a + SIZE );
49   vector< int > c2( SIZE );
50   cout << "\n\nIl vettore v4 prima della sostituzione dei "
51        << "\nvalori maggiori di 9 e di copiare:\n";
52   copy( v4.begin(), v4.end(), output );
53   replace_copy_if( v4.begin(), v4.end(), c2.begin(),
54                           greater9, 100 );
55   cout << "\nIl vettore c2 dopo la sostituzione di tutti i "
56        << "\n valori maggiori di 9 in v4:\n";
57   copy( c2.begin(), c2.end(), output );
58
59   cout << endl;
60   return 0;
61 }
62
63 bool greater9( int x )
64 {
65    return x > 9;
66 }
```

Esecuzione:

```
Il vettore v1 prima di sostituzione tutti i 10:
```

```
10 2 10 4 16 6 14 8 12 10
Il vettore v1 dopo la sostituzione di tutti i 10 con 100:
100 2 100 4 16 6 14 8 12 100

Il vettore v2 prima della sostitituzione di tutti i 10 e della copia:
10 2 10 4 16 6 14 8 12 10
Il vettore c1 dopo la sostuzione di tutti i 10 in v2:
100 2 100 4 16 6 14 8 12 100

Il vettore v3 prima della sostituzione dei valori maggiori di 9:
10 2 10 4 16 6 14 8 12 10
Il vettore v3 dopo la sostituzione dei valori maggiori di 9 con 100:
100 2 100 4 100 6 100 8 100 100

Il vettore v4 prima della sostituzione dei valori maggiori di 9 e di copiare:
10 2 10 4 16 6 14 8 12 10
Il vettore c2 dopo la sostituzione di tutti i valori maggiori di 9 in v4:
100 2 100 4 100 6 100 8 100 100
```

La linea 22

```
replace( v1.begin(), v1.end(), 10, 100 );
```

utilizza la funzione `replace()` per sostituire tutti gli elementi con valore 10 dell'intervallo da `v1.begin()` a `v1.end()` escluso del vector `v1` con il nuovo valore 100. Gli iteratori forniti come primi due argomenti devono essere iteratori forward, in modo che l'algoritmo possa modificare gli elementi nella sequenza.

Le linee 32 e 33

```
replace_copy( v2.begin(), v2.end(),
                    c1.begin(), 10, 100 );
```

utilizzano la funzione `replace_copy()` per copiare tutti gli elementi nell'intervallo da `v2.begin()` a `v2.end()` escluso dal vector `v2` sostituendo tutti gli elementi con valore 10 con il nuovo valore 100. Gli elementi sono copiati nel vector `c1` a partire dalla posizione `c1.begin()`. Gli iteratori forniti come primi due argomenti devono essere iteratori di input. L'iteratore fornito come terzo argomento deve essere un iteratore di output, in modo che l'elemento da copiare sia inserito nella posizione di copia. Questa funzione restituisce un iteratore posizionato dopo l'ultimo elemento copiato nel vector `c2`.

La linea 42

```
replace_if( v3.begin(), v3.end(), greater9, 100 );
```

utilizza la funzione `replace_if()` per sostituire tutti gli elementi dell'intervallo da `v3.begin()` a `v3.end()` escluso per cui la funzione predicativa unaria `greater9` restituisce true. La funzione `greater9` è definita alla linea 63 e restituisce true se il valore passatole è maggiore di 9 e false altrimenti. Il valore 100 sostituisce ogni valore maggiore di 9. Gli iteratori forniti come primi due argomenti devono essere iteratori forward, in modo che l'algoritmo possa modificare gli elementi nella sequenza.

Le linee 53 e 54

```
replace_copy_if( v4.begin(), v4.end(), c2.begin(),
                    greater9, 100 );
```

utilizzano la funzione `replace_copy_if()` per copiare tutti gli elementi nell'intervallo da `v4.begin()` a `v4.end()` escluso dal vector `v4`. Gli elementi per cui la funzione predicativa unaria `greater9` restituisce true sono sostituiti con il valore 100. Gli elementi sono posti nel vector `c2` a partire dalla posizione `c2.begin()`. Gli iteratori forniti come primi due argomenti devono essere iteratori di input. L'iteratore fornito come terzo argomento deve essere un iteratore di output, in modo che l'elemento da copiare possa essere inserito nella posizione di copia. Questa funzione restituisce un iteratore posizionato dopo l'ultimo elemento copiato nel vector `c2`.

5.5 Gli algoritmi numerici

Il programma in `codice14.cpp` mostra l'uso di alcuni comuni algoritmi numerici della STL tra cui `random_shuffle()`, `count()`, `count_if()`, `min_element()`,`max_element()`, `accumulate()`, `for_each()` e `transform()`.

```
 1 // codice14.cpp
 2 // Esempi di algoritmi numerici nella STL.
 3 #include <iostream>
 4 #include <algorithm>
 5 #include <numeric>     // accumulate e' definita qui
 6 #include <vector>
 7
 8 using namespace std;
 9
10 bool greater9( int );
11 void outputSquare( int );
12 int calculateCube( int );
13
14 int main()
15 {
16   const int SIZE = 10;
17   int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
18   vector< int > v( a1, a1 + SIZE );
19   ostream_iterator< int > output( cout, " " );
20
21   cout << "Il vettore v prima di random_shuffle(): ";
22   copy( v.begin(), v.end(), output );
23   random_shuffle( v.begin(), v.end() );
24   cout << "\nIl vettore v dopo random_shuffle(): ";
25   copy( v.begin(), v.end(), output );
26
27   int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
28   vector< int > v2( a2, a2 + SIZE );
29   cout << "\n\nIl vettore v2 contiene: ";
30   copy( v2.begin(), v2.end(), output );
31   int result = count( v2.begin(), v2.end(), 8 );
```

```
32   cout << "\nNumero di elementi uguali a 8: " << result;
33
34   result = count_if( v2.begin(), v2.end(), greater9 );
35   cout << "\n Numero di elementi maggiori di 9: " << result;
36
37   cout << "\nIl minimo elemento nel vettore v2 e': "
38        << *( min_element( v2.begin(), v2.end() ) );
39
40   cout << "\n Il massimo elemento nel vettore v2 e': "
41        << *( max_element( v2.begin(), v2.end() ) );
42
43   cout << "\nIl totale degli elementi nel vettore v e': "
44        << accumulate( v.begin(), v.end(), 0 );
45
46   cout << "\nIl quadrato di ogni intero nel vettore v e':";
47   for_each( v.begin(), v.end(), outputSquare );
48
49   vector< int > cubes( SIZE );
50   transform( v.begin(), v.end(), cubes.begin(),
51                   calculateCube );
52   cout << "\nIl cubo di ogni intero nel vettore v e':";
53   copy( cubes.begin(), cubes.end(), output );
54
55   cout << endl;
56   return 0;
57 }
58
59 bool greater9( int value ) { return value > 9; }
60
61 void outputSquare( int value ) { cout << value * value << ' '; }
62
63 int calculateCube( int value ) { return value * value * value; }
```

Esecuzione:

```
Il vettore v1 prima di random_shuffle(): 1 2 3 4 5 6 7 8 9 10
Il vettore v dopo random_shuffle(): 5 4 1 3 7 8 9 10 6 2

Il vettore v2 contiene: 100 2 8 1 50 3 8 8 9 10
Numero di elementi uguali a 8: 3
Numero di elementi maggiori di 9: 3
Il minimo elemento nel vettore v2 e': 1
Il massimo elemento nel vettore v2 e': 100

Il totale degli elementi nel vettore v e': 55
Il quadrato di ogni intero nel vettore v e': 25 16 1 9 49 64 81 100 36 4
Il cubo di ogni intero nel vettore v e': 125 64 1 27 343 512 729 1000 216 8
```

La linea 34

```
result = count_if( v2.begin(), v2.end(), greater9 );
```

utilizza la funzione `count_if()` per contare gli elementi dell'intervallo da `v2.begin()` a `v2.end()` del vettore `v2` per cui la funzione predicativa `greater9` restituisce true. La funzione `count_if()` richiede due argomenti che devone essere (almeno) iteratori di input.

Le linee 37 e 38

```
cout << "\n\nIl minimo elemento nel vettore v2 e': "
      << *( min_element( v2.begin(), v2.end() ) );
```

utilizzano la funzione `min_element()` per localizzare l'elemento più piccolo nell'intervallo da `v2.begin()` a `v2.end()` del vettore `v2`. La funzione restituisce un iteratore di input posizionato sull'elemento più piccolo o, se l'intervallo è vuoto, restituisce l'iteratore passato come primo elemento. La funzione prende due argomenti che devono essere (almeno) iterarori di input. Una seconda versione di questa funzione prende come terzo argomento una funzione binaria che confronta gli elementi nella sequenza. La funzione binaria prende due argomenti e restituisce il valore true se il primo elemento è più piccolo del secondo e false altrimenti.

Buona Abitudine 2 *È una buona abitudine controllare se l'intervallo specificato nella chiamata a `min_element()` non sia vuoto o controllare che il valore restituito non sia l'iteratore `end()`.*

Le linee 40 e 41

```
cout << "\n Il massimo elemento nel vettore v2 e': "
     << *( max_element( v2.begin(), v2.end() ) );
```

utilizzano la funzione `max_element()` per localizzare l'elemento più grande nell'intervallo da `v2.begin()` a `v2.end()` nel vector `v2`. La funzione restituisce un iteratore di input posizionato sull'elemento più grande, o se l'intervallo è vuoto, restituisce lo stesso iteratore. La funzione prende due argomenti che devono essere (almeno) iteratori di input. Una seconda versione di questa funzione prende come terzo argomento una funzione binaria che confronta gli elementi nella sequenza. La funzione binaria prende due argomenti e restituisce un valore bool.

Le linee 43 e 44

```
cout << "\n\nIl totale degli elementi nel vettore v e': "
     << accumulate( v.begin(), v.end(), 0 );
```

utilizzano la funzione `accumulate()` (il cui prototipo è nel file di intestazione `<numeric>`) pet sommare i valori nell'intervallo da `v.begin()` a `v.end()` nel vector `v`. I due argomenti della funzione devono essere (almeno) iteratori di input. Una seconda versione della funzione prende come terzo argomento una funzione che determina come effettuare la sommatoria degli elementi. Essa deve prendere due argomenti e restituire un risultato. Il primo argomento di questa funzione è il valore corrente dell'accumulazione. Il secondo è il valore dell'elemento corrente nella sequenza da accumulare. Per esempio, per accumulare la somma dei quadrati di ogni elemento, potreste utilizzare la funzione

```
int sumOfSquares( int accumulator, int currentValue )
{
   return accumulator + currentValue * currentValue;
}
```

La linea 47

```
for_each( v.begin(), v.end(), outputSquare );
```

utilizza la funzione `for_each()` per applicare la funzione `outputSquare` a ogni elemento compreso nell'intervallo da `v.begin()` a `v.end()` nel vector `v`. La funzione passata come terzo parametro dovrebbe prendere come argomento l'elemento corrente e non dovrebbe modificarlo. La funzione `for_each()` richiede due iteratori che siano (almeno) iteratori di input.

Le linee 50 e 51

```
transform( v.begin(), v.end(), cubes.begin(),
                calculateCube );
```

utilizzano la funzione `transform()` per applicare la funzione `calculateCube` a ogni elemento compreso nell'intervallo da `v.begin()` a `v.end()` nel vector `v`. La funzione passata come quarto argomento dovrebbe prendere come argomento l'elemento corrente, non dovrebbe modificarlo e dovrebbe restituire il valore trasformato. La funzione `transform()` richiede che i primi argomenti siano (almeno) iteratori di input e che il terzo argomento sia (almeno) un iteratore di output. Il terzo argomento specifica dove deve essere posto il valore trasformato. Notare che il terzo argomento può essere uguale al primo, nel qual caso il risultato della trasformazione viene memorizzato nello stesso vettore che deve essere trasformato.

5.6 Gli algoritmi fondamentali di ordinamento e ricerca

Il programma in `codice15.cpp` mostra l'uso di alcuni algoritmi di ricerca e di ordinamento che fanno parte della STL, come `find()`, `find_if()`, `sort()` e `binary_search()`.

```
 1 // codice15.cpp
 2 // Esempi d'uso delle funzionalita' di ricerca e ordinamento.
 3 #include <iostream>
 4 #include <algorithm>
 5 #include <vector>
 6
 7 using namespace std;
 8
 9 bool greater10( int value );
10
11 int main()
12 {
13   const int SIZE = 10;
14   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
15   vector< int > v( a, a + SIZE );
16   ostream_iterator< int > output( cout, " " );
17
18   cout << "Il vettore v contiene: ";
19   copy( v.begin(), v.end(), output );
20
21   vector< int >::iterator location;
```

```
22   location = find( v.begin(), v.end(), 16 );
23
24   if ( location != v.end() )
25      cout << "\nTrovato 16 alla locazione "
26           << ( location - v.begin() );
27   else
28      cout << "\n16 non trovato";
29
30   location = find( v.begin(), v.end(), 100 );
31
32   if ( location != v.end() )
33      cout << "\nTrovato 100 alla locazione "
34           << ( location - v.begin() );
35   else
36      cout << "\n100 non trovato";
37
38   location = find_if( v.begin(), v.end(), greater10 );
39
40   if ( location != v.end() )
41      cout << "\nIl primo valore maggiore di 10 e' "
42           << *location << "\n; e' stato trovato alla locazione "
43           << ( location - v.begin() );
44   else
45      cout << "\nNon esistono valori maggiori di 10";
46
47   sort( v.begin(), v.end() );
48   cout << "\nIl vettore v dopo sort(): ";
49   copy( v.begin(), v.end(), output );
50
51   if ( binary_search( v.begin(), v.end(), 13 ) )
52      cout << "\n13 e' stato trovato in v";
53   else
54      cout << "\n13 non e' stato trovato in v";
55
56   if ( binary_search( v.begin(), v.end(), 100 ) )
57      cout << "\n100 e' stato trovato in v";
58  else
59      cout << "\n100 non e' stato trovato in v";
60
61   cout << endl;
62   return 0;
63 }
64
65 bool greater10( int value ) { return value > 10; }
```

Esecuzione:

```
Il vettore v contiene: 10 2 17 5 16 8 13 11 20 7
```

```
Trovato 16 alla locazione 4
100 non trovato
Il primo valore maggiore di 10 e' 17; e' stato trovato alla locazione 2
Il vettore v dopo sort(): 2 5 7 8 10 11 13 16 17 20
13 e' stato trovato in v
100 non e' stato trovato in v
```

La linea 22

```
location = find( v.begin(), v.end(), 16 );
```

utilizza la funzione `find()` per localizzare il valore 16 nell'intervallo da `v.begin()` a `v.end()` escluso nel vector `v`. La funzione prende due argomenti che devono essere (almeno) iteratori di input. La funzione restituisce un iteratore di input posizionato sul primo elemento che contiene il valore o un iteratore che indica la fine della sequenza.

La linea 38

```
location = find_if( v.begin(), v.end(), greater10 );
```

utilizza la funzione `find_if()` per localizzare il primo valore contenuto nel vettore `v`, nell'intervallo da `v.begin()` a `v.end()`, per cui la funzione predicativa unaria `greater10` restituisce true. La funzione `greater10` è definita alla linea 65, e prende un intero e restituisce un valore bool che indica se l'argomento passato (un intero) è maggiore di 10. La funzione `find_if()` prende due argomenti che devono essere (almeno) iteratori di input. La funzione restituisce un iteratore di input posizionato sul primo elemento per cui la funzione predicativa restituisce true o un iteratore che indica la fine della sequenza.

La linea 47

```
sort( v.begin(), v.end() );
```

utilizza la funzione `sort()` per disporre gli nell'intervallo da `v.begin()` a `v.end()` escluso nel vector `v` in ordine crescente. La funzione richiede due argomenti iteratore ad accesso casuale. Una seconda versione di questa funzione prende come un terzo argomento una funzione predicativa binaria che prende due argomenti che sono valori nella sequenza e restituisce un risultato bool che indica la disposizione ordinata (se il valore restituito è true i due elementi confrontati sono ordinati).

Errore Tipico 5 *Se tentate di ordinare un container utilizzando un iteratore che non è ad accesso casuale commettete un errore di sintassi. La funzione* `sort()` *richiede un iteratore ad accesso casuale.*

La linea 51

```
if ( binary_search( v.begin(), v.end(), 13 ) )
```

utilizza la funzione `binary_search()` per determinare se il valore 13 è nell'intervallo da `v.begin()` a `v.end()` escluso nel vector `v`. La sequenza di valori deve essere prima ordinata in modo crescente. La funzione `binary_search()` richiede almeno due argomenti iteratori forward. La funzione restituisce un risultato bool che indica se il valore è stato trovato o meno nella sequenza. Una seconda versione di questa funzione prende come quarto argomento una funzione predicativa binaria che prende due argomenti che sono valori nella sequenza e restituisce un risultato bool. La funzione predicativa restituisce true se i due elementi confrontati sono ordinati.

5.7 swap, iter_swap e swap_ranges

Il programma in codice16.cpp mostra l'uso delle funzioni iter_swap(), swap() e swap_ranges() che servono a scambiare di posto gli elementi.

```
 1 // codice16.cpp
 2 // Esempi di iter_swap(), swap() e swap_ranges().
 3 #include <iostream>
 4 #include <algorithm>
 5
 6 using namespace std;
 7
 8 int main()
 9 {
10    const int SIZE = 10;
11    int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
12    ostream_iterator< int > output( cout, " " );
13
14    cout << "L'array a contiene:";
15    copy( a, a + SIZE, output );
16
17    swap( a[ 0 ], a[ 1 ] );
18    cout << "\nL'array a dopo lo scambio di a[0] e a[1] "
19         << "usando swap():";
20    copy( a, a + SIZE, output );
21
22    iter_swap( &a[ 0 ], &a[ 1 ] );
23    cout << "\nL'array a dopo lo scambio di a[0] e a[1] "
24         << "usando iter_swap():";
25    copy( a, a + SIZE, output );
26
27    swap_ranges( a, a + 5, a + 5 );
28    cout << "\nL'array a dopo lo scambio dei primi 5 elementi "
29         << "con gli ultimi 5:";
30    copy( a, a + SIZE, output );
31
32    cout << endl;
33    return 0;
34 }
```

Esecuzione:

```
L'array a contiene: 1 2 3 4 5 6 7 8 9 10
L'array a dopo lo scambio di a[0] e a[1] usando swap(): 2 1 3 4 5 6 7 8 9 10
L'array a dopo lo scambio di a[0] e a[1] usando iter_swap(): 1 2 3 4 5 6 7 8 9 10
L'array a dopo lo scambio dei primi 5 elementi con gli ultimi 5: 6 7 8 9 10 1 2 3 4 5
```

La linea 17

```
swap( a[ 0 ], a[ 1 ] );
```

utilizza la funzione `swap()` per scambiare due valori. In questo esempio, sono scambiati il primo e il secondo elemento dell'array `a`. La funzione prende come argomenti riferimenti ai due valori da scambiare.

La linea 22

```
iter_swap( &a[ 0 ], &a[ 1 ] );
```

utilizza la funzione `iter_swap()` per scambiare due elementi. La funzione prende come argomenti due iteratori forvard (in questo caso, puntatori agli elementi di un array) e scambia i valori degli elementi a cui si riferiscono gli iteratori.

La linea 27

```
swap_ranges( a, a + 5, a + 5 );
```

utilizza la funzione `swap_ranges()` per scambiare gli elementi nell'intervallo da `a` ad `a + 5` escluso, con gli elementi che partono dalla posizione `a + 5`. La funzione richiede tre iteratori forward come argomenti. I primi due specificano l'intervallo di elementi nella prima sequenza che saranno scambiato con gli elementi della seconda sequenza, che parte dall'iteratore specificato come terzo argomento. In questo esempio le due sequenze di valori sono nello stesso array, ma in generale possono essere in array o container diversi.

5.8 `copy_backward`, `merge`, `unique` e `reverse`

Il programma in `codice17.cpp` mostra l'uso delle funzioni della STL `copy_backward()`, `merge()`, `unique()` e `reverse()`.

```
 1 // codice17.cpp
 2 // Esempio di utilizzo delle funzioni copy_backward(), merge(),
 3 // unique() and reverse().
 4 #include <iostream>
 5 #include <algorithm>
 6 #include <vector>
 7
 8 using namespace std;
 9
10 int main()
11 {
12    const int SIZE = 5;
13    int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
14    int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
15    vector< int > v1( a1, a1 + SIZE );
16    vector< int > v2( a2, a2 + SIZE );
17
18    ostream_iterator< int > output( cout, " " );
19
20    cout << "Il vettore v1 contiene: ";
21    copy( v1.begin(), v1.end(), output );
22    cout << "\nIl vettore v2 contiene: ";
```

```
23    copy( v2.begin(), v2.end(), output );
24
25    vector< int > results( v1.size() );
26    copy_backward( v1.begin(), v1.end(), results.end() );
27    cout << "\nDopo copy_backward(), results contiene: ";
28    copy( results.begin(), results.end(), output );
29
30    vector< int > results2( v1.size() + v2.size() );
31    merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
32                results2.begin() );
33    cout << "\nDopo merge() di v1 e v2, results2 contiene:";
34    copy( results2.begin(), results2.end(), output );
35
36    vector< int >::iterator endLocation;
37    endLocation = unique( results2.begin(), results2.end() );
38    cout << "\nDopo unique(), results2 contiene:";
39    copy( results2.begin(), endLocation, output );
40
41    cout << "\nIl vettore v1 dopo reverse(): ";
42    reverse( v1.begin(), v1.end() );
43    copy( v1.begin(), v1.end(), output );
44
45    cout << endl;
46    return 0;
47 }
```

Esecuzione:

```
Il vettore v1 contiene: 1 3 5 7 9
Il vettore v2 contiene: 2 4 5 7 9
Dopo copy_backward(), results contiene: 1 3 5 7 9
Dopo merge() di v1 e v2, results2 contiene: 1 2 3 4 5 5 7 7 9 9
Dopo unique(), results2 contiene: 1 2 3 4 5 7 9
Il vettore v1 dopo reverse(): 9 7 5 3 1
```

La linea 26

```
copy_backward( v1.begin(), v1.end(), results.end() );
```

utilizza la funzione `copy_backward()` per copiare gli elementi nell'intervallo da `v1.begin()` a `v1.end()` escluso nel vector `v1` e porli nel vector `results` a partire dall'elemento prima di `results.end()` operando verso l'inizio del vector. La funzione restituisce un iteratore posizionato come ultimo elemento copiato nel vector `results` (cioè l'inizio di `results` perché la copia è effettuata al contrario). Gli elementi sono posti in `results` nello stesso ordine di `v1`. Questa funzione richiede tre iteratori bidirezionali (iteratori che possono essere incrementati e decrementati per iterare in avanti e all'indietro attraverso una sequenza, rispettivamente). La differenza principale tra `copy()` e `copy_backward()` è che l'iteratore restituito da `copy()` è posizionato dopo l'ultimo elemento copiato mentre l'iteratore restituito da `copy_backward()` sull'ultimo elemento copiato (ovvero il primo

elemento nella sequenza). Inoltre, `copy()` richiede due iteratori di input e un iteratore di output come argomenti.

Le linee 31 e 32

```
merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
           results2.begin() );
```

utilizzano la funzione `merge()` per combinare due sequenze di valori disposti in ordine erescente in una terza sequenza, anch'essa ordinata in modo crescente. La funzione richiede cinque argomenti iteratore. I primi quattro devono essere (almeno) iteratori di input mentre l'ultimo deve essere (almeno) un iteratore di output. I primi due argomenti specificano l'intervallo di elementi nella prima sequenza ordinata (`v1`), i due seguenti specificano l'intervallo di elementi nella seconda sequenza ordinata (`v2`) e l'ultimo specifica l'indirizzo di partenza nella terza sequenza (`results2`) dove verranno uniti gli elementi. Una seconda versione di questa funzione prende come quinto argomento una funzione predicativa binaria che specifica il tipo di ordinamento.

Osservate che la linea 30 crea il vector results con numero di elementi pari a `v1.size() + v2.size()`. Se si vuole utilizzare la funzione `merge()` come è mostrato qui, occorre che la sequenza dove verranno riposti i risultati sia di dimensioni pari almeno alla somma delle due sequenze da unire. Se non volete allocare il numero di elementi per la sequenza risultante prima dell'operazione di `merge()`, potete utilizzare le seguenti istruzioni

```
vector< int > results2;
merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
    back_inserter( results2 ) );
```

L'argomento `back_inserter( results2 )` utilizza il template di funzione `back_inserter()` (file di intestazione `<iterator>`) per il container `results2`. Una funzione `back_inserter()` chiama la funzione `push_back()` del container di default per inserire un elemento alla fine del container. Cosa più importante, se si inserisce un elemento in un container che non ha più elementi disponibili, la dimensione del container aumenta automaticamente. In questo modo, non si ha bisogno di conoscere in anticipo il numero di elementi del container. Ci sono altre due funzioni per l'inserimento, `front_inserter()` (per inserire un elemento all'inizio del container specificato) e `inserter()` (per inserire un elemento prima dell'iteratore fornito come secondo argomento nel container fornito come terzo argomento).

La linea 37

```
endLocation = unique( results2.begin(), results2.end() );
```

utilizza la funzione `unique()` sulla sequenza di elementi ordinata nell'intervallo da `results2.begin()` a `results2.end()` del vettore `results2`. Dopo aver applicato questa funzione a una sequenza ordinata contenente valori duplicati, nella sequenza resterà soltanto una copia di ogni valore. La funzione prende due argomenti che devono essere (almeno) iteratori forward. La funzione restituisce un iteratore posizionato dopo l'ultimo elemento nella sequenza di valori unici. I valori di tutti gli elementi nel container dopo l'ultimo valore unico sono indefiniti. Una seconda versione di questa funzione prende come terzo argomento una funzione predicativa binaria che indica come verificare l'uguaglianza di due elementi.

La linea 42

```
reverse( v1.begin(), v1.end() );
```

utilizza la funzione `reverse()` per invertire l'ordine di tutti gli elementi nell'intervallo da `v1.begin()` a `v1.end()` escluso nel vettore `v1`. La funzione prende due argomenti che devono essere (almeno) iteratori bidirezionali.

5.9 `inplace_merge`, `unique_copy` e `reverse_copy`

Il programma in `codice18.cpp` mostra l'uso delle funzioni della STL `inplace_merge()`, `unique_copy()` e `reverse_copy()`.

```
 1 // codice18.cpp
 2 // Esempio delle funzioni inplace_merge(),
 3 // reverse_copy(), and unique_copy().
 4 #include <iostream>
 5 #include <algorithm>
 6 #include <vector>
 7 #include <iterator>
 8
 9 using namespace std;
10
11 int main()
12 {
13    const int SIZE = 10;
14    int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
15    vector< int > v1( a1, a1 + SIZE );
16
17    ostream_iterator< int > output( cout, " " );
18
19    cout << "Il vettore v1 contiene: ";
20    copy( v1.begin(), v1.end(), output );
21
22    inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
23    cout << "\nDopo inplace_merge(), v1 contiene: ";
24    copy( v1.begin(), v1.end(), output );
25
26    vector< int > results1;
27    unique_copy( v1.begin(), v1.end(),
28                    back_inserter( results1 ) );
29    cout << "\nDopo unique_copy(), results1 contiene: ";
30    copy( results1.begin(), results1.end(), output );
31
32    vector< int > results2;
33    cout << "\nDopo reverse_copy(), results2 contiene: ";
34    reverse_copy( v1.begin(), v1.end(),
35                      back_inserter( results2 ) );
36    copy( results2.begin(), results2.end(), output );
37
38    cout << endl;
39    return 0;
```

```
40 }
```

ESECUZIONE:

```
Il vettore v1 contiene: 1 3 5 7 9 1 3 5 7 9
Dopo inplace_merge(), v1 contiene: 1 1 3 3 5 5 7 7 9 9
Dopo inique_copy(), results1 contiene: 1 3 5 7 9
Dopo reverse_copy(), results2 contiene: 9 9 7 7 5 5 3 3 1 1
```

La linea 22

```
inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
```

utilizza la funzione `inplace_merge()` per unire due sequenze ordinate di elementi nello stesso container. In questo esempio, gli elementi da `v1.begin()` a `v1.begin() + 5` escluso sono uniti con gli elementi da `v1.begin() + 5` a `v1.end()`. Questa funzione richiede tre argomenti che devono essere (almeno) iteratori bidirezionali. Una seconda versione di questa funzione prende come quarto argomento una funzione predicativa binaria per confrontare gli elementi in due sequenze.

Le linee 27 e 28

```
unique_copy( v1.begin(), v1.end(),
                back_inserter( results1 ) );
```

utilizzano la funzione `unique_copy()` per effettuare una copia di tutti gli elementi nella sequenza ordinata di valori da `v1.begin()` a `v1.end()` eliminando i valori duplicati. Gli elementi copiati sono posti nel vector `results1`. I primi due argomenti devono essere (almeno) iteratori di input e l'ultimo argomento deve essere (almeno) un iteratore di output. In questo esempio non abbiamo "preallocato" abbastanza elementi in `results1` per memorizzare tutti gli elementi copiati da `v1`. Invece, chiamiamo `back_inserter()` (definita nel file di intestazione `<iterator>`) per aggiungere elementi alla fine del vector `v1`. La funzione `back_inserter()` utilizza la funzionalità della classe `vector` di inserire elementi alla fine del vector. Dato che `back_inserter()` inserisce un elemento piuttosto che sostituire il valore di un elemento esistente, il vector è in grado di ridimensionarsi per accogliere ulteriori elementi. Una seconda versione della funzione `unique_copy()` prende come quarto argomento una funzione predicativa binaria per verificare l'uguaglianza degli elementi.

Le linee 34 e 35

```
reverse_copy( v1.begin(), v1.end(),
   back_inserter( results2 ) );
```

utilizzano la finzione `reverse_copy()` per effettuare una copia alla rovescia degli elementi nell'intervallo da `v1.begin()` a `v1.end()` escluso. Gli elementi copiati sono inseriti nel vector `results2` tramite un oggetto `back_inserter()` per assicurare che il vector possa aumentare le proprie dimensioni per accogliere il numero appropriato di elementi. La funzione `reverse_copy()` richiede che i primi due argomenti siano (almeno) iteratori bidirezionali e che il terzo sia (almeno) un iteratore di output.

5.10 Algoritmi non discussi

Questa è la lista di alcuni algoritmi della STL non discussi in questa dispensa:

1. `set_difference()`, `set_intersection()`, `set_symmetric_difference()`, `set_union()` (operazioni su insiemi)
2. `lower_bound()`, `upper_bound()`, `equal_range()`
3. `make_heap()`, `sort_heap()`, `push_heap()`, `pop_heap()` (algoritmi di ordinamento)
4. `adjacent_difference()`, `partial_sum()`, `nth_element()`, `inner_product()`, `partition()`, `stable_partition()`, `next_permutation()`, `prev_permutation()`, `rotate()`, `adjacent_find()`, `partial_sort()`, `partial_sort_copy()`, `stable_sort()`

6 La classe bitset

La classe `bitset` semplifica la creazione e la manipolazione di insiemi di bit utilizzati, ad esempio, per rappresentare i flag. Questa dispensa non analizza la classe `bitset`.

7 Gli oggetti funzione

Questa dispensa non tratta gli oggetti funzione. In breve, gli oggetti funzione e gli adattatori di funzione vengono utilizzati per rendere la STL più flessibile. Un oggetto funzione contiene una funzione che può essere invocata tramite l'oggetto utilizzando `operator()`. I prototipi degli oggetti funzione e degli adattatori della STL si trivano in `<functional>`. Un oggetto funzione può anche incapsulare dati insieme con la funzione che contiene. Un esempio di oggetto funzione della STL è `less< T >`:

```
class less {
  public:
    less (int v) : val (v) {}
    int operator () (int v) {
       return v < val;
    }
  private:
    int val;
};
```

Questo oggetto funzione deve essere creato specificando un valore intero:

```
less  less_than_five (5);
```

Quando viene chiamato il costruttore, il valore di `v` è assegnato all'attributo `val`. Quando l'oggetto funzione è applicato, il valore di ritorno dell'operatore `operator()` dice se l'argomento passato all'oggetto funzione è minore di `val`:

```
cout << "2 is less than 5: " << (less_than_five (2) ? "yes" : "no");
```

L'esecuzione di questa istruzione produce:

```
2 is less than 5:  yes
```

A volte è necessario passare oggetti funzione come argomenti ad algoritmi e a costruttori di contenitori associativi.

8 Specializzare i container mediante ereditarità

È possibile specializzare i containers mediante ereditarietà. Ad esempio, la classe `FileEditor`, contenuta nei file `FileEditor.h` e `FileEditor.cpp` di seguito riportati, specializza un contenitore di tipo `vector<string>` allo scopo di realizzare un semplice editor di file di testo.

L'idea di base è che un oggetto `FileEditor`, mediante ereditarietà, acquisisce le struture dati di `vector<string>`. Queste strutture dati risultano inizialmente vuote. Il costrutture di `FileEditor` usa i metodi di `FileEditor` come funzioni di utilità per modificare le strutture dati; le strutture dati sono modificate inserendo informazioni prelevate da un file di testo (ogni riga del file diviene un elemento del contenitore).

L'esempio potrebbe essere reso più realistico inserendo ulteriori metodi (ad esempio, un metodo per la sostituzione di stringhe quale tipica funzione *search/replace*).

```
 1 // FileEditor.h
 2 #ifndef FILEEDITOR_H
 3 #define FILEEDITOR_H
 4
 5 #include <string>
 6 #include <vector>
 7 #include <iostream>
 8
 9 class FileEditor : public vector<string> {    // --------- ereditarieta' !!
10 public:
11
12   FileEditor( );
13   FileEditor( string  );
14   void open ( string );
15   void write( ostream & = cout );
16   void write_with_linenumbers( ostream & = cout );
17
18 private:
19
20 };
21
22 #endif
```

Quello che segue è il codice per l'implementazione della classe `FileEditor`.

```
1 #include <fstream>
2 #include "FileEditor.h"
3
4 FileEditor::FileEditor( ) {
5 }
```

```
 6
 7 FileEditor::FileEditor( string filename ) {
 8   open(filename);
 9 }
10
11 void FileEditor::open( string filename ) {
12   ifstream in;
13   in.open(filename.c_str(), ios::in);
14   string  line;
15   while ( getline(in, line) )
16      push_back(line);
17 }
18
19 void FileEditor::write( ostream & out = cout ) {
20   for ( iterator w=begin(); w!=end(); w++)
21      out << *w << endl;
22 }
23
24 void FileEditor::write_with_linenumbers( ostream & out = cout ) {
25   i=1;
26   for ( iterator w=begin(); w!=end(); w++) {
27      out << *w << endl;
28      i++;
29   }
30 }
```

Il codice che segue mostra un programma di test della classe `FileEditor`. Il programma apre un file di testo specificato come argomento oppure, in assenza di tale argomento, il file di default `dati.txt`. Dopo l'apertura del file, il programma stampa in output il contenuto del file includendo il numero di linea.

```
 1 // codice19.cpp
 2 // Test della classe FileEditor
 3
 4 #include <iostream>
 5 #include <stdlib.h>
 6 #include "FileEditor.h"
 7
 8 using namespace std;
10
11 int main(int argc, char* argv[])
12 {
13   FileEditor file;
14
15   if (argc>1) {
16      file.open( argv[1] );
17   } else {
18      file.open( "dati.txt" );
```

```
19   }
20
21   file.write_with_linenumbers();
22
23   return 0;
24 }
```

Riferimenti bibliografici

[1] H. M. Deitel and P. J. Deitel. *C++ Tecniche Avanzate di Programmazione*. Apogeo, Gennaio 2001.

[2] B. Eckel. *Thinking in C++, 2nd Edition* (volume 2, chapter 7). `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`.

[3] *Standard Template Library Programmer's Guide*. `http://www.sgi.com/tech/stl/`.

[4] J. Kirman. *A modest STL Tutorial*. `http://www.cs.brown.edu/people/jak/proglang/cpp/stltut/tut.ps.zip`.