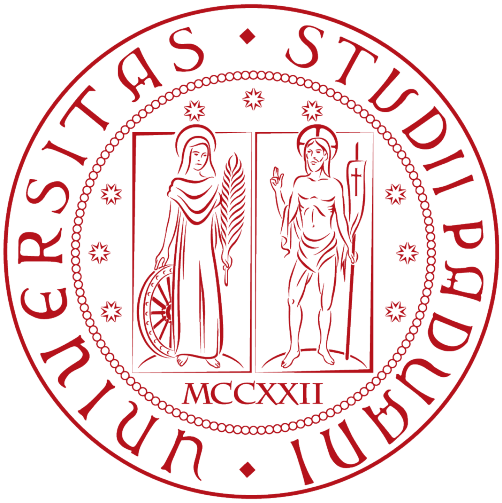
UNIVERSITÀ DEGLI STUDI DI PADOVA

SCUOLA DI SCIENZE



CORSO DI LAUREA IN INFORMATICA

PAO

QUICK REFERENCE

Riassunto Programmazione ad Oggetti

Federico Silvio Busetto Matricola:1026925

ANNO ACCADEMICO 2014-2015

Indice

1 Introduzione **3**
- 1.1 Licenza . . . 3
- 1.2 Riferimenti utili . . . 3
- 1.3 Errata corrige . . . 3

2 Classi e Oggetti **4**
- 2.1 Modularizzazione . . . 4
- 2.2 Costruttori e Assegnazione . . . 4
  - 2.2.1 Costruttore standard . . . 4
  - 2.2.2 Costruttore di copia . . . 5
  - 2.2.3 Assegnazione . . . 5
  - 2.2.4 Costruttore ad un parametro . . . 5
  - 2.2.5 Ridefinizione di operatori . . . 5

3 Classi Collezione e Argomenti correlati **6**
- 3.1 Il problema dell'interferenza . . . 6
  - 3.1.1 Assegnazione Profonda . . . 6
  - 3.1.2 Costruttore di Copia . . . 7
  - 3.1.3 Distruttore . . . 7
- 3.2 Cast . . . 7
- 3.3 Overloading degli operatori e Dichiarazioni d'amicizia . . . 8
- 3.4 Classi contenitore e iteratori . . . 8
  - 3.4.1 Iteratori . . . 8
- 3.5 Condivisione controllata della memoria . . . 9
  - 3.5.1 Puntatori smart . . . 10

4 Template **11**
- 4.1 Template di funzione . . . 11
- 4.2 Template di classe . . . 12
  - 4.2.1 Metodi di template di classe . . . 12
  - 4.2.2 Dichiarazioni d'amicizia in template di classe . . . 12
- 4.3 Membri statici in template . . . 12

5 I contenitori della libreria STL **13**
  - 5.0.1 Costruttori, metodi e operatori . . . 13
- 5.1 Iteratori . . . 14
  - 5.1.1 Metodi e Operatori . . . 14
- 5.2 Sequenze . . . 14
  - 5.2.1 Funzionalità fornite . . . 15

Elenco delle figure

Capitolo 1. Introduzione

Queste dispense sono un riassunto super condensato del libro *Programmazione ad Oggetti di F. Ranzato* Negli esempi si useranno principalmente le classi indicate nel libro, soprattutto le classi orario, come classe di un tipo concreto (ma rimpiazzabile da qualunque altro a scelta).
DISCLAIMER: Queste note vanno usate solo per ripassare velocemente gli argomenti trattati, non sono da utilizzarsi come sostituto alla frequenza delle lezioni o allo studio approfondito sul libro di testo consigliato.

1.1 Licenza

Il codice LaTeXdi questa dispensa è distribuito sotto licenza *GPL v.3* `https://www.gnu.org/licenses/gpl-3.0.en.html`.

1.2 Riferimenti utili

In questa sezione vengono riproposti link utili d'approfondimento degli argomenti trattati:
`http://linux.die.net/man/1/g++`[Manuale g++] `https://www.sgi.com/tech/stl/`[Standard Template Library's Programming Guide - SGI] `https://gcc.gnu.org/onlinedocs/libstdc++/index.html`[GNU listdc++ FSF]

1.3 Errata corrige

Potete segnalare errori, sviste, correzioni o miglioramenti aprendo una pull request al repository presente su github a questo indirizzo: `https://github.com/fedsib/PAO-quick-reference`. Ogni contributo è benvenuto.

Capitolo 2. Classi e Oggetti

Le *struct* al contrario delle classi, permettono di usare la rappresentazione interna dell'ADT.

2.1 Modularizzazione

Classi:

- Definizione di Interfaccia.h, il file di header con tutte le dichiarazioni dell'ADT (campi dato e metodi)
- Implementazione dei suoi metodi nel file .cpp
- in un terzo file, ci va il main

Istruzioni per modularizzazione:

- *g++ -c orario.cpp* compila orario.cpp
- *g++ -c main.cpp* compila il main
- *g++ main.o orario.o* linka i binari

L'information hiding è garantito dal fatto che all'utente verrà consegnato solo il compilato orario.o

2.2 Costruttori e Assegnazione

2.2.1 Costruttore standard

Il costruttore standard è reso disponibile dal linguaggio, ed è un costruttore di default (senza parametri)
Comportamento: Alloca la memoria e lascia i valori dei tipi primitivi e derivati (puntatori, riferimenti e array) indefiniti. Per i tipi classe richiama il costruttore di default dell'oggetto, campo dato per campo dato.
Esempio: Per costruire un Dataora (oggetto composto da due sotto oggetti, una data e un orario), il costruttore standard di Dataora invocherà il costruttore di default di data e il costruttore di default di orario, standard oppure ridefiniti.
NOTA: Nel costruttore di default, gli oggetti di altre classi possono essere costruiti richiamando il loro costruttore di default, come avviene a pag. 50 del libro dove il costruttore di default della classe telefonata, costruisce di default due oggetti di tipo orario: *telefonata::telefonata(text){numero = 0;}*
$NOTA_2$**:** Nel caso di oggetti composti da altri oggetti, omettere nella lista d'inizializzazione del costruttore il costruttore di copia del campo dato di tipo oggetto, farà invocare il costruttore di default di quell'oggetto.

2.2.2 Costruttore di copia

Il costruttore di copia ha firma orario(const orario&){}, è riconoscibile in quattro casi:

- orario pippo = o; dove o è un oggetto di tipo orario precedentemente creato con l'istruzione orario o; In questo primo caso abbiamo una nuova variabile che viene dichiarata, seguita da un assegnazione. La combinazione tipo, nomevariabile, assegnazione, indica che si tratta in realtà di un costruttore di copia
- orario pippo(mezzanotte); se si dichiara una nuova variabile di tipo e gli si passa un altro oggetto dello stesso tipo, non è un costruttore ad un parametro, bensì un costruttore di copia.
- Nel caso di passaggio di parametri per valori in funzioni
- Nel caso di ritorno per valore in funzioni

Comportamento: Copia campo per campo.
Per approfondire: `http://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm`

2.2.3 Assegnazione

L'assegnazione è data dal simbolo = che è un operatore ridefinibile con firma *C& operator=(const C&)*.
Comportamento: L'assegnazione standard tra due oggetti, assegna i valori membro a membro, invocando l'assegnazione standard o ridefinita del tipo. E' pericoloso in quanto può generare interferenze e condivisione non voluta di memoria se l'assegnazione tra campi riguarda puntatori.

2.2.4 Costruttore ad un parametro

Il costruttore con un solo parametro agisce anche come convertitore di tipo (ciò può essere bloccato usando la keyword *explicit*)
Esempio: il costruttore *orario(int){}* genera una conversione implicita dal tipo int al tipo orario (ma non il viceversa).
Sarebbe come ridefinire *operator int()* dichiarandolo nella parte pubblica della classe orario, generando una conversione esplicita.

2.2.5 Ridefinizione di operatori

Un operatore può essere ridefinito come metodo interno alla classe o come funzione esterna. Come metodo, ha sempre un parametro in meno rispetto a quelli richiesti, ad esempio un operatore binario come operator+, avrà un solo parametro tra parentesi, mentre un operatore unario non avrà parametri.Questo vale solo per gli operatori ridefiniti come metodi propri della classe, se invece fossero ridefiniti come funzioni esterne, allora avrebbero un parametro in più di tipo classe (quello che era l'oggetto di invocazione, viene passato per parametro).
Un operatore è ridefinibile solo se tra i parametri ha almeno un tipo definito dall'utente. Ridefinendo un operatore come funzione esterna, non si necessita dell'operatore di scoping nella definizione del metodo nel file .cpp ma l'operatore perderà l'accesso alla parte privata della classe.
NOTA: gli operatori [] e () possono essere ridefiniti solo come metodi propri.
`https://stackoverflow.com/questions/4172722/what-is-the-rule-of-three`(The Rule of Three - StackOverflow)

Capitolo 3. Classi Collezione e Argomenti correlati

3.1 Il problema dell'interferenza

L'interferenza è un problema che consiste in condivisione non voluta della memoria. E'spesso causata da puntatori gestiti in modo improprio.
Mettiamo di avere due campi dato puntatore ad un tipo. *int * p,q;* L'assegnazione standard *p = q;* copia i campi puntatore, cioè assegna al puntatore di sinistra p l'R-valore del puntatore di destra q. L'R-valore di q in realtà è un indirizzo di memoria, e quindi ora p punta all'area di memoria, precedentemente puntata da q.
Quindi il problema nasce perché l'assegnazione standard copia i campi puntatore, ma non gli oggetti ai quali essi puntano, generando aliasing (p e q possono essere usati indistintamente). Oltre all'assegnazione, questo capita anche con i costruttori di copia.
Bisogna prestare particolare attenzione a due aspetti:

- Condivisione di memoria
- Funzioni che modificano oggetti

Il problema può essere risolto, ridefinendo la triade della memoria, ovvero il costruttore di copia, l'assegnazione e il distruttore.

3.1.1 Assegnazione Profonda

Per ridefinire l'assegnazione occorre ridefinire operator=, la cui firma è *C& operator=(const C&);*
operator= ritorna un tipo per riferimento (questo permette di fare assegnazioni a cascata) e prende come secondo paramentro un oggetto di tipo compatibile, passato per riferimento costante (si evita una copia inutile).
operator= va sempre inserito come metodo pubblico della classe da ridefinire, spesso occorrono anche due metodi statici di supporto definiti esternamente ad operator= (per esempio nelle liste):

- *static nodo* copia(nodo*);*
- *static void distruggi(nodo*);*

A questo punto all'interno di *operator=* Dovremmo:

- usare *operator!=* in modo da assicurarci di non stare assegnando lo stesso oggetto a se stesso, operazione inutile.
- liberare la memoria dall'oggetto puntato dal puntatore di sinistra
- fare la copia profonda dell'oggetto e spostare il puntatore
- ritornare l'oggetto di invocazione tramite **this*

Riportiamo qui, ad esempio *operator*= di bolletta

```
bolletta& bolletta::operator=(const bolletta& b){
    if(this != &b){
        distruggi(first);
        first = copia(b.first);
    }
    return *this;
}
```

src/distruttore.cpp

3.1.2 Costruttore di Copia

Basta invocare la funzione di copia, passandoci come parametro il puntatore, nella lista d'inizializzazione del costruttore. Il corpo sarà quindi vuoto.

3.1.3 Distruttore

Il comportamento standard di un distruttore è l'opposto di quello di un costruttore, un distruttore in una funzione distrugge prima:

- le variabili locali di f
- il temporaneo anonimo ritornato da f
- i parametri di f passati per valore

Seguendo esattamente l'ordine contrario rispetto a quello di costruzione degli oggetti.
Una funzione alloca i parametri da destra a sinistra (ecco perché i parametri con valore di default vanno indicati il più a destra possibile, saranno i primi ad essere costruiti), il distruttore dealloca da sinistra a destra (seguendo l'ordine di lettura).
Esempio: *void f(A* $a_1, Aa_2, ..., Aa_n$*)* alloca $a_n, ..., a_2, a_1$ e distrugge $a_1, Aa_2, ..., Aa_n$
Comportamento:

- Viene eseguito il corpo del distruttore
- Vengono invocati gli opportuni distruttori (standard o ridefiniti), nell'ordine inverso a quello di costruzione

Ridefinizione: Va ridefinito sempre e soltanto all'interno della classe la sua firma, per la classe C è $\sim C()$
NOTA: Il distruttore, può essere dichiarato virtuale, per rendere una classe virtuale polimorfa

3.2 Cast

Si rimanda al libro di testo, l'argomento è piuttosto immediato.

Overloading operatori

- *Operatore di standard output*: L'overloading dell'operatore di output (*operator«*) conviene sia sempre fatto come funzione esterna alla classe, anziché come metodo pubblico. Questo però, impedisce l'accesso ai campi e ai metodi definiti nella parte privata della classe. Devono essere usati solo metodi pubblici della classe, come metodi getter

(i metodi che restituiscono il valore di un campo dati). Oppure va definita come funzione amica della classe. L'operatore di output ritorna un'oggetto di tipo ostream per riferimento (in modo da poter usare « a cascata) e prende uno o due parametri conforme se è definito come metodo pubblico all'interno della classe, oppure come funzione esterna. Come metodo pubblico avrà un solo parametro di tipo ostream&, come funzione esterna avrà anche un parametro oggetto della classe, passato per riferimento costante. Per essere usato deve includere *iostream* e usare le opportune direttive d'uso.

- *Operatore di indicizzazione*: L'overloading dell'operatore di indicizzazione (*operator[]*) è spesso usato all'interno di contenitori, in modo da usare la classe contenitore come usualmente si usa un'array. La sua firma è *T& operator[](iteratore it) const* E' un metodo costante, e ritorna puntatori costanti a T. E' necessaria una dichiarazione di amicizia tra iteratore e contenitore, affinché contenitore possa accedere al campo puntatore.

- *Operatore di delete:* Ha firma *void operator delete(void *)* Non ha oggetto di invocazione, e non ritorna nulla. Quello standard (invocabile tramite l'operatore di scoping ::delete) si limita ad invocare i distruttori opportuni, solitamente è ridefinito per ottenere una condivisione controllata della memoria.

3.3 Overloading degli operatori e Dichiarazioni d'amicizia

Dichiarazioni d'amicizia

Tramite la keyword *friend* è possibile definire relazioni di amicizia, esponendo la parte privata di una classe, alla funzione o alla classe dichiarata come amica.
L'uso dell'amicizia avviene in due tappe fondamentali:

- La dichiarazione d'amicizia, posta sempre dentro la classe (qui genericamente indicata con T), anche se la dichiarazione è dentro la classe, la funzione è sempre esterna

- La definizione della funzione, all'interno del file .cpp, va preceduta dalla keyword friend

Esempio di dichiarazione: *friend ostream& operator«(ostream& o, const T&);*

Le dichiarazioni d'amicizia vanno usate solo se strettamente necessario, e qualora vi sia un'alternativa vanno sempre evitate

3.4 Classi contenitore e iteratori

I contenitori standard definiti dal linguaggio verranno trattati più avanti.

3.4.1 Iteratori

Un iteratore generalizza il concetto di puntatore. E' un'oggetto, che punta ad un altro oggetto. Solitamente è una classe pubblica contenuta dentro una classe contenitore.
$NOTA_1$**:** Va sempre preceduta da una dichiarazione d'amicizia, nella parte privata della classe contenitore. La dichiarazione d'amicizia garantisce a tutti i membri di iteratore, l'accesso alla parte privata di contenitore, che sarebbe altrimenti inaccessibile. Al contrario dei metodi, le classi annidate non hanno accesso ai membri delle altre classi più esterne.

$NOTA_2$**:** Anche la classe contenitore deve essere una classe amica della sottoclasse iteratore perché altrimenti non potrebbe avere accesso ai campi privati di iteratore (soprattutto al campo puntatore). La classe contenitore necessita di accedere alla parte privata di iteratore per definire i suoi metodi pubblici.
$NOTA_3$**:** La classe iteratore va dichiarata prima della dichiarazione dei metodi che usano iteratori.
$NOTA_4$**:** Nella classe iteratore non vanno ridefiniti assegnazione, costruttori di copia e distruttori.
L'iteratore è una funzionalità fornita dalla classe contenitore, che fornirà anche dei metodi pubblici per definire iteratori iniziali e finali (oltre a ridefinire l'operatore di indicizzazione come detto sopra).
Riassumendo:

- contenitore contiene una classe pubblica iteratore e la dichiara come sua amica
- iteratore dichiara come sua amica la classe contenitore
- iteratore offre gli operatori di confronto e gli operatori di incremento e decremento prefisso e postfisso.
- contenitore ridefinisce usando l'amicizia l'operatore di indicizzazione [] come metodo pubblico
- contenitore offre anche una ridefinizione dei metodi costanti *begin()*, che ritorna l'iteratore al primo elemento di contenitore e *end()* che ritorna l'iteratore alla posizione successiva a quella finale del contenitore.

3.5 Condivisione controllata della memoria

Tenendo per riferimento la classe bolletta, per ottenere una condivisione controllata della memoria occorre:

- Aggiungere dentro la classe nodo (classe privata, interna a bolletta) un campo contatore per contare i riferimenti a nodo.
- Ridefinire l'operatore di delete di nodo, in modo che se il campo contatore è uguale a zero si invochi la delete standard di nodo, altrimenti si scali semplicemente di uno questo campo.
  - se il puntatore non è nullo, cioè c'è qualcosa da deallocare
  - casto esplicitamente il puntatore da void* a nodo*
  - scalo uno dal contatore dei riferimenti
  - a questo punto, se il contatore vale zero
  - invoco la delete sul nodo successivo della lista
  - invoco la delete standard sulla lista

La ridefinizione di delete di bolletta, è la seguente:

```
1 void bolletta::nodo::operator delete(void* p){
    if(p){
        nodo* q = static_cast<nodo*>(p);
        q->riferimenti--;
5       if(q->riferimenti == 0){}
```

```
            delete q->next;
            ::delete q;
        }
    }
}
```

src/delete.cpp

3.5.1 Puntatori smart

I puntatori smart nascono per automatizzare la gestione del campo riferimenti. Basterebbe ridefinire la triade della memoria per puntatori a nodo ma non è possibile (i puntatori sono tipi derivati e non oggetti di una classe definita dal programmatore). Di seguito sono elencate le peculiarità di Smartp.

- La classe Smartp ha un solo campo dati di tipo Nodo*.
- È preceduta da una dichiarazione incompleta della classe Nodo.
- Ridefinisce la triade della memoria (assegnazione, costruttore di copia e distruttore).
- Ridefinisce i due operatori di confronto $==$ e $!=$
- Ridefinisce l'operatore di dereferenziazione $*$e l'operatore di accesso a membro $\rightarrow$

Cambia anche la classe Nodo.

- Ora è una lista è di Smartp
- Cambia il costruttore a due parametri *Nodo(const telefonata&, const Smartp&)*
- Non occorre più ridefinire la delete di Nodo.

Operatori di Smartp

operator$*$ ritorna un riferimento al Nodo puntato dal puntatore inscatolato nello Smartp (in poche parole, swrappa il puntatore). Ritorna $*$punt.

operator$\rightarrow$ trasforma Smartp in un puntatore a Nodo (Nodo$*$), che viene poi dereferenziato e, con l'operatore di selezione di membro seleziona il campo dati specificato. Ritorna punt.

Capitolo 4. Template

I template sono usati per implementare il polimorfismo parametrico, in modo da rendere disponibile una programmazione generica sui tipi. Essi si dividono in:

- Template di funzione
- Template di classe

4.1 Template di funzione

Un template di funzione rappresenta la descrizione di un metodo, per generare automaticamente istanze di funzione che differiscono per il tipo degli argomenti.

```
template<class T>
T min(T a, T b) {}

min<int>(i,j);        //istanziazione esplicita a int (converte
    implicitamente i tipi dei parametri)
```

src/template.cpp

L'istanziazione implicita avviene invocando normalmente quella esplicita (che converte implicitamente i tipi dei parametri). I tipi della funzione, vengono dedotti dai tipi dei parametri attuali.
La sintassi per dichiarare un template è la seguente:
template <lista parametri> ad es. *template <class T, int size>*
I parametri di *lista parametri* possono essere:

- Parametri di tipo: preceduti dalle keyword *class* o *typename*
- Parametri valore: preceduti dal tipo del valore (int, nell'esempio precedente)

La deduzione dei tipi avviene tramite quattro tipi di conversioni:

- Da L a R valore (T& $\rightarrow$ T)
- Da array a puntatore (T[] $\rightarrow$ T$*$)
- Verso il const (T $\rightarrow$ const T)
- Da valore a riferimento costante (T $\rightarrow$const T&)

Il compilatore controlla i tipi da sinistra a destra.//
NOTA: Il compilatore memorizza le definizioni ma non le compila, genererà il codice macchina solo quando istanzierà i tipi giusti.
Vi sono due modi di compilazione

- **Per inclusione:** Le definizioni dei template sono poste nei file header. Ciò viola il principio dell'*Information Hiding* e genera codice duplicato, rallentando la compilazione; si piò forzare il tipo con una dichiarazione esplicita d'istanziazione *template int min(int,int)*. Usando la direttiva *g++ -fno-implicit-templates* , verrà generato il codice solo per le istanziazioni esplicite.

- **Per separazione:** Le dichiarazioni dei template vanno poste in un file header, le definizioni in un altro file. La definizione va preceduta dalla keyword *export*. Questa tipologia di compilazione non è supportata da tutti i compilatori.

4.2 Template di classe

I template di classe sono utili per avere varie classi che differiscono per il tipo di campi dato, metodi e classi interne.
Occorre sempre un'istanziazione esplicita e i parametri possono avere valori di default.
La definizione del template è necessaria quando si deve operare su oggetti della classe (l'istanza serve per allocare lo spazio degli oggetti) , mentre non è necessaria per puntatori, riferimenti e dichiarazioni incomplete di classe.

4.2.1 Metodi di template di classe

E' possibile definire i metodi all'esterno della classe usando la sintassi:
Nome_classe<Tipo>::Nome_Metodo(){}

4.2.2 Dichiarazioni d'amicizia in template di classe

In un template di classe possono apparire tre tipologie di dichiarazioni d'amicizia;

- Dichiarazione di una classe (o funzione) friend non template (concreta) dentro ad un template di classe. Diventano amiche di istanze del template mentre le classi che contengono funzioni specifiche, usate nella classe template vanno dichiarate prima della classe templetizzata.
- Dichiarazione di un template di classe (o funzione) friend associato (ha tra i suoi parametri, alcuni dei parametri del template C). Tutte le istanze del template C hanno per amica una e una sola istanza del template associato. È il caso più comune, usa dichiarazioni incomplete. va prestata attenzione all'ordine di dichiarazione dei componenti.
- Amicizia con template non associati (gli insiemi dei parametri tr i due template sono disgiunti). Ogni dichiarazione dentro il template C va preceduta da: *template <class Tp>*

Esempi:

Si rimanda agli esempi del libro a pag. 137-144
Un piccolo estratto,
Per le dichiarazioni di template associati: Ad ogni istanza di *QueueItem* è associata una sola istanza di amica della classe *Queue//*
Per le dichiarazioni di template non associate: Tutte le istanze di *Queue* amiche di ogni istanza di *QueueItem* . Avremmo la classe *Queue<int>* amica di *QueueItem<string>*

4.3 Membri statici in template

Ogni istanza di T ha propri campi dati e metodi statici.
ATTENZIONE: Ogni istanza significa, uno comune per ogni tipo.
L'inizializzazione del campo dati statico (istanziato solo se usato), è esterna alla classe, ad es. *template <class T> int Queue<T>::contatore=0;*

Capitolo 5. I contenitori della libreria STL

Un contenitore, è un template di classe C parametrico sul tipo T degli elementi contenuti (alcuni invece sono parametrici su 2 tipi, come *map* e *multimap*).
Ogni classe contenitore definito nella STL ha tra i suoi membri i seguenti quattro tipi:

- *C::value_type*: Il tipo T degli oggetti memorizzati nel contenitore deve essere assegnabile ma non deve necessariamente definire un costruttore di default.
- *C::iterator*: Il tipo iteratore, usato per iterare sugli elementi del contenitore fornisce la ridefinizione di *operator∗* che ritorna un *value_type&*
- *C::const_iterator*: È l'iteratore usato solo per accedere agli elementi del contenitore. Viene usato anche per accedere ad elementi di contenitori costanti.
- *C::size_type*: Rappresenta la distanza tra due iteratori.

5.0.1 Costruttori, metodi e operatori

Un contenitore della Standard Template Library offre i seguenti:

Gestori della memoria

- *C(const C&)*: costruttore di copia ridefinito
- *C& operator=(const C&)* ridefinizione dell'operatore di assegnazione
- $\tilde{C}()$ distruttore

Metodi e operatori

- *size_type size()*: È un intero maggiore o uguale a zero e rappresenta la dimensione del contenitore, cioè il numero di elementi contenuti.
- *bool empty()*: Equivalente a *c.size() == 0* ritorna vero se il contenitore è vuoto, falso altrimenti.
- *size_type max_size()*: Ritorna la massima dimensione che il contenitore c può avere.
- Nelle otto classi contenitore (*vector*,*list*,*slist*,*deque*,*set*,*map*,*multiset*,*multimap*) gli elementi sono memorizzati in un'ordine ben definito, quindi vengono offerte anche le ridefinizioni di:

  operator== e *operator !*=: Ritornano rispettivamente *true* se dati due contenitori b e c, *b.size()==c.size()* se gli elementi contenuti in base al tipo sono uguali e *false* altrimenti (l'inverso per l'altro operatore). Per confrontare i tipi sottostanti, viene invocato *operator*== del tipo contenuto.

  operator<, *operator* <=, *operator* >, *operator* >=: *operator*< ritorna *true* se la sequenza del primo contenitore è minore a quella del secondo in base all'ordine lessicografico, *false* altrimenti. Gli altri operatori hanno comportamenti simili.

5.1 Iteratori

L'iteratore *Past the end (PTE)* (non dereferenziabile) punta sempre alla posizione successiva all'ultimo elemento del contenitore.
Un iteratore è detto valido se è dereferenziabile, oppure è l'iteratore *PTE*.
La classe contenitore se offre un iteratore, fornisce sempre due metodi pubblici:

- *C::iterator (oppure C::const_iterator) begin()*: ritorna un iteratore al primo elemento del contenitore c, se c è vuoto ritorna l'iteratore *PTE*
- *C::iterator (oppure C::const_iterator) end()*: ritorna sempre l'iteratore *PTE*

5.1.1 Metodi e Operatori

Oltre a rendere sempre disponibile una ridefinizione di *operator*$*$ un iteratore può offrire una ridefinizione degli operatori di incremento (e decremento) prefisso e postfisso:

- *C::iterator& operator++()*: Incremento prefisso (notare il tipo di ritorno per riferimento), l'iteratore può diventare *PTE* dopo l'incremento
- *C::iterator& operator–()*: Decremento prefisso, può ritornare un iteratore che punta all'ultimo elemento, se l'operatore è applicato all'iteratore *PTE*
- *C::iterator operator++(int)*: Incremento postfisso (notare il parametro e il tipo di ritorno per valore)
- *C::iterator operator–(int)*: Decremento postfisso (notare il parametro e il tipo di ritorno per valore)

Per *vector* e *deque* sono disponibili iteratori ad accesso casuale (permettono di spostarsi di un numero arbitrario di elementi). Per questi iteratori vengono resi disponibili anche gli operatori di confronto.
Formule utilizzabili per iteratori ad accesso casuale, sia n una variabile intera e it,it1,it2 iteratori ad accesso casuale con $0 \leq n \leq c.size()$:

it += *n* equivalente a n volte ++it se n>0, n volte –it se n<0

it + *n* uguale a

it += *n* (col segno meno si effettuano i decrementi)

it[n] equivalente a $*$*(it+n)* ritorna l'oggetto dereferenziando la posizione dell'iteratore dopo l'incremento (o il decremento)

it1 < *it2* ritorna *true* se it1 punta ad un elemento precedente quello puntato da it2 all'interno del contenitore.

5.2 Sequenze

I contenitori sequenza permettono di memorizzare gli elementi in ordine lineare stretto, stabilito dall'utente del contenitore. Sono contenitori sequenza *list*, *slist*, *vector* e *deque*

5.2.1 Funzionalità fornite

Costruttori

C(C::size_type n, C::value_type t): Costruttore a due parametri. Costruisce il contenitore c contenente n copie dell'elemento t.

C(C::size_type n): Costruttore ad un parametro, costruisce c con n elementi inizializzati al valore di default del template. A patto che il tipo T al quale è istanziato il template renda disponibile il suo costruttore di default.

Metodi di inserimento

C::iterator insert(C::iterator it, C::value_type t): Inserisce l'elemento t prima dell'elemento puntato da it e torna un iteratore all'elemento inserito

void insert(C::iterator it, C::size_type n, C::value_type t): inserisce n copie di t prima dell'elemento puntato da it.

Metodi di cancellazione

Vengono forniti due metodi di erase:

C::iterator erase(C::iterator it): Distrugge l'elemento puntato da it e ritorna un iteratore che punta al successivo.

C::iterator erase(C::iterator it1,C::iterator it2): È equivalente a rimuovere gli elementi compresi tra [it1,it2) e ritorna un iteratore all'elemento successivo a quello puntato da it2.

void clear(): È equivalente a *c.erase(c.begin(), c.end()*, in pratica rimuove tutti gli elementi del contenitore.

Metodi particolari

Per i contenitori *vector*, *list* e *deque* sono disponibili i metodi

void push_back(C::value_type t): Inserisce l'elemento t in coda al contenitore

void pop_back(): rimuove l'elemento in coda al contenitore.

Per i contenitori *list* e *deque* sono disponibili i metodi

void push_front(C::value_type t): Inserisce l'elemento t in testa al contenitore

void pop_front(): rimuove l'elemento in testa al contenitore.