UNIVERSITÀ
DI PARMA

STL (Standard Template Library)

Alberto Ferrari

- la Standard Template Library è stata progettata per gestire ***insiemi di dati*** in modo comodo ed efficiente ***senza conoscere dettagli implementativi***
- fa parte dello ***standard*** C++
- è basata sulla ***programmazione generica***

STL

- la ***Standard Template Library*** fornisce una serie di ***classi*** standard che realizzano dei contenitori *(vettori, liste, alberi, etc.)*
- un contenitore (***container***) è una ***collezione*** di oggetti di tipo omogeneo
- la STL del C++ è basata sulla ***programmazione generica***
  - es. la classe "***list***" viene definita una sola volta: il tipo degli elementi è un tipo generico T
  - non è necessario conoscere a priori quale sarà il tipo T, è sufficiente che questo soddisfi alcuni requisiti (***constraints***)
  - per esempio: il tipo T dei *search tree* deve essere ordinabile (**<**)

contenitori (1)

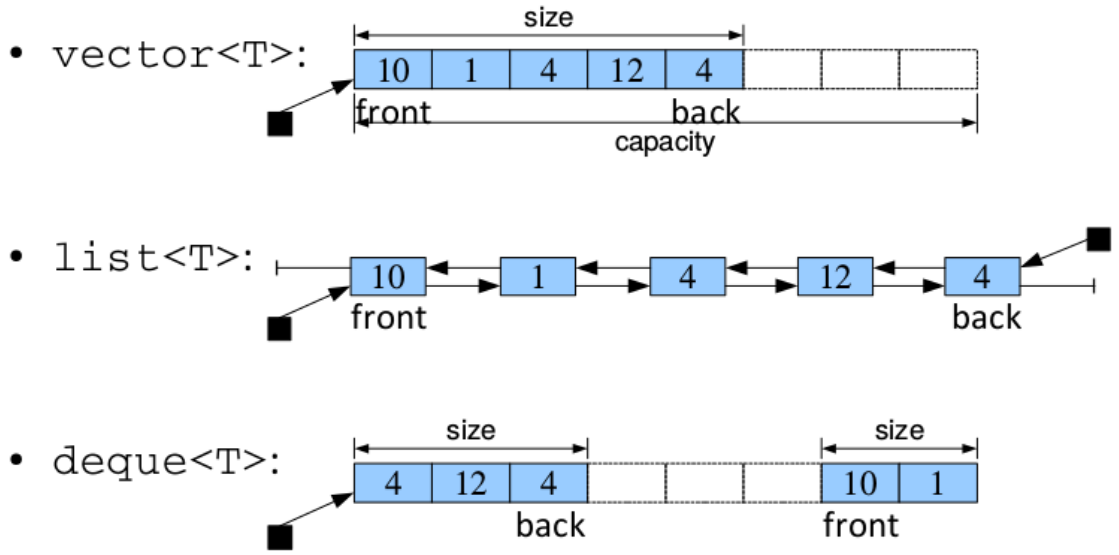
- ***sequenze***
  - sono i contenitori che impongono un ***ordinamento lineare*** delle posizioni degli elementi,
    - dati due elementi si può sempre definire quale sta prima e quale sta dopo
    - gli elementi sono ordinati in base al modo in cui sono inseriti (es. ***list***)
- ***adattatori***
  - non sono contenitori indipendenti ma si appoggiano ad un altro container e ne ***estendono le funzionalità*** (es. ***priority_queue***)

contenitori (2)

- ***quasi-contenitori***
  - non esportano tutte le funzionalità di un contenitore propriamente detto (es. ***string***)
- ***contenitori associativi***
  - ogni elemento è composto da ***chiave*** e ***valore***
  - gli elementi sono ***individuati*** per mezzo della ***chiave***
  - ***non*** impongono un ***ordinamento lineare*** nelle posizioni degli elementi (es. ***map***)

sequenze

- ***vector***
  - è un *array* con *dimensione variabile*
- ***list***
  - è una *lista doppiamente concatenata*
- ***deque***
  - è un *vettore circolare* (*double-ended queue*)



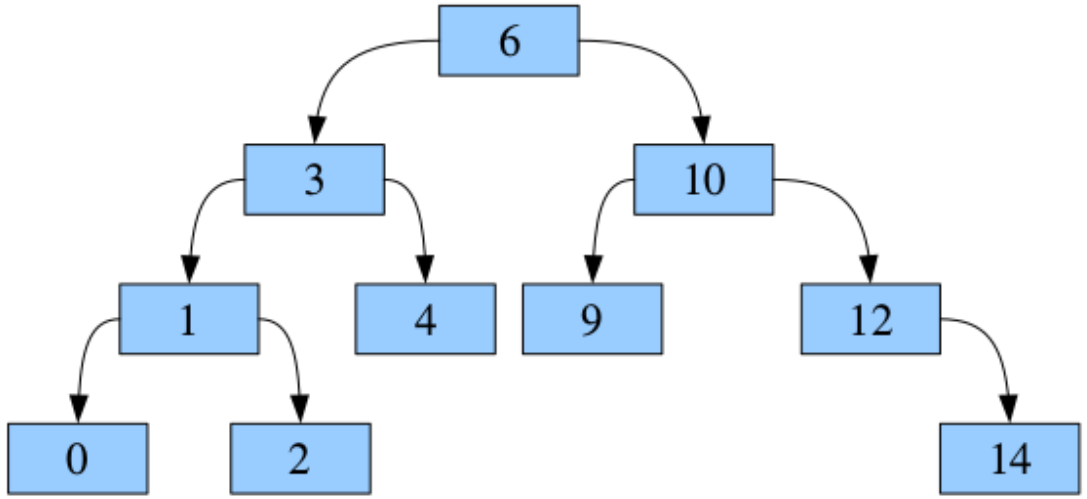
vector – accesso agli elementi

- la notazione **[]**
  - può essere usata per
    - ***leggere*** un elemento
    - ***cambiare*** un elemento che ha già un ***valore***
  - ***non*** può essere usata per
    - ***inizializzare*** un elemento
  - ***non controlla*** se l'elemento esiste
- per ***aggiungere*** elementi (in ordine di posizione) si usa il metodo ***push_back()***

contenitori associativi - set

- contenitori che offrono una ***ricerca efficiente*** basata sul valore di una ***chiave***
- la chiave può essere di un qualsiasi tipo T (***ordinabile***).

- **`set<T>`**
  - è un ***albero binario di ricerca*** di chiavi T



esempio di contenitore associativo: set

- ogni elemento coincide con la sua ***chiave***
- ogni elemento può comparire al più ***una volta***
- per motivi di efficienza memorizza gli elementi in ***ordine*** rispetto al loro ***valore*** (i set sono implementati come ***alberi binari***)

set - esempio

```
void printSet(set<int> s) {
  for (set<int>::iterator
    i=s.begin(); i!=s.end(); i++) {
          cout << *i << " ";
  } cout << endl;
}
…
set<int> tree;
tree.insert(75); tree.insert(25);
tree.insert(50); tree.insert(10);
tree.insert(49); tree.insert(99);
printSet(tree);
```

- un ***set<int>*** è un albero binario di interi
- la struttura set mantiene l'albero ***ordinato*** durante l'inserimento

- output:
- **10 25 49 50 75 99**

set – ricerca e cancellazione


```
if (tree.find(49) != tree.end())
  cout << "49 is in" << endl;

bool is_2_in = tree.find(2) != tree.end();
cout << "is 2 in ? " << is_2_in << "\n";
```

```
output
49 is in
is 2 in ? 0
```

```
printSet(tree);
tree.erase(10);
printSet(tree);
tree.erase(1000);
printSet(tree);
```

```
output
10 25 49 50 75 99
25 49 50 75 99
25 49 50 75 99
```

multiset – valori multipli


```
multiset<int> tree;
tree.insert(75); tree.insert(10);
tree.insert(5); tree.insert(10);
cout << "75 is present " << tree.count(75)
     << " times\n";
cout << "10 is present " << tree.count(10)
     << " times\n";
cout << "2 is present " << tree.count(2)
     << " times\n";
```

```
output
75 is present 1 times
10 is present 2 times
2 is present 0 times
```

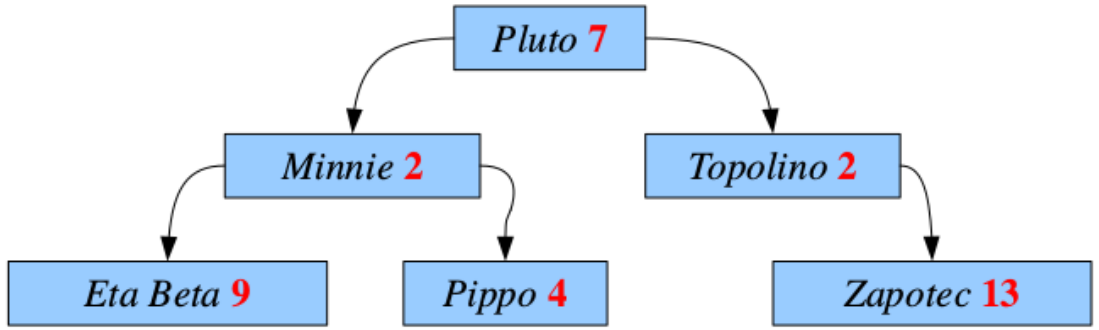
```
printSet(tree);
tree.erase(10);
printSet(tree);
tree.erase(1000);
printSet(tree);
```

```
output
10 25 49 50 75 99
25 49 50 75 99
25 49 50 75 99
```

contenitori associativi - map

- contenitori che offrono una ***ricerca efficiente*** basata sul valore di una chiave
- la chiave può essere di un qualsiasi tipo T (***ordinabile***)

- **`map<T, ValueT>`**
  - è un ***albero binario di ricerca*** di coppie chiave-valore (pair<T, ValueT>)



esempio di contenitore associativo: map

- è un insieme di ***coppie*** ordinate di elementi (pair) formate da ***chiave*** (first) e ***dato*** (second)
- es: `map<string, int> voto;`
- per motivi di efficienza memorizza gli elementi in ordine rispetto al valore della chiave
- se non si specifica un ordinamento, viene usato quello di default

map – esempio

```
map<string, int> m;
m["Pippo"] = 4;     m["Minnie"] = 2;
m["Topolino"] = 2; m["Zapotec"] = 13;
m["Eta Beta"] = 9; m["Pluto"] = 7;
for (map<string, int>::iterator
     i=m.begin(); i!=m.end(); i++) {
  cout<<i->first<<" : "<< i->second<<"; ";
}
```

```
output
Eta Beta : 9; Minnie : 2; Pippo : 4;
Pluto : 7; Topolino : 2; Zapotec : 13;
```

```
map<string, int> m2 = {
  {"Pippo", 4}, {"Minnie", 2},
  {"Topolino", 2}, {"Zapotec", 13},
  {"Eta Beta", 9}, {"Pluto", 7}
};
cout << "Size: " << m2.size() << endl;
m2.erase("Eta Beta");
cout << "Size: " << m2.size() << endl;
```

```
output
Size: 6
Size: 5
```

iteratori

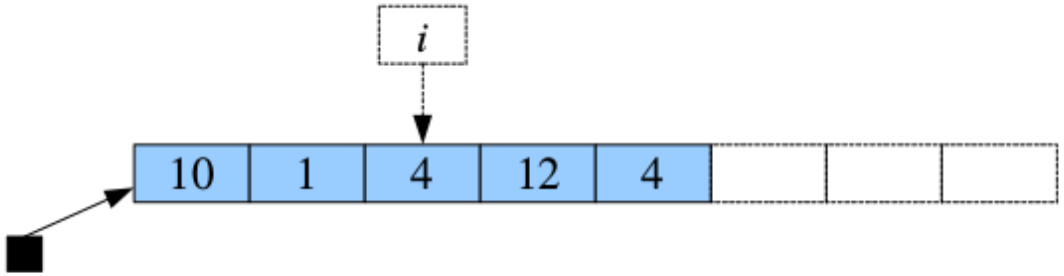
- la «***navigazione***» (accesso agli elementi di un contenitore) avviene tramite gli ***iteratori***
- ogni classe container ha il suo tipo di iteratore ma la ***sintassi*** e la ***semantica*** sono le stesse
- un ***iteratore*** è un oggetto che «***punta***» ad un elemento del contenitore
  - l'iteratore è un'astrazione del concetto di puntatore
- gli iteratori sono progettati per ***nascondere*** i ***dettagli implementativi*** e fornire un modo ***uniforme*** per esplorare le strutture dati

iteratori: operazioni

- gli iteratori offrono un set di ***operazioni comuni*** basate su un'interfaccia che non dipende dal tipo di contenitore iterato
- operazioni fornite:
  - ***lettura***
  - ***scrittura***
  - ***inserimento***
  - ***cancellazione***

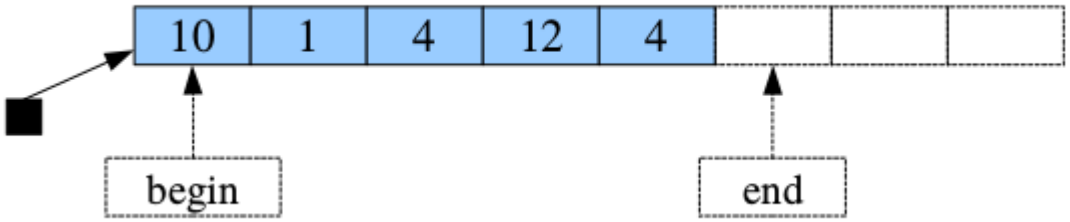
iteratori: operatori

- ***dereferenziazione***
  (accesso all'elemento puntato)
  - `*i`
- passaggio all'elemento ***successivo/precedente*** nella sequenza
  - `i++, i--`
- test di ***uguaglianza/disuguaglianza***
  - `i==j, i!=j`



iteratori: metodi

- su ogni contenitore **`c`** sono definiti due metodi
- **`c.begin()`**
  - restituisce un iteratore che punta al ***primo elemento*** di **`c`**
- **`c.end()`**
  - restituisce un iteratore che punta all'elemento ***successivo dell'ultimo***



tipi di iteratori

- gli iteratori sono classificati secondo il tipo di operazioni ad essi applicabili
  - ***forward iterators***: si può applicare l'operazione **++**
  - ***bidirectional iterators***: si possono applicare le operazioni **++** e **--**
  - ***random access iterators***: si possono applicare le operazioni **++** e **--** e si può ***accedere*** a qualsiasi elemento in ***un solo passo***
- ogni categoria include le precedenti

iteratori constant e mutable

- un iteratore di qualsiasi tipo è
  - ***constant*** se l'operatore ***** restituisce l'elemento puntato come ***r-value***
  - ***mutable*** se l'operatore ***** restituisce l'elemento puntato come ***l-value***
- se una classe container ha iteratori mutable, ha anche iteratori const

reverse iterators

- se una classe container ha iteratori ***bidirezionali***, per passare gli elementi in ordine inverso si possono usare i ***reverse iterators***
- la funzione membro ***rbegin()*** restituisce un iteratore che punta all'***ultimo*** elemento
- la funzione membro ***rend()*** restituisce un valore speciale che può essere usato per verificare se un reverse iterator punta ***oltre*** il ***primo*** elemento
- l'operatore **++** fa avanzare un reverse iterator in ***senso inverso***

visualizza elementi vector<int>

```
void stampa_vector_int(vector<int> v){
 vector<int>::iterator i;
 // scorro gli elementi in ordine:
 for(i=v.begin(); i!=v.end(); i++){
    cout << *i; // stampo l'elemento
 }
}
```

- la funzione ***visualizza*** gli elementi di un vettore di ***interi***
- ***v.begin()*** restituisce un ***iteratore*** all'elemento di testa (***front element***)
- ***v.end()*** restituisce un ***iteratore*** che punta ad un immaginario elemento ***successivo all'ultimo***

funzione generica per vector

```
template<class T>
void stampa_vector(vector<T> v){
  typename vector<T>::iterator i;
  // scorro gli elementi in ordine:
 for(i=v.begin(); i!=v.end(); i++){
    cout << *i; // stampo l'elemento
 }
```

- quando si usa un'espressione generica a sinistra del "::", si rende necessaria la parola chiave ***typename***
- ***vincolo***: il tipo T deve avere il vincolo di essere ***stampabile*** (definito l'operatore **<<** su ostream)
- la funzione può essere invocata su vector di interi, caratteri, stringhe, etc. ma non su altri tipi che non abbiano ridefinito **<<**

funzione generica per ogni contenitore


```
template<class Cont>
void stampa_vector(Cont c){
  typename Cont::iterator i;
  // scorro gli elementi in ordine:
 for(i=c.begin(); i!=c.end(); i++){
    cout << *i; // stampo l'elemento
 }
```

- esempio di funzione generica che stampa gli elementi di un ***generico container***
- ***vincolo:***
  - ***Cont*** deve supportare gli iteratori quindi funziona con ***qualsiasi contenitore*** STL (vector, list, etc.)

esempio: iteratori su lista di interi

… da begin a end

```
int main() {
  list<int> first;
  list<int> second {34,23,65,12,3,67,123};
  cout << "first size = "
       << first.size() << endl;
  cout << "second size = "
       << second.size() << endl;
  for (list<int>::iterator
       i=second.begin();
       i!=second.end(); ++i) {
     cout << *i << endl;
  }
}
```

… da end a begin

```
int main() {
  list<int> l {34,23,65,12,3,67,123};
  cout << "contenuto di l ..." << endl;
  for (list<int>::iterator
       i=prev(l.end());
       i!=prev(l.begin()); i--) {
     cout << *i << ", ";
  }
  // o meglio così
  for (list<int>::reverse_iterator
       i=l.rbegin(); i!=l.rend(); i++) {
     cout << *i << ", ";
  }
}
```

altri esempi

```
vector<int> v;
v.push_back(1); // append an element
vector<int>::iterator it = v.begin();
// insert an element in third position
v.insert(it, 2);
```

```
int main() {
  vector<int> v {21,33,20,15,2,66,8};
  for (vector<int>::iterator
       i=v.begin(); i!=v.end();i++) {
    if (*i%2==1) {
       v.erase(i);
    }
  };
  return 0;
}
```

deque

```
int main() {
  deque<int> v {21,33,5,2,66,8,87};
  cout<< "contenuto di l ..." << endl;
  for (deque<int>::iterator
       i=v.begin(); i!=v.end(); i++) {
     cout << *i << ", ";
  }
  cout<<"remove "<< v.back()<< endl;
  v.pop_back();
  cout <<"<remove "<< v.front()<<endl;
  v.pop_front();
}
```

riferimenti per container con relativi metodi ***http://en.cppreference.com/w/cpp/container***

STL

algoritmi

algoritmi

- nella STL sono presenti numerosi ***algoritmi generici*** per eseguire le operazioni più comuni
- gli algoritmi sono **indipendenti dai contenitori**
- per l'utilizzo degli algoritmi della STL è necessario includere l'header ***<algorithm>***
- ***alcuni*** esempi di algoritmi:
  - operazioni sequenziali senza modifiche: ***find***, for_each
  - operazioni di modifica degli elementi: ***fill***, replace, copy
  - algoritmi di ordinamento: ***sort***

find

- individua il ***primo*** elemento che corrisponde al valore cercato

```
vector<int> v;
fillVector(v,10);
vector<int>::iterator it;
int rval;
rval = rand()%100;
it = find(v.begin(),v.end(),rval);
if (it != v.end())
   cout << rval << " found at pos "
        << it-v.begin() << endl;
else
   cout << rval << " not found"
        << endl;
```

replace

- ***sostituisce*** con un ***nuovo*** valore il contenuto degli elementi il cui valore coincide con quello specificato

```
int oldval,newval;
cout << "old value: ";
cin >> oldval;
cout << "new value: ";
cin >> newval;
replace(v.begin(), v.end(),
          oldval, newval);
```

copy

- ***copia*** gli elementi di una sequenza in un'altra
- nell'esempio v2 ha 20 elementi tutti con valore 99
- la dimensione di v2 è maggiore di quella di v1
- il ***valore*** dei primi elementi di v2 viene ***sostituito*** con quello degli elementi di v1

```
vector<int> v2(20,99);
printVector(v2);
copy(v.begin(),v.end(),v2.begin());
printVector(v2);
```

```
99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
copy(v.begin(),v.end(),v2.begin());
1 | 6 | 10 | 16 | 30 | 44 | 75 | 81 | 88 | 98 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 | 99 |
```

sort

- la funzione template ***sort*** realizza l'***ordinamento*** dei dati di un contenitore *in place* (nello stesso container di partenza)
- i parametri formali sono due ***iteratori*** che si riferiscono al ***primo*** e al ***successivo all'ultimo*** elemento della parte del contenitore da ordinare
  - per ordinare tutto il container si utilizzano ***begin*** e ***end***

sort – esempio vector

```
vector<int> v;
fillVector(v,10); printVector(v);
sort(v.begin(),v.end()); printVector(v);

void printVector(vector<int> v) {
   for (auto elem : v)
      cout << elem << " | ";
   cout << endl;
}

void fillVector(vector<int> &v, int n) {
   srand(std::time(nullptr));
   for (int i=0; i<n; i++)
     v.push_back(rand()%(n*10));
}
```

- gli iteratori begin e end si riferiscono al primo elemento del vector e al successivo all'ultimo

```
vector
71 | 1 | 97 | 98 | 98 | 88 | 46 | 74 | 96 | 70 |
sort(v.begin(),v.end());
1 | 46 | 70 | 71 | 74 | 88 | 96 | 97 | 98 | 98 |
```

sort – esempio array

```
const int DIM = 5;
int contArray[DIM];
fillArray(contArray,DIM);
printArray(contArray,DIM);
sort(contArray,contArray+DIM);
printArray(contArray,DIM);

void fillArray(int v[], int n) {
  srand(std::time(nullptr));
  for (int i=0; i<n; i++) v[i] = rand()%(n*10);
}

void printArray(int v[], int n) {
  for (int i=0; i<n; i++) cout << v[i] << " | ";
  cout << endl;
}
```

- la funzione ***sort*** opera anche con gli array
- i parametri sono i ***puntatori*** al primo e al successivo all'ultimo elemento dell'array
  - un array è definito tramite il puntatore al suo primo elemento

```
array
14 | 44 | 15 | 13 | 49 |
sort(contArray,contArray+DIM);
13 | 14 | 15 | 44 | 49 |
```

documentazione STL

- online
  - http://en.cppreference.com/w/
  - http://it.cppreference.com/w/
- offline
  - http://en.cppreference.com/w/Cppreference:Archives
- la documentazione sarà ***consultabile*** durante le sessioni di esame
  - in alcune prove sarà richiesto di ***non*** utilizzare le funzionalità messe a disposizione dalla STL