Standard Template Library

Stefano Lacaprara

Cosa sono le STL

- Insieme di classi e algoritmi di uso generale Standardizzate
  - Stringhe, vettori, mappe associative, liste, stack ...
  - Algoritmi per manipolarle (sort, shuffle, copy, ...)
- Sono disponibili come Librerie su tutti i compilatori
- Sono Template per essere piu' generali possibile (vediamo dopo)
- Le funzionalita' sono garantite, veloci, affidabili, ottimizzate.

std::string

- Partiamo dalla piu' semplice (e utilissima)
- #include <string>
- Rappresenta una stringa di caratteri, simile concettualmente a `char*` ma molto piu' potenti e semplici da usare.
- La rappresentazione interna NON ci interessa
  - (l'ha scritta uno bravo, molto piu' bravo di noi, e ci garantisce che funziona ed e' scritta bene)
- Ha una interfaccia molto estesa (vediamo qualcosa)

```
string(const char*); // costruttore
string s(“hello world”);
cout << s << endl;
s+=”!”;
s.size();
s.empty();
s.append(“ Oggi e' martedi' “);
s.erase(12); // cancella da pos 12 alla fine
s.erase(12,4); // cancella da pos 12 4 caratteri
s.replace(0,5,”ciao”); // sostituisci caratteri da 0 a 5 con “ciao”
int a = s.find(“o”); // trova la prima occorrenza di “o”
const char* cs = s.c_str(); // converte a char*
```

Riusare codice: Template

- (Una piccola parentesi)

- Partiamo da un esempio concreto:
- Voglio scrivere una funzione che calcoli il massimo di due numeri
- Niente di piu' facile: una riga di codice!

```
int getMax(int a, int b) { return a>b? a: b; }
```

E se voglio fare con float?

- Come prima

```
float getMax(float a, float b) {
    return a>b? a: b;
}
```

- E double? Idem
- E ...

Ma anche...

- Posso aver definito una classe che e' in grado di avere un ordinamento
- Ovvero, la classe ha un `operator<(...)`
- Per esempio potrei farlo per i miei Complex
-
-
- Ordino i Complex a seconda del loro modulo

```
bool operator<(Complex a, Complex b) {
    return a.mod()<b.mod()? true : false;
}
```

Se voglio fare getMax tra Complex

- Riscrivo il codice di prima

```
Complex getMax(Complex a, Complex b) {
    return a>b? a: b;
}
```

Che cosa cambia?

- Ho scritto 3 diverse funzioni che fanno la stessa identica cosa! Non e' furbo.
- L'unica cosa che cambia e' l'oggetto (int, float, Complex) che viene maneggiato
- Qualunque sia l'oggetto, la funzione fa le stesse cose, nel ns esempio chiamare `operator<(...)`
- Poi ogni classe implementa l'operatore come crede

Template

- Posso dire al C++ di implementare una funzione con un tipo generico, e far scrivere le implementazioni con i tipi concreti al compilatore
- Si usano i Template (“prototipo”)
- Vediamo come ...

```
template <T>
T getMax (T a, T b) {
   return (a>b?a:b);
}
```

- Come si usa?

```
int a = getMax(1,2); // uso getMax con int
float b = getMax(1.,2.); // getMax con float
Complex c = getMax(Complex(1,1),
                   Complex(2,3)); // con Complex
Pettegolo p = getMax(pettegolo1, pettegolo2);
// con Pettegolo, posto che esista Pettegolo::operator<()
```

Economicita'

- Con un sola funzione faccio generare al compilatore tutta una classe di funzioni a seconda di come uso la funzione nel mio codice
- Se uso GetMax(int, int) compilatore genera funzione con int
- Se con float, idem
- Se non uso double, il compilatore non genera funzioni

Altro esempio: classi contenitori

- Ad un contenitore non server sapere cosa contiene
- Interfaccia minima
  - Inserisci oggetto
  - Rimuovi oggetto
  - Ritorna oggetto numero N
  - Ritorna numero oggetti
- Non mi interessa sapere se contiene int, float, Complex o Pettegoli, deve fare sempre le stesse cose

Classe Pair (coppia)

- Una classe che contiene due oggetti di un certo tipo

```
class Pair {
  public:
    Pair (int a, int b) : a_(a), b_(b) { }
    int first() const { return a_;}
    int second() const {return b_;}
  public:
   int a_,b_;
};
```

E se double, o complex?

- Come prima, dovrei scrivere una class Pair per int, una per double, una per Complex, etc...
- Oppure uso template

```
template <typename T> class PairSame {
  public:
    PairSame(T a, T b): a_(a), b_(b) { }
    T a() const { return a_; }
    T b() const { return b_; }
  private:
    T a_;
    T b_;
};
```

Ancora piu' generale

- Un Pair che contiene due oggetti di tipo diverso

```
template <typename T1, typename T2> class PairSame {
  public:
    PairSame(T1 a, T2 b): a_(a), b_(b) { }
    T1 a() const { return a_; }
    T2 b() const { return b_; }
  private:
    T1 a_;
    T2 b_;
};
```

Ma...

- Devo fare io tutta la fatica di scrivere codice generale come questo???
- NO
- Uso le STL
  - #include <utility>
  - pair<int, Complex> coppia(1,Complex(3.,4.));
  - int i = coppia.first;
  - Complex c = coppia.second;

Standard Template Library

- Al solito, c'e' chi ha scritto un sacco di (ottimo) codice che noi possiamo semplicemente usare
- Documentazione su http://www.sgi.com/tech/stl/
  - Container, algoritmi, coppie, ...
  - Non si scrive C++ senza le STL
-

vector<T>

- Contenitore universale di elementi di tipo T
  - T puo' essere qualunque cosa
- #include <vector>
  - Tutte le STL sono nel namespace *std*
  - *using namespace std;*
- vector<T>
  - Contenitore per qualunque tipo di oggetti
  - Contenitore di dimensione variabile
  - Efficente come o piu' di array

Vector<T>

```
vector<int> v; // creo un vettore di int (vuoto)

v.push_back(1); // inserisco “1” dentro v
v.push_back(3); // inserisco “3” dentro v

int num = v.size() // quanti elementi contiene v (2)

cout << v[0] << endl; // accedo al primo elemento di v (==1)
cout << v[1] << endl; // secondo elemnto (==3)

cout << v[2] << endl; // ERRORE (ho solo due elementi!)
```

Ciclo (I)

- Per accedere a tutti gli elementi

```
// dopo aver riempito il vettore

for (int i ; i<v.size(); ++i) {
   cout << i << " " << v[i] << endl;
}
```

Ciclo (II)

- Uso “iteratori”
- Come se fossero puntatori agli elementi
- vector::begin() ritorna iteratore a primo elemento
- vector::end() iteratore a “fine+1”
- Si va avanti con ++, +N

```
vector<Complex> vc;
// dopo aver riempito il vettore

for (vector<Complex>::const_iterator v = vc.begin();
      v!=vc.end() ; ++v)   cout << *v << endl;
```

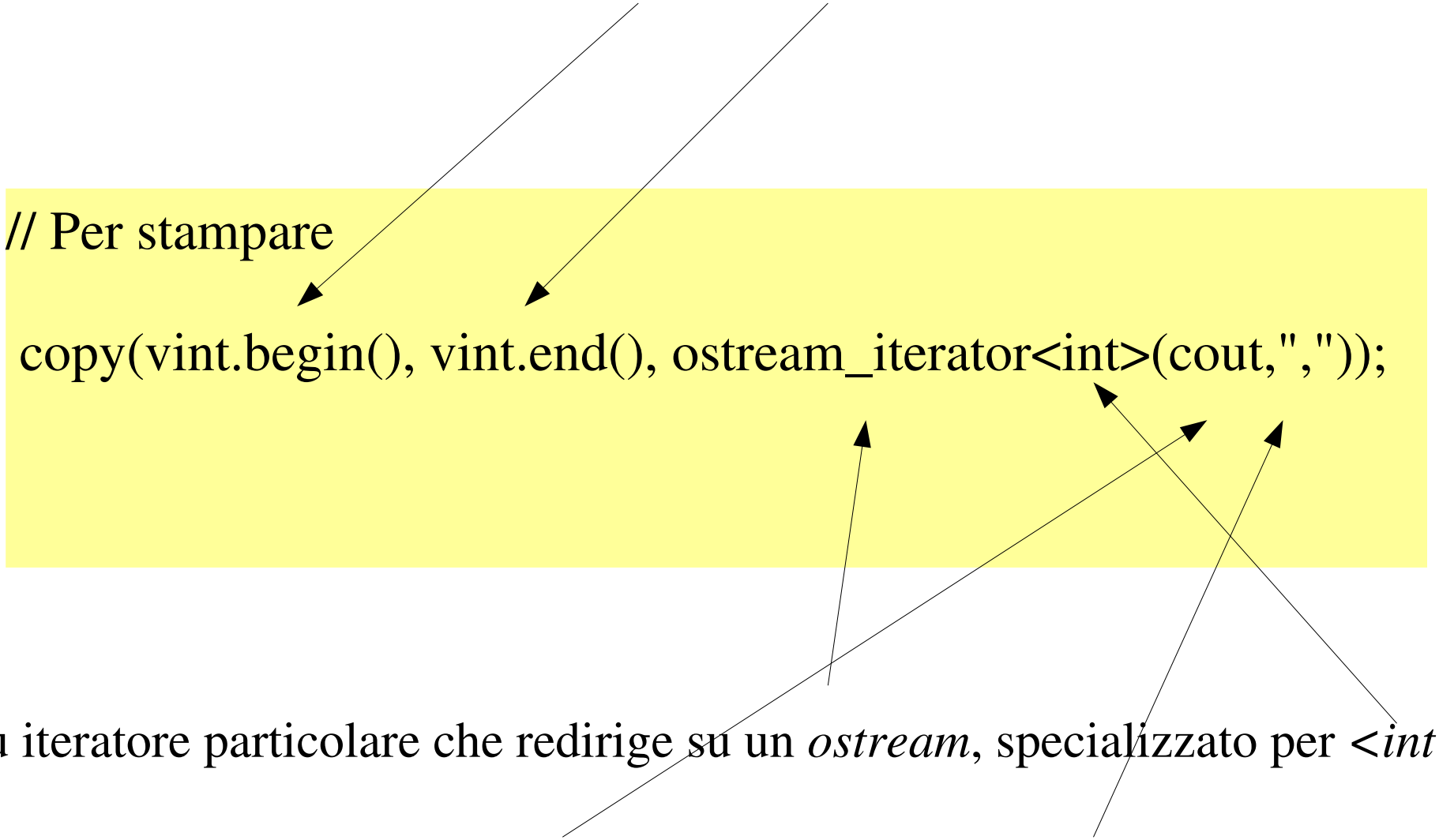
Molto di piu'

- Posso usare “algoritmi”
- #include <algorithm>
  - sort(inizio, fine);
  - random_shuffle(inizio, fine);
  - count(...)
  - count_if(...)
  - copy(...)
  - unique(...)
  - for_each(...)
  - .....

```
vector<int> vint;

// riempio vint

sort(vint.begin(), vint.end()); // ordino

random_shuffle(vint.begin(), vint.end()); // disordino

int n=count(vint.begin(), vint.end(), 5); // quanti 5 ci sono?

vector<int> altro(vint.size()); // creo vettore di dimensione
vint.size()

copy(vint.begin(), vint.end(), altro.begin()); // copio vint in altro
```

Copio vettore vint da inizio a fine

```
// Per stampare
copy(vint.begin(), vint.end(), ostream_iterator<int>(cout,","));
```



Su iteratore particolare che redirige su un *ostream*, specializzato per <*int*>

Come *ostream* gli passo *cout* (schermo) e aggungo una “,” dopo ogni volta

Predicati

- Sono funzioni che prendono uno (o piu') argomenti e ritornano un bool (vero/falso)
- Possono essere definite a vs piacimento e passate ad alcuni algoritmi: es
  - count_if // conta quante volte il predicato e' vero

```
bool less_than_7(int i) {
  return i<7;
}

int main() {
vector<int> vint;

// riempio vint

// quanti sono i numeri minori di 7
int n=count_if(vint.begin(), vint.end(), less_than_7);

}
```

Funtori

- Funzioni che agiscono sugli elementi di un container e ritornano void
- Utili se vogliamo che un algoritm esegua nostro codice
  - Li definisco (come credo)
  - Poi li passo come argomento ad un algoritmo

```
void print(int i) {
  cout << “Il mio valore e' “ << i <<endl;
}

int main() {
vector<int> vint;

// riempio vint

// eseguo “print(int)” per ogni elemento del vector
for_each(vint.begin(), vint.end(), print);

}
```

Map

- E' un container associativo
- Cioe' associa agli elementi una chiave
  - Tipico esempio elenco del telefono
  - Nome -> numero
- #include <map>
- E' ordinata (quindi la chiave deve essere di un tipo per cui esiste *operator<)*
- C'e' un solo elemento per una data chiave

```
#include <map>
#include <string>

int main() {
map<string, int> elenco;

elenco[“Stefano”]=3491234567
elenco[“Marco”]=3499876543
}
```

- Si possono usare loop e algoritmi esattamente come per vector
- In piu', si puo accedere per chiave
- Int num=elenco[“Stefano”]; // nb chiave deve esistere!

```
#include <map>
#include <string>

int main() {
map<string, int> elenco;

elenco[“Stefano”]=3491234567
elenco[“Marco”]=3499876543

  for (map<string, int>::const_iterator e=elenco.begin();
         e!=elenco.end(); ++e) {
        cout << "Il telefono di " << e->first << " e': "
                << e->second << endl;
  }
}
```

map contiene pair!

Ricerca

- iterator i=elenco.find(key);
  - Se non trova, ritorna end() (fine +1)
  - Se trova, torna iteratore che punta a elemento
  - Devo sempre controllare che i!=elenco.end()

Esercizi

- Generalizzate i Complessi in modo che parte reale e complessa possano essere int, float, double, etc (template)
- Modificate Pettegoli in modo che Popolazione usi un vector<Pettegoli> invece di una array
- Scrivete una classe che rappresenta un elenco del telefono, con le funzionalita' che vi sembrano utili (stampa, ricerca, ordinamento, inserimento, cancellazione, ...)