Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

3 Novembre 2014

Programmazione Generica

- **Polimorfismo attraverso l'ereditarietà:** Attraverso le classi, esiste la possibilità di definire nuovi tipi. I membri di ogni classe possono essere sia dati che funzioni e solo alcuni di essi possono essere accessibili dall'esterno(*data hiding)* .Ogni nuovo tipo può essere corredato di un insieme di operazioni (overload degli operatori) e ulteriormente espanso grazie all'ereditarietà. Attraverso l'ereditarietà virtuale si può realizzare un codice polimorfo

- **Polimorfismo attraverso la "programmazione generica",** la quale consente di applicare lo stesso codice a tipi diversi, cioè di definire **template** (ovvero dei **modelli**) di classi e funzioni parametrizzando i tipi utilizzati:
  - nelle classi, si possono parametrizzare i tipi dei dati-membro;
  - nelle funzioni (e nelle funzioni-membro delle classi) si possono parametrizzare i tipi degli argomenti e del valore di ritorno.

Programmazione Generica

- In questo modo si raggiunge il massimo di indipendenza degli algoritmi dai dati a cui si applicano: per esempio, un algoritmo di ordinamento può essere codificato una sola volta, qualunque sia il tipo dei dati da ordinare.

- sono di grande utilità in quanto consente di scrivere codice "generico". Ciò è particolarmente vantaggioso nel creare classi strutturate identicamente, ma differenti solo per i tipi dei membri e/o per i tipi degli argomenti delle funzioni-membro. Il discorso è analogo per le funzioni

- Il compilatore genera dal template tutte le funzioni/classi concrete, a seconda di quanto richiesto nel programma

Classi Template

- Una classe template è identificata dalla presenza, davanti alla definizione della classe, dell'espressione:

```
template < class T >
```

- dove T rappresenta il parametro, esprimente il tipo generico che verrà utilizzato nella dichiarazione di uno o più membri della classe.

- la *parola-chiave* class non ha qui il significato usuale: indica che T è il nome di un tipo (anche *nativo*), e non necessariamente di una classe

Classi Template

- I parametri di un template possono anche essere più di uno, nel qual caso, nella definizione della classe e nelle definizioni esterne delle sue funzioni-membro, tutti i parametri vanno specificati con il prefisso class e separati da virgole. Esempio:

```
template  < class T1,classT2, class T3 >
```

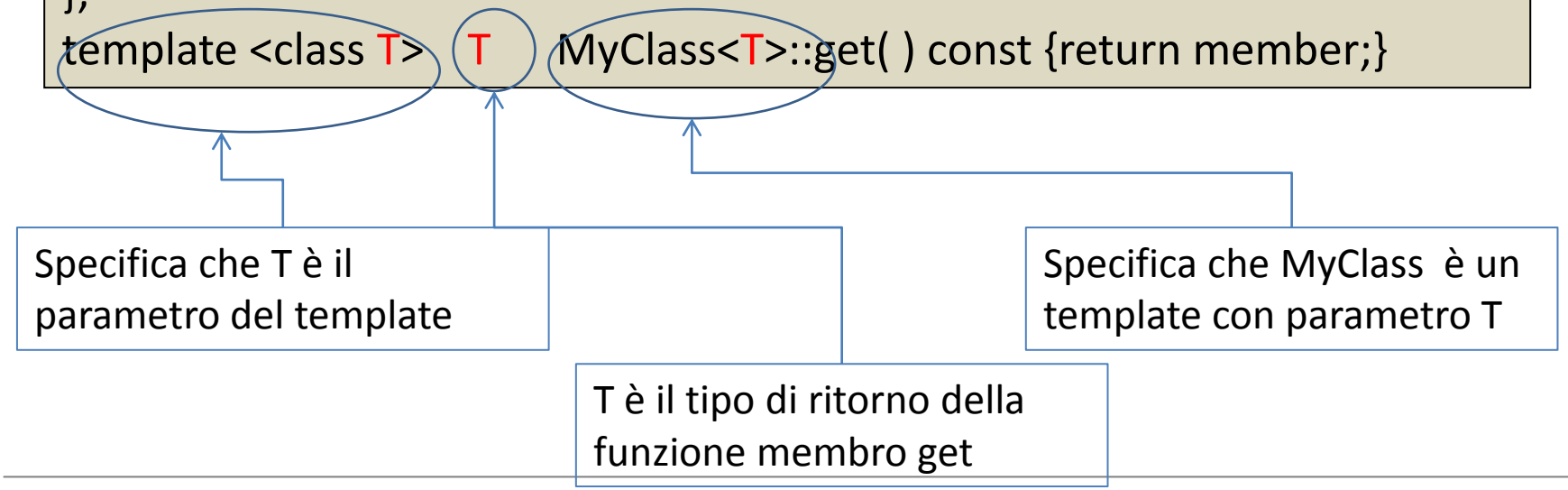
- I template non possono essere definiti nell'ambito di un blocco. Vanno sempre definiti all'interno di un namespace, o nel namespace globale o anche nell'ambito di un'altra classe (templated o no). Si può far ereditare una classe template da un'altra classe (non necessariamente templated).

- Non è inoltre ammesso definire nello stesso ambito due classi con lo stesso nome, anche se hanno diverso numero *di parametri* oppure se una classe è template e l'altra no (in altre parole l'overload non è ammesso).

Classi Template

- Esempio: Classe con un unico attributo, e un getter dell'attributo. Il tipo dell'attributo, così come il valore di ritorno del getter è parametrico:

```
template<class T>  class MyClass{
public:
MyClass(const T& m): member(m){}
T get( ) const;
private:
T member;
};
template <class T>   T    MyClass<T>::get( ) const {return member;}
```



Classi Template

Esempio: tipo intero e tipo circle

```
template<class T> class MyClass{
public:
MyClass(const T& m): member(m){}
T get( ) const ;
private:
T member;
};
template <class T> T MyClass<T>::get( ) const {
return member;
}
```

```
#include<iostream>
#include"shapes.h"
using namespace std;
int main(){
int a=3;
MyClass<int> A(a);
int b= A.get();
cout << b <<endl;
circle C(1,3, 3);
MyClass<circle> A(C);
circle B= A.get();
B.print()<<endl;
return 0;
}
```

Program output:

```
 3
Circle:[3,3], Radius=1
```

Funzioni Template

- Analogamente, le funzioni template sono funzioni generiche che hanno come parametro il tipo di argomento:

```
template<class T> ret_Type function_Name( argType(T ));
```

Dove T rappresenta un tipo generico che verrà utilizzato come argomento della funzione, ret_Type è il tipo di ritorno e function_Name è il nome della funzione

- Nel caso in cui si debbano specificare più di un tipo parametrico, si fa con la seguente sintassi (es. per 2 argomenti)

```
template<class T1, class T2> ret_Type function_Name(argType(T1) , argType(T2));
```

- Il tipo parametrico può essere anche riferito al valore di ritorno

Funzioni Template

Esempio: massimo
di un array
con casting di tipo nel return

```
template<class T1, class T2> T2 max( T1 *v, int size){
T1 max=v[0]; for(int i=1;i<size;i++)
if(max<v[i]) max=v[i];
return (T2) max;
}
```

```
int main(){
int main(){ int arrayI[7]={4,7,2,4,9,3,2}; double
arrayD[6]={7.1,9.4,2.6,5.7,4.8,6.9}; cout <<
max <int, int>(arrayI,7) <<endl; cout << max
<double, double>(arrayD,6) <<endl; cout <<
max <double, int>(arrayD,6) << endl; return 0;
}}
```

Program output:
9
9.4
9

Funzioni Template

- Template function per ordinare un array A di n elementi di tipo generico T:

```
//template function for sorting (insertion sort algorithm, increasing order)
template <class T> void Sort (T* A, int n){
 int i,k;
for (int i=1; i<n; i++){
  T x=A[i]; k=i; k--;
  while (k >= 0 && A[k]>x){
    A[k+1] = A[k];
    k--;
  }
  A[k+1]=x;
}
}
```

```
int main() {
 int arrayI[7]={4,7,2,4,9,3,2};
 double arrayD[6]={7.1,9.4,2.6,5.7,4.8,6.9};
Sort(arrayI,7);
Sort(arrayD,6);
return 0;
}
```

- Se l'argomento è di tipo definito dall'utente, la classe che corrisponde a T dovrà ovviamente anche contenere tutti gli overload degli operatori necessari per eseguire confronti e scambi fra gli elementi dell'array.

La Standard Template Library

- La stessa Libreria Standard del C++ mette a disposizione strutture precostituite di classi template. In particolare, La Standard Template Library fornisce:
- **container**, che rappresentano le strutture dati di base;
- **iteratori**, che generalizzano il concetto di puntatore C/C++;
- **algoritmi** generici che operano sui container attraverso iteratori
- Combinazioni di algoritmi, container e iteratori: realizzazione di componenti di alto livello in termini di efficienza e utilizzabilità

I Container STL

• Concettualmente, i container sono degli oggetti che contengono altri oggetti. Usano certe proprietà di base degli oggetti (la possibilità di copiarli, ordinarli, etc.) ma altrimenti non dipendono dal tipo di oggetto che contengono

•Tutti i container sono oggetti di tipo "templated", per cui è possibile utilizzarli con qualunque tipo di dati. I "container" predefiniti nel linguaggio (cioè gli arrays) possono essere usati come container STL, beneficiando delle funzionalità della STL (algoritmi come sort etc)

- **Container Sequenziali:** Collezioni lineari di oggetti dello stesso tipo T (vector, list,deque)
- **Container Associativi:** Generalizzazione delle sequenze STL. Gli elementi sono indicizzati non da interi ma da un qualunque tipo generico (set, map)

I Container STL

Container Sequenziali:

- vector: proprietà principali sono inserzione/rimozione in coda, e accesso diretto a qualunque elemento in tempo costante. Si allunga automaticamente se richiesto.
- deque: come vector, con accesso diretto a qualunque elemento ma possibilità di inserzione/rimozione alla fine e all'inizio
- list: inserzione/rimozione efficente in qualunque punto della sequenza, ma è possibile solo l'accesso sequenziale (e necessita di iteratori bidirezionali)

I Container STL

Per la scelta del container si deve tenere conto di vari fattori:

•vector, deque sono rappresentati in modo contiguo in memoria. Usano iteratori di tipo random access;

•list (e tutti gli associative container) memorizzano gli elementi in locazioni di memoria non consecutive e manipolano solo gli indirizzi. Per questi container si usano iteratori bidirezionali.

• inserimento e cancellazione hanno “effetti collaterali” (copie, swap) maggiori nei sequence container (a parte list) rispetto agli associative container, per il vincolo della rappresentazione contigua in memoria.

I Container STL - vector

- Alcune funzioni membro principali:
  - size() numero di elementi
  - push_back(element) inserisce elemento in coda (no push_front())
  - pop_back(element) rimuove elemento in coda (no pop_front())
  - empty() vera se vector è vuoto
  - clear() cancella tutti gli elementi dell'array
  - at(n) ritorna l'elemento alla posizione n (con bound checking)

- Operatori di base:
  - = sostituisce tutti gli elementi di un vettore a quelli di un altro vettore
  - == confronto elemento per elemento fra vectors
  - [n] accesso diretto all'elemento n (no bound checking)

- Accesso diretto, utilizza random iterators

I Container STL - list

- Alcune funzioni principali:
  - size() numero di elementi
  - push_back(element) e push_front()
  - pop_back(element) e pop_front()
  - empty() vera se lista è vuota
  - clear() cancella tutti gli elementi

- Operatori di base:
  - = sostituisce tutti gli elementi di una lista a quelli di un altra lista
  - == confronto elemento per elemento

- Accesso agli elementi: sequenziale, solo attraverso iteratori bidirezionali (no at() o [ ])

- Metodi specifici: sort, reverse,remove,unique,....

I Container STL

- Per sapere se il container cnt è vuoto:

```
if(cnt.empty())... // non if(cnt.size()==0), lento!
```

- Per rimuovere elementi con valore 16.5 dal container cnt si usa una combinazione dell' algoritmo generico remove e poi la member function erase:

```
cnt.erase(remove(cnt.begin(), cnd.end(), 16.5), cnt.end());
```

- quando cnt è una list, è meglio usare la member function specializzata:

```
cnt.remove(16.5); // cnt is a list
```

Gli Iteratori STL

- Gli ***iteratori*** sono oggetti pointer-like utilizzati per attraversare in sequenza i container. Sono indirizzatori per container in maniera analoga a come int* può essere usato come indirizzatore di un array di interi.

- La STL definisce diversi tipi di iteratori a seconda del tipo di container. Per vector e deque, sono random access. Per gli altri container, sono bidirezionali. Accessibili attraverso l'include <iterator>

- Di solito si scorre una sequenza incrementando un iteratore in un range **[first, last)**. Se **first==last** il range è vuoto. Ogni container ha le member function begin() e end() che ritornano gli iteratori di inizio e fine sequenza (la locazione immediatamente successiva all'ultimo elemento).

Gli Iteratori STL

- *random access iterator*: i test di uguaglianza (**==**) e diseguaglianza (**!=**), la dereferenziazione (*****), l'incremento prefisso e postfisso (**++**). più le seguenti funzionalità, che li rendono assolutamente omologabili ai puntatori:
  - l'operatore di accesso diretto **[ ]** (**r[n]** significa ***(r+n)**);
  - l'addizione o la sottrazione di un intero (**r+n**, **n+r**, **r-n**) e le operazioni **r+=n** e **r-=n**;
  - la differenza fra random access iterators (**r-s**, che produce un risultato intero);
  - i confronti (**r<s**, **r>s**, **r<=s**, **r>=s,** **==**che producono risultato **bool**).
- Funzioni utili per la manipolazione di iteratori (in particolare se non di tipo random access), sempre definiti nell'header <iterator.h>:
  - advance(it, n): incrementa it di n posizioni (it++ n volte). Se n<0 come it– n volte)
  - distance(it1,it2) numero di posizioni tra it1 e it2.

Algoritmi Generici

• La STL fornisce vari algoritmi di uso generale (ad esempio, **find**, **sort**) che si possono usare con tutti i container che supportano iteratori del tipo richiesto dall'algoritmo

Accessibili attraverso l'include <algorithm>

buona "guida" online: http://www.cplusplus.com/reference/algorithm

• In alcuni casi, sebbene l'algoritmo generico si possa utilizzare con un dato container, le sue prestazioni non sono ottimali. In tal caso, il container definisce una member function con lo stesso nome, che ha prestazioni migliori. Ad esempio:

- **list** ha una member function **sort**, diversa dall'algoritmo generico sort (che opera su container con random access iterators) e che non si può usare con **list**
- l'algoritmo generico **find** ha tempo lineare (è sequenziale), mentre gli associative containers hanno una member function **find** con prestazioni migliori, con tempo logaritmico;

Algoritmi Generici

Alcuni esempi:

-find (it_first, it_last, value), trova l'elemento = valore in un range, ritorna un iteratore

-for_each(it_first, it_last, operation), esegue operation su ciascun elemento in un range,

– **random_shuffle(it_first, it_last)** riorganizza la sequenza secondo un ordine random in un range;

– **remove(it_first, it_last, value)** rimuove dalla sequenza tutti gli elementi con valore value (non cambia la dimensione, ritorna un iteratore all'ultimo elemento dopo la rimozione);

– **transform(it_first, it_last, it_first2, it_result,operation)** applica un'operazione(unaria/binaria) agli elementi di uno o due range e scrive il risultato in un altro range; restituisce un iteratore.

– **sort(it_first, it_last, compare)** ordina gli elementi di un container in ordine crescente, secondo l'operatore < o secondo il function object definito da compare.

Algoritmi Generici-Oggetti Funzione

- Molti algoritmi STL hanno una versione che accetta una funzione come parametro.

  Di solito si tratta di **predicati**, ovvero funzioni che restituiscono un bool, ma che possono svolgere anche altre operazioni. Un modo conveniente per passare una funzione consiste nel passare un **function object**: un **function object** è qualcosa che può essere applicato a zero o più argomenti e restituisce un valore o modifica lo stato della computazione.

- Le funzioni ordinarie C e C++ soddisfano tale definizione, ma anche ogni classe che ridefinisca il function call operator, operator(). Ad esempio:

```
class multiply {
public:
int operator()(int x, int y){return x*y;}
};
```

  - La STL fornisce templated function object per tutte le operazioni aritmetiche del linguaggio, per i confronti, operazioni logiche

Algoritmi Generici -Esempio

Si sostituisce con transform il contenuto di un array con i quadrati dei suoi elementi originari:

```
#include<iostream>
#include<algorithm>
#include<functional>
using namespace std;
int main(){
int x[10]; for (int i=0; i<10; i++) x[i]=i;
transform(&x[0],&x[10],&x[0],&x[0], multiplies<int>());
for(int i=0;i<10;i++)cout  << x[i] << endl;
return 0;
}
```

(gli algoritmi generici possono essere anche applicati agli array “convenzionali” del linguaggio)

Algoritmi Generici-Esempio

- Esempio di Ordinamento:

```
#include <algorithm> // sort
#include <functional> // greater
using namespace std;
// ...
sort(&a[0], &a[1000]); // ordine crescente
sort(&a[0], &a[1000], greater<int>()); // ordine decrescente.
```

- Nel secondo caso, sort crea un ordine decrescente, grazie all‘ oggetto predicato greater<T>() che incapsula l'operatore “>” del tipo T in un function object

Esempio-uso di vector/list

- Esempio dell'uso dei container vector e list, con tipi nativi e con oggetti:

vector_list.cpp