Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 7

Luca Formaggia

MOX
Dipartimento di Matematica "F. Brioschi"
Politecnico di Milano

A.A. 2006/2007

Classi amiche

Templates
- Deduzione degli argomenti
- Overloading di funzioni templates

Classi template
- specializzazione
- specializzazione di funzioni template

templates per specificare policies
- organizzazione dei files

Classi friend

Abbiamo visto come *funzioni* possano essere definite `friend` di una classe.
È u=in realtà possibile dichiarare `friend` una classe:

```
class A{
friend class B;
private:
double a;...
```

Tutti i metodi della classe `B` possono accedere ai membri protetti e privati di `A`.

Templates

Il C++ ci permette di definire variabili, funzioni ed altre entità usandi dei tipi specifici.
Tuttavia ci sono molti algoritmi che funzionano in modo analogo per tipi diversi.
Una funzione `sort(vector<int>&)`, che ordina un vettore (di interi in questo caso, in realtà richiede solo che sugli oggetti su cui opera sia definita una relazione d'ordine, espressa tramite l'operatore < (per esempio). Il codice sarà sostanzialmente identico in tutti i casi.

Templates

L'*overloading* di funzioni può essere utilizzato in questo caso allo scopo di generalizzare la funzione, ma questo obbliga a replicare il codice , con evidenti sprechi e problemi di manutenzionedel codice stesso.
Sarebbe utili potre trattare anche il tipo come un argomento della funzione.

In C++ questa possibilità viene fornita della *funzioni template* (template functions).

Un primo esempio

Scegliamo un caso semplice: la funzione `max(T & a, T&b)` che torna il massimo tra due oggetti `a` e `b` di tipo generico `T`, per cui si richiede solo che sia definito l'operatore `<`.

```
template <typename T>
inline T const& max (T const& a, T const& b)
return a < b ? b : a;
```

```
template <typename T>
inline T const& max (T const& a, T const& b)
{return a < b ? b : a;
}
```

Questa è un esempio di definizione di una template function, la cui sintassi generale è

```
template < lista di parametri separati da virgola >
return-type nome (argomenti){
corpo della funzione}
```

Un parametro della forma typename T definisce il simbolo `T` come un tipo generico, che può essere richiamato sia negli argomenti che nel corpo della funzione.

Typename o class?

Al posto delal parola chiave `typename` si può alternativamente usare `class`:

```
template <class T>
inline T const& max (T const& a, T const& b)
{return a < b ? b : a;
}
```

Questo non vuole dire che `T` debba essere una classe e non un tipo primitivo. Infatti le due dizioni sono assolutamente equivalenti.

Uso di una template function

Supponiamo che il codice precedente sia contenuto nel file `max.hpp`.

```
#include <string>
#include ”max.hpp”
int main(){
int i = 42;
std::cout « ”max(7,i): ” « ::max(7,i) « std::endl;
double f1 = 3.4;
double f2 = -6.7;
std::cout « ”max(f1,f2): ” « ::max(f1,f2) « std::endl;
std::string s1 = mathematics;
std::string s2 = math;
std::cout « ”max(s1,s2): ” « ::max(s1,s2) « std::endl;
}
```

Quando il compilatore incontra, per esempio, `max(f1,f2)` dal tipo degli argomenti ricava che deve istanziare la funzione `max<double>()`.
Solo in quel momento procede alla compilazione di tale funzioneche ha gli stessa semantica di

```
template <class T>
const& max (double const& a, double const& b)
{return a < b ? b : a;
}
```

Nota: Nell'esempio precedente abbiamo fatto uso dello *scope resolution operator* `::` per indicare che vogliamo usare la funzione `max` definita nello *scope* globale. Questo per evitare ambiguità con la funzione `max` definita nella Standard Library.

Deduzione degli argomenti

Al momento della chiamata di una funzione template il valore attuale del parametro temmplate (template parameter) viene risolto sulla base del tipo degli argomenti. Situazioni ambigue non vengono risolte automaticamente:

```
double a; int b; max(a,b);
```

è ambiguo. Si può risolvere in due modi differenti

```
max(a,static_cast<double>(b));//usa max<double>
max<double>(a,b);//usa max<double>
```

Nel secondo caso si qualifica esplicitamente il parametro template e ci si affida alla conversione implicita `int` $\longrightarrow$ `double`.
Ovviamente è possibile anche fare
`max<double>(s,static_cast<double>(b))`.

Parametri template

Le funzioni template hanno due tipi di parametri

- I parametri template *(template parameters)* veri e propri, che sono identificati dal costrutto `template <typename T1,typename T2..>`.
- Gli argomenti template *(call templates)* che sono quelli che appaiono negli argomenti della funzione:

  ```
  ... max(T const &,T const &)
  ```

I template parameters possono essere in numero arbitrario e possono comparire sia come argomenti, sia nel corpo della funzione. Nel momento dell'uso della funzione il tipo degli argomenti attuali permettono di risolvere i corrispondenti parametri.

Un esempio più complesso

```
template<typename RT, typename T> inline RT max(T const
& ,T const &){
return static_cast<RT> a < b ? b : a;}
```

In questo caso il primo parametro deve essere specificato. Per esempio

```
max<float>(4,5)
```

viene istanziato come `max<float,int>`.

Overloading di funzioni template

```
int const& max (int const& a, int const& b) {...}
template <typename T>
T const& max (T const& a, T const& b){...}
template <typename T> T const& max (T const& a, T const&
b, T const& c){...}
int main(){
::max(7, 42, 68); // chiama la funzone con 3 argomenti
::max(7.0, 42.0); // chiama max<double>
::max('a', 'b'); // calls max<char>
::max(7, 42); // chiama la funzione max()
::max<>(7, 42); // chiama max<int>
```

Un esempio più utile

```
#include<cstring> // massimo di due valori di ogni tipo
template <typename T>
T const& max (T const& a, T const& b)
{
return a < b ? b : a; }
// massimo di due puntatori
template <typename T>
T* const& max (T* const& a, T* const& b){
return *a < *b ? b : a; }
// massimo di 2 stringhe C
char const* & max (char const* & a, char const* & b) {
return std::strcmp(a,b) < 0 ? b : a; }
```

Una buona regola

Se si voule implementare l'overloading di funzioni template evitare cambiare più del necesario. Limitarsi a cambiare il numero di parametri o a specificare esplicitamente alcuni parametri.

Altrimenti si possono avere dei risultati inaspettati.

Dichiarazione e definizione

Anche per le funzioni template si può separare la dichiarazione

```
template <typename T>
T const& max (T const& a, T const& b);
```

dalla definizione, che contiene anche il corpo della funzione.

```
template <typename T>
T const& max (T const& a, T const& b){
return a < b ? b : a; }
```

Tuttavia, a causa del ruolo particolare che la *istanzazione* per una funzione template, l'organizzazione della unità di compilazione è qui diversa. Se ne discuterà più avanti.

Classi template

Il concetto di template si estende anche alle classi. Un template di una classe (*class template*) ha una dichiarazione della forma:

```
template <typename T>
class A{
dichiarazione dei membri, qui T è un tipo generico
};
```

Una istanza di `A<T>` è detta *classe template* (*template class*): `A<double> pippo`. L'oggetto (`pippo`) è detto *oggetto template* (*template object*).
Purtroppo in Italiano la differenza semantica tra *class template* e *template class* non viene rese in modo altrettanto efficace.

```
template<typename T> class Vcr {
  int lenth;              // number of entries in vector
  T* vr;                  // entries of the vector
public:
  Vcr(int, T*);           // constructor
  Vcr(const Vcr&);        // copy constructor
  ~Vcr();                 // destructor

  int size();
  inline T& operator[](int i);
  Vcr& operator=(const Vcr&);   // overload =
  Vcr& operator+=(const Vcr&);  // v += v2
  Vcr& operator-=(const Vcr&);  // v -= v2
  T maxnorm () const;           // maximum norm
  T twonorm() const;            // L-2 norm

  template<class S>
  friend S dot(const Vcr<S>&, const Vcr<S>&);// dot product
};
```

Metodi di una classe template

I *metodi* di una classe template sono a tutti gli effetti delle funzioni template. Quello che li caratterizza rispetto ai metodi di una classe ordinaria (oltre al possibile utilizzo dei tipi generici parametri della classe template) è il fatto che vengono istanziati solo al momento dell'uso effettivo.

```
Vcr<double> a; ...
a.maxnorm();
```

il compilatore istanzierà `Vcr<double>::maxnorm()` ma non, per esempio, `Vcr<double>::twonorm()`. Questo aspetto è importante e verrà discusso in seguito.

Utilizzo di una classe template

```
Vcr<double> a,p;
Vcr<int> b;
Vcr<float *> z;
Vcr<Vcr<double> > c;//si noti lo spazio tra le >
double h=dot(a,p);// usa dot<double>
```

Definizione dei membri

```
template<typename T>
Vcr<T>::Vcr(const Vcr & vec) {
vr = new T [lenth = vec.lenth];
for (int i = 0; i < lenth; i++) vr[i] = vec[i];
}
template<typename T> Vcr<T>& Vcr<T>::operator=(const Vcr&
vec) {
if (this != &vec) {
if (lenth != vec.lenth ) error(bad vector sizes);
for (int i = 0; i < lenth; i++) vr[i] = vec[i];
}return *this;}
```

All'interno dello *scope* della classe e dei suoi membri si può omettere il <T> per specificare `Vcr<T>`.
Si può tuttavia essere espliciti.

Definizione della funzione dot

```
template<typename S>
S dot(const Vcr<S> & v1, const Vcr<S> & v2) {
if (v1.lenth != v2.lenth ) error(bad vector sizes);
S tm = v1[0]*v2[0];
for (int i = 1; i < v1.lenth; i++) tm += v1[i]*v2[i];
return tm; }
```

Qui il <S> in Vcr<S> non può essere omesso, in quanto dot non è un membro della classe.

Nota: S è qui una variabile muta, avrei potuto benissimo scrivere template<typename T>.

Parametri template... parte seconda

I parametri di un template non devono necessariamente essere tipi. Possono anche essere dele costanti intere (più precisamente *integral constant expressions*: int, enum, char...)

```
template <typename T, int NDIM>
class Point{
private:
T data[NDIM];
... }; ...
Point<double,3> p;//un punto di R^3.
```

Attenzione `int n;... Point<double,n> p;` è un errore perchè `n` non è qui una costante.

Parametri di default

In un template di una classe è possibile specificare dei parametri template di default (non è possibile per le funzioni template). Si segue la solita regola da destra verso sinistra: se un parametro non ha un valore di default quelli alla sua sinistra NON possono averlo.

```
template <typename T, int NDIM=2>
class Point{
...
}; ...
template <typename T=double> Vcr{
...}
```

In questo caso `Point<int> a;` istanzia un `Point<int,2>` mentre `Vcr p;` un `Vcr<double>`.

Prerequisiti sui parametri template

L'utilizzo di funzioni o classi template presuppone che al momento della istanza le operazione nel corpo della funsione e dei membri istanziati abbia senso per il valore attuale del parametro.
Per esempio, se creo `Vcr<Point> a;` e utilizzo il metodo `a.l2norm()` occore che siano definite per il tipo `Point` tutte le operazioni che `Vcr<T>::l2norm()` esegue sul tipo generico `T` (presumibilmente `sqrt` e `operator *`).
Questo, in generale, non può essere verificato dal compilatore che al momento della *istanza*. È quindi opportuno che nella documentazione di una funzione/classe template siano indicati i prerequisiti richiesti alle classi utilizzabili come parametri attuali.

Specializzazione

Un'aspetto fondamentale delle classi template è la possibilità di specializzarle. Per esempio, la classe `Vcr<T>` rappresenta un *vettore generico* che implementa dei metodi che sono effettivamente utilizzabili per una ampia gamma di tipi.
Tuttavia, per alcuni tipi potrebbe essere necessario, o semplicemente computazionalmente conveniente, utilizzare dei metodi specializzati.
Per esempio, supponiamo di voler utilizzare `Vcr<complex>`, che memorizza degli elementi `complex` della Standard Library *(usare l'header `<complex>`)*. Il metodo generico `maxnorm()` e la funzione `dot` con ogni probabilità non funzionano per tale tipo (o non producono i risultati voluti).

Specializzazione parziale

```
template<class T> class Vcr< complex<T> > {
  int lenth;
  complex<T>* vr;
public:
  Vcr(int, complex<T>*);
  Vcr(const Vcr&);
  ~Vcr(){ delete[] vr; }
  int size() { return lenth; }
  complex<T>& operator[](int i) const { return vr[i]; }
        ...
        ...
  T maxnorm () const;
  T twonorm() const;

  template<class S>
  friend complex<S> dot(const Vcr<complex<S> >&,
                        const Vcr<complex<S> >&);
};
```

Specializzazione completa

```
template<>
class Vcr< complex<double> > {
  int lenth;
  complex<double>* vr;
public:
  Vcr(int, complex<double>*);
  Vcr(const Vcr&);
  ~Vcr(){ delete[] vr; }
  int size() { return lenth; }
  complex<double>& operator[](int i) const {return vr[i];}
     ...
     ...
  double maxnorm() const;
  double twonorm() const;

  friend complex<double> dot(const Vcr&, const Vcr&);
};
```

Metodi di classi templates specializzate

```
template<class T> T Vcr< complex<T> >::maxnorm() const {
T nm = abs(vr[0]);
for (int i = 1; i < lenth; i++) nm = max(nm,abs(vr[i]));
return nm; }
```

Occorre includere gli headers <complex> e <cmath> (per la funzione max). Si è assunto l'uso della direttiva using per portare i nomi della STL nello *scope*.

```
double Vcr< complex<double> >::maxnorm() const {
double nm = abs(vr[0]);
for (int i = 1; i < lenth; i++) nm = max(nm,abs(vr[i]));
return nm; }
```

Specializzazione di dot

```
template<class T>
complex<T> dot(const Vcr< complex<T> >& v1,
const Vcr< complex<T> >& v2){
if (v1.lenth != v2.lenth ) error(bad vector sizes);
complex<T> tm = v1[0] * conj(v2[0]);
for (int i = 1; i < v1.lenth; i++) tm += v1[i]*conj(v2[i]);
return tm; }
```

Overloading di dot

```
complex<double> dot(const Vcr< complex<double> >& v1,
const Vcr< complex<double> >& v2){
if (v1.lenth != v2.lenth ) error(bad vector sizes);
complex<double> tm = v1[0] * conj(v2[0]);
for (int i = 1; i < v1.lenth; i++) tm += v1[i]*conj(v2[i]);
return tm; }
```

In questo caso questa specializzazione è inutile, è messa solo a scopo illustrativo.

Altri esempi

```
template <typename T1, typename T2>
class MyClass{
...}
```

Specializzazione per puntatori (parziale)

```
template <typename T1, typename T2>
class MyClass<T1*, T2*>{
...}
```

Specializzazione per secondo tipo `int`

```
template <typename T>
class MyClass<T,int>{
...}
```

Specializzazione completa

```
template <>
class MyClass<char,char>{
...}
```

Specializzazione di funzioni template

Si possono specializzare le funzioni template. Attenzione il meccanismo è diverso da quello dell'overloading (anche se talvolta i risultati simili)

```
template <typename T> void swap (T & a, T & b);
template <typename T> void swap (vector<T> & a, vector<T>
& b);
template <> void swap (int & a, int & b);
```

Il simbolo `template<>` indica che è una specializzazione.

Alcune regole

- La definizione della classe o funzione template di base deve essere visibile quando si definisce una specializzazione.
- Quando si specializza una classe template bisogna fornire la definizione di *tutti* i metodi della classe specializzata
- La specializzazione di una funzione template deve avere la stessa struttura della funzione base, cioè corrispondere a una possibile istanza della funzione generale (anche se rimangono dei tipi incogniti).
- Al momento della istanza che corrisponde a una specializzazione sia la definizione del classe/funzione template specializzata che (almeno) la dichiarazione di quella generale devono essere visibili.

Una nota

Nella specializzazione di una funzione template si può omettere di caratterizzare completamente il nome della funzione solo quando il compilatore può dedurre il tipo dagli argomenti:

```
template <typename T>
bool myfun(){T var,...}
```

La scrittura

```
template <> bool myfun(){int var,...}
```

è errata: il tipo non è deducibile dagli argomenti. Per specializzare occorre scrivere

```
template <> bool myfun<int>(){int var,...}
```

Ordine di specializzazione

Quando utilizzo una classe o una funzione template di cui ho una 'famiglia' di specializzazioni quale sceglie il compilatore? La regola è piuttosto complessa, si da qui gli elementi fondamentali per le funzioni template.
Consideriamo le dichiarazioni seguenti:

```
template<typename T> abs(T);
template<typename T> abs(complex<T>);
double abs(double);
```

e il seguente pezzo di codice

```
a=abs(2.0);// (1)
b=abs(2);// (2)
complex<double> z(2,3);
double c=abs(z);// (3)
```

1. Usa `double abs(double)` perchè è una funzione non template la cui firma corrisponde esattamente.
2. Usa `abs<int>(int)` Perchè è una istanza di una funzione template la cui firma corrisponde esattamente.
3. Usa `abs<double>(complex(double))` perchè esiste una specializzazione che si adatta meglio.

Le regole

Nell'ordine il compilatore sceglie

- La funzione non template la cui firma corrisponde esattamente.
- La funzione template specializzata la cui firma corrisponde esattamente (senza conversoni). Se c'è più di una possibilità sceglie la più specializzata.
- La funzione che si adatta meglio tramite conversioni implicite. Attenzione, argomenti di funzioni template il cui tipo è determinato dal paramtero del template NON vengono convertiti implicitamente.
- Se ci sono più di una funzione che soddisfa i requisiti allo stesso livello emette un errore.

Regola d'oro: Aiutate il compilatore a capire cose volete fare. Non fidatevi della sua preveggenza: evitate conversioni implicite.

Templates per specificare policies

Presentiamo un esempio in cui la tecnica dei templates viene usata per specificare una *policy*, cioè una modalità di operare sui dati. Supponiamo di volere confrontare due stringhe e di voler avere la possibilità discegliere se fare in confronto case sensitive o case insensitive

N.B Nell'x esempio seguente bisogna usare gli headers `<strings>` e `<cctype>` (per la funzione `char toupper(char)`).

```
template <class S=Ncomp>
int compare(const string& v1, const string & v2) {
  for (int i = 0; i < v1.size() && i < v2.size(); i++ )
  if ( !S::eq(v1[i], v2[i]) ) return S::st(v1[i], v2[i]);
  return v2.size() - v1.size();
}
class Ncomp { // normal compare
public:
 static bool eq(char a, char b) { return a == b; }//equal
 static bool st(char a, char b) { return a < b; } //smaller
};
class Nocase {// compare by ignoring case
public:
  static bool eq(char a, char b) {//case-insensitive
    return toupper(a) == toupper(b);
  }
  static bool st(char a, char b) { //case-insensitive
    return std::toupper(a) < std::toupper(b);
  }
```

```
int main(){
string v1;
string v2;
...
cout<< compare(v1,v2); // case sensitive
cout<< compare<Nocase>(v1,v2); // case sensitive
```

organizzazione dei files

stack.hpp:

```
#ifndef STACK_HPP
#define STACK_HPP

#include <vector>

template <typename T>
class Stack {
  private:
    std::vector<T> elems;
  public:
    Stack();
    void push(T const&);
    void pop();
    T top() const;
};

#endif
```

stackdef.hpp:

```
#ifndef STACKDEF_HPP
#define STACKDEF_HPP

#include <stack.hpp>

template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);
}
...

#endif
```