Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 3

Luca Formaggia

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

A.A. 2006/2007

Introduzione al C++(3)

- Namespaces
- La direttiva using
- Class e struct
- Dichiarazione e definizione
- Il disegno di una classe
- Costruttori e distruttori
- assert
- inline
- variabili statiche

Namespace

Abbiamo visto il concetto di scope. Intrinsicamente legato a tale concetto è quello di namespace (spazio di nomi), che introduce uno *scope* dando la possibilità di attribuirne un nome.

Rappresentano il modo più elementare di implementare la tecnica del *name hiding* e di raggruppare funzionalità comuni in *moduli*.

Esempio

Supponiamo di avere fatto una serie di funzioni per il calcolo di diverse norme di un vettore di R^n, memorizzato in un `vector<double>` e di volerle raggruppare sotto il namespace `Norme`:

```
namespace Norme{
typedef double Real; typedef vector<Real> Vettore;
Real oneNorm(Vettore const & v);
Real twoNorm(Vettore const & v);
Real infNorm(Vettore const & v);
...
}
```

Nello stesso file (o in un altro file, come vedremo in seguito) avremo le definizioni:

```
namespace Norme{
Real oneNorm(Vettore const & v){
...
}
...
```

Nel programma che utilizza tali funzioni (la cui dichiarazione deve ovviamente essere visibile) si avrà

```
Norme::Real pippo;
Norme::Vettore vec;
...
pippo=Norme::twoNorm(vec);
```

la direttiva *using*

```
//Dichiarazioni del namespace Norme
...
int main(){
using Norme::Real;
using Norme::Vettore;
using Norme::twoNorm;
using namespace Norme;
Real pippo;
Vettore vec;
...
pippo=twoNorm(vec);
```

La direttiva using *NM::nome* richiama il *nome* del namespace *NM* nello scope corrente. La direttiva using namespace *NM* richiama tutti nomi del namespace *NM* nello scope corrente.

namespace innestati

I namespace si possono innestare uno entro l'altro:

```
namespace AlgebraLineare {
namespace Norme {
...
}}
...
AlgebraLineare::Norme::Real pippo;
using namespace AlgebraLineare;
typedef Norme::Vettore V;
V vec;
..
```

namespace innestati

I namespace si possono innestare uno entro l'altro:

```
namespace AlgebraLineare {
namespace Norme {
typedef double Real;
typedef vector<Real> Vettore;
crossprod(const Vettore & a, const Vettore & b, Vettore
& res); ...
}}
.... int main(){
AlgebraLineare::Norme::Real pippo;


...
AlgebraLineare::Norme::crossprod(a,b,c);
}
```

namespace alias

Si possono creare alias. Utile per semplificarsi il lavoro quando il nome è lungo

```
...
namespace ALN=AlgebraLineare::Norme;
ALN::Real pippo;
typedef ALN::Vettore V;
V vec;
..
```

namespace senza nome

Se in un file (o più propriamente una unità di compilazione –prossima lezione) è presente un `namespace` senza nome i nomi di variabili e funzioni li generati sono nomi globali visibile esclusivamente in tale unità. Si ricorda che una unita' di compilazione è formata da un file cpp (più gli eventuali file includi con la direttiva `include`)

```
namespace{
int a;// (non sarebbe necessario)
void f(int &, double const *);
}
...
a=5;// OK sono nello stesso file
...

c=f(a,f);// OK sono nello stesso file
```

La funzione `f` è locale alla unità di compilazione e invisibile alle altre.

Class e struct

La direttiva class permette di aggiungere al linguaggio tipi complessi (detti classi), definiti dall'utente.
La *classe* permette di incapsulare in una unica entità logica dati (variabili) e *funzioni* che operano sui dati.
Permette inoltre di dare diversi livelli di accesso ai dati e funzioni in essa contenuti.
Dati e funzioni di una classe sono detti membri della classe. In particolare le funzioni sono dette metodi di una classe.
L'istanziazione di una classe viene detto oggetto.
Le direttive `struct` (ereditata dal C) e `class` sono sinonimi: cambiano solo le modalità di default di accesso ai membri.

```
class Inutile{
public:
void set(const double & value){i=value;};
double get() const{return i;};
private:
double i; };// ricordarsi il ;


...
Inutile pippo;
pippo.set(3.5);//attribuisco il valore
cout«pippo.get();// stampo il valore
double a =pippo.i// NO! i è membro privato!
```

- get() è un metodo costante perchè non altera i membri della classe.
- L'accesso ai membri pubblici di un oggetto avviene attraverso l'operatore .detto di *accesso ai membri* (member access).

differenza tra class e struct

`class` Per default l'accesso ai membri è `private`

```
class MyClass{
double a,b,c;
double solve();
...}
```

`a, b, c` e `solve()` sono privati! Non posso accedervi dall'esterno.

`struct` Per default l'accesso ai membri è `public`

```
struct MyStruct{
double a,b,c;
double solve();
...
}
```

`a, b, c` e `solve()` sono pubblici!

Interfaccia e implementazione

Se volete usare `struct` fatelo solo per collezioni di dati con accesso pubblico (come era nel C).
Per qualunque struttura più complessa usate `class`.

L'insieme dei *membri pubblici* di una classe viene spesso chiamata interfaccia pubblica(o semplicemente interfaccia) della classe.
I metodi *privati* di una classe possono essere richiamati solo da altri metodi della classe. Fanno parte della implementazione della classe.

Dichiarazione di una classe

La dichiarazione di una classe contiene le dichiarazioni delle variabili e dei metodi.

```
#include <vector>
...
class Myvector{
public:
MyVector(const int ndim);//costruttore
double at(const int i) const;
....
private:
std::vector<double> v;};
```

Essa permette al compilatore di eseguire i controlli sintattici sui tipi (cioè verificare che essi sono usati coerentemente) e di stabilire la dimensione dell'*oggetto*. Una classe non può essere usata se prima non è dichiarata (o meglio se la dichiarazione non è visibile)

Definizione dei membri di una classe

La definizione fornisce la funzionalità dei metodi e i valori delle eventuali costanti statiche (che vedremo più avanti).

```
MyVector::MyVector(const int ndim): v(ndim){};
double MyVector::at(const int i) const
{return v[i-1]; }
```

Talvolta la definizione è integrata nella dichiarazione:

```
class Myvector{
public:
MyVector(const int ndim):v(ndim){};//costruttore
double at(const int i) const;
{return v[i-1];}
....
private:
std::vector<double> v;};
```

ma più sovente la dichiarazione è contenuta in un header file `hpp` mentre la definizione è in un file cpp (che `include` il file cpp corrispondente)

Il disegno di una semplice classe

Voglio creare una semplice classe per matrici bidimensionali piene di tipo *double* con le seguenti caratteristiche:

- Dimensionamento dinamico.
- Possibilità di accesso ai singoli elementi con il metodo `at(int i, int j)` con numerazione a partire da 1.
- Possibilità inserimento ai singoli elementi con il metodo `set(int i, int j, double v)` con numerazione a partire da 1.
- Controllo dei limiti dell'array quando si accede agli elementi.
- Possibilità di conoscere facilmente le dimensioni della matrice.

Un primo layout

```
class MyMat0{
private:
int nr,nc;
double * dati;
public:
void build(int const nrow; int const ncol);
inline int nrow()const;
inline int ncol()const;
inline double at(const int i, const int j) const;
void set(const int i, const int j, double const & val);
}
```

Come si usa?

```
MyMat0 m;// Cosa succede al momento della istanza?
m.build(10,10);Alloca spazio per la matrice
m.set(1,1,10.0);
...
```

- Cosa succede al momento della istanzazione di *m* ?
- Cosa succede quando *m* esce dallo suo scope ? In particolare: Chi libera la memoria nella heap che `build` ha allocato per fare spazio alla matrice?
- mi piacerebbe poter scrivere `MyMat0 m(10,10)` per dimensionare una matrice 10x10, o `MyMat0 m2(m)` (equivalentemente `MyMat0 m2=m`), cioè inizializzare una MyMat0 con un'altra istanza di MyMat0

costruttori e distruttori automatici

Se non viene esplicitamente fornito un costruttore (di cui parleremo più avanti) il linguaggio costruisce un oggetto istanziando i membri della classe, nell'ordine in cui sono elencati, con il cosidetto *costruttore di default automatico*. Il costruttore di default non prende argomenti.

Se non viene esplicitamente fornito un distruttore, all'uscita dell'oggetto dal suo *scope* viene chiamato il distruttore di default per tutti i membri, nell'ordine *inverso* rispetto all'ordine in cui sono elencati. Il distruttore non prende mai argomenti.

Il processo di costruzione

Quindi `MyMat0 m` crea `m` chiamando `int()` per `nr` e `nc` e `double *()` per `dati`. (il che è quello che ci aspettiamo).

Però sarebbe meglio assicurarsi che `nr` e `nc` siano inizializzati a 0 e *sopratutto* che il puntatore sia inizializzato al puntatore nullo (è buona pratica, per le ragioni che vedremo, inizializzare i puntatori al puntatore nullo).

Infine con il costruttore di default non si può inizializzare con valori `MyMat0 m(10,10)`. Per questi motivi forniremo alla nostra classe dei *costruttori*.

Il costruttore

Il costruttoreè un metodo speciale della classe che ha come nome il nome della classe stessae può avere argomenti, ma NON ha un tipo di ritorno (nemmeno void!!).
Come tutti i metodi può essere definito nel corpo della dichiarazione della classe oppure esternamente, mettendo nella classe solo la *dichiarazione*.
Nel caso i metodi vengano definiti al di fuori della dichiarazione della classe, deve essere usato le *scope resolution operator* `::`, nella forma `NomeClasse::NomeMetodo`.

Dichiarazione con costruttore

```
class MyMat0{
private:
int nr,nc;
double * dati;
public:
MyMat0(); //costruttore di default (non automatico)
MyMat0(int n, int m); //costruttore con argomenti
void build(int const nrow; int const ncol);
inline int nrow()const;
inline int ncol()const;
inline double at(const int i, const int j) const;
double set(const int i, const int j, double const & val);
}
```

Costruttore di default

```
MyMat0::MyMat0(){}
```

Questo costruttore fa esattamente quello che farebbe il costruttore fornito dal linguaggio. Se lo metto nel corpo della dichiarazione della classe si omette il `MyMat0::`.

```
MyMat0::MyMat0(): nr(0),nc(0),dati(0) {}
```

Costruttore con lista di inizializzazione esplicita per i membri. Qui inizializzo gli interi a 0 e il puntatore al puntatore nullo.
Avrei anche potuto fare cosi:

```
MyMat0::MyMat0(){nr=0;nc=0;dati=0;}
```

ma la forma precedente è da preferirsi perchè nella seconda forma i membri vengono prima costruiti (automaticamente) e poi *assegnati*.

Costruttore con argomenti

```
MyMat0::MyMat0(int n, int m):
nr(n), nc(m), dati(0){
build(n,m);}
```

Inizializzo e poi nel corpo del costruttore chiamo il metodo `build` che avevo già previsto e alloca la memoria (lo vediamo dopo).

Avrei potuto allocare la memoria direttamente nel costruttore. In questo caso è più conveniente farlo nella lista di inizializzazione.

```
MyMat0::MyMat0(int n, int m): nr(n), nc(m),
dati(new double[n*m]) {}
```

Nota: in presenza di membri const o reference si deve fornire esplicitamente un costruttore alla classe.

Distruttore di default

Il distruttore è un metodo speciale il cui nome è `~NomeClasse` e non ha tipo di ritorno e *non prende argomenti*.
Il distruttore fornito dal linguaggio chiama il distruttore dei membri.
Nel nostro caso significa che vengono "distrutti" `nc` , `nr` e `dati`.

Ma non si preoccupa di restituire al sistema operativo la memoria eventualmente allocata dinamicamente (e 'puntata' da `dati`), semplicemente perchè non può sapere che deve farlo!.

```
class MyMat0{
private:
int nr,nc;
double * dati;
public:
MyMat0(); //costruttore di default
MyMat0(int n, int m); //costruttore con argomenti
~MyMat0();
void build(int const nrow; int const ncol);
inline int nrow()const;
inline int ncol()const;
inline double at(const int i, const int j) const;
double set(const int i, const int j, double const & val);
}
```

Un distruttore per MyMat0

```
MyMat0::~MyMat0(){if (dati !=0) delete [] dati; }
```

Se avessi scritto

```
MyMat0::~MyMat0(){delete [] dati; }
```

sarebbe stato giusto lo stesso, grazie al fatto che ho inizializzato `dati` al puntatore nullo . Infatti `delete` applicato al puntatore nullo non esegue alcuna operazione!.

Costruttore di copia

Cosa succede se io faccio `MyMat0 a(10,10);`
`MyMat0 b=a; //oppure MyMat0 b(a);`
Si attiva il cosidetto costruttore di copia.

Il costruttore di copia usa l'istanza di una classe per inizializzare un nuovo oggetto della stessa classe.
Lo fa usando inizializzando i membri della nuova classe con gli analoghi membri della classe che si vuole copiare.

Ma nel nostro caso ci va bene?

No! E per 2 motivi. Il primo "formale": avendo fornito espicitamente dei costruttori (anche solo uno) il C++ prevede che i costruttori "automatici" forniti dal linguaggio non siano più disponibili (*shading*).

Il secondo è SOSTANZIALE. Con il costruttore di copia fornito del linguaggio (costruttore di copia automatico) `MyMat0 m(a);` copierebbe il puntatore `a.dati` in `m.dati`, mentre io voglio copiare il contenuto dell'area di memoria associata e non il puntatore!.

Chiariamo meglio

Con il costruttore di copia fornito automaticamente si avrebbe la situazione seguente

```
MyMat0 a(10,10);
a.set(1,1,1.5);// etc. etc.
MyMat0 b=a;
b.set(1,1,3.5);//anche a.at(1,1) è ora 3.5!!!
```

```
class MyMat0{
private:
int nr,nc;
double * dati;
public:
MyMat0(); //costruttore di default
MyMat0(int n, int m); //costruttore con argomenti
MyMat0(MyMat0 const & m);
~MyMat0();
void build(int const nrow; int const ncol);
inline int nrow()const;
inline int ncol()const;
inline double at(const int i, const int j) const;
double set(const int i, const int j, double const & val);
}
```

Il costruttore di copia di MyMat0

```
MyMat0::MyMat0(MyMat0 const & m):
nr(m.nr), nc(m.nc) {
if(nc*nr==0)dati=0;
else dati=new double[nr*nc];
double * const & mp=m.dati;//per semplificarmi la vita
for (int i=0;i< nr*nc;++i)dati[i]=mp[i];
}
```

Lo shading dei costruttori

La classe definisce uno 'scope'e quindi anche per i metodi di una classe si applicano le regole generali. Il costruttore è un particolare metodo di una classe il cui nome coincide con il nome della classe (la sua firma invece dipende anche dagli argomenti).
Applicando le regole generali allora non sorprende che:

Se si fornisce esplicitamente un costruttore (di qualunque genere) i costruttori automatici (di default e di copia) vengono oscurati. Se servono devono essere forniti esplicitamente

Il resto per completare la classe

```
double MyMat0::at(int const i, int const j) const
{assert(i>0 && i<=nr); assert(j>0 && i<=nc);
return *(dati + (j-1) + (i-1)*nc);}

void MyMat0::set(int const i, int const j, double const
& v){
assert(i>0 && i<=nr);assert(j>0 && i<=nc);
*(dati + (j-1) + (i-1)*nc)=v;}

void MyMat0::build(int const n, int const m)
{if (dati !=0) delete[] dati; dati=new double[n*m];
nr=n;nc=m}
```

assert

Usando l'header `<cassert>` o `<assert.h>` si ha a disposizione il comando `assert`:
`assert(espressione logica)`
Se l'espressione è `false` il programma abortisce con un messaggio di errore. È un metodo un pò brutale di controllo degli errori.
Se si compila con la opzione `-DNDEBUG` gli assert vengono disattivati.

la direttiva inline

Se un metodo viene dichiarato *inline* nella dichiarazione di una classe il compilatore cercherà di risolverlo "in linea", cioè senza una chiamata a una funzione vera e propria.
Questo genera codice più efficiente . Però è efficace solo per funzioni piccole.
Inoltre rende più difficile il debug .

Variabili statiche

Una variabile o un metodo può essere definito `static`.
Una variabile statica è unica per tutte le istanze della classe. Serve quindi per rappresentare dati comuni della classe e non dell'oggetto.
Le funzioni statiche sono le solo che possono modificare le variabili statiche.

Esempio di membri statici

```
class TriaElement{public: TriaElement()
...
static const int numnodes=3; }
...
//dimensiono un vettore
vector<int> nodeID(TriaElement::numnodes);
```

Rispetto a una variabile statica la classe si comporta sostanzialmente come un `namespace`.
`TriaElement a; int n=a.numnodes` è un errore perchè l'operatore `.` permette di accedere a membri di oggetti e non alle variabili statiche della classe.

Riassunto

La firma del costruttore di copia:

```
T::T(const T & a)
```

La firma dell'operatore di assegnamento di copia (copy assignement):

```
T& T:: operator =(const T & a)
```

Se si fornisce un costruttore di default privato la classe è non costruibile.
Se si fornisce un costruttore di copia privato la classe è non copiabile.
Se si fornisce un operatore di assegnazione (di copia) privato la classe è non assegnabile.