Programmazione Avanzata per il Calcolo Scientifico
Lezione N. 5

Luca Formaggia

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

A.A. 2006/2007

Le classi (parte 2)

- variabile e funzioni statiche
- Dereferenziamento di membri
- Funzioni friend
- La variabile this

Operatori

- Overloading di operatori
- Operatore di assegnazione
- conversione implicita

COnsideriamo la definizione seguente, contenuta in un header file (.hpp)

```
class WS{
public:
WS();
...
static int howmany();
static void reset();
private:
static int counter; // dichiarazione
...
};
```

Nel file sorgente (.cpp) corrispondente

```
WS::WS(){... ++counter;};
WS::~WS(){... -counter;};
int WS::counter=0;// definizione
...
int WS::howmany(){return WS::counter;}
void WS::reset(){WS::counter=0;}
```

Nell'esempio si mostra una classe che tiene conto del numero di oggetti della classe istanziati attraverso una variabile statica.

- Variabili e metodi statici sono membri della classe e non della istanze della classe. Vengono indirizzate con l'operatore di risoluzione di scopo (p. es `WS::howmany`).
- Variabili statiche membri privati della classe possono essere accedute solo tramite *metodi statici*.
- Le variabili statiche vengono dichiarate nel corpo della classe, ma definite all'esterno (tipicamente nel file sorgente). La definizione è l'unico caso in cui si può accedere direttamente un membro statico privato.

Definizione di costanti statiche

Anche i membri (variabili) costanti statici vengono definiti nel corpo della classe ma *inizializzati* al di fuori di esso.
Possono fare eccezione variabili statiche intere ed enumerazioni, che possono essere inizializzate direttamente nel corpo della classe

```
class pippo{
public:
static const int np=5;//OK è intero
static const float pi=3.14;//NO!
static const float pi;
...};
const int pippo::np;serve comunque!
const float pippo::pi=3.14; //OK
```

La definizione fuori del corpo della classe serve per poter accedere alla variabile statica come oggetto (p.es `pippo p; int i=p.np`)

L'operatore ->

```
class sphere{...
point2D radius;...
};
...
sphere * S;...
point2D R=S->radius;
```

Per referenziare il membro `radius` dell'oggetto di tipo `sphere` «puntato»da `S` ho usato l'operatore di *deferenziamento di membri* (member dereferencing operator):

$$\texttt{S->radius} \leftrightarrow \texttt{(*S).radius}$$

Funzioni amiche di una classe

Una funzione dichiarata `friend` nella definizione di una classe può accederne direttamente i *membri privati*.

```
class element{
public:
friend point2D computeIntersection(element & a, element
&b);
private:
double x;...
...};
point2D computeIntersection(element & a, element &b){
double z=a.x*b.y;//possibile perche friend!
...
```

Quando fare una funzione friend

Spesso la funzionalità di una funzione friend può essere ottenuta aggiungendo un metodo alla classe.
In generale è meglio usare un metodo della classe, ma talvolta l'uso di una funzione friend è da preferirsi (o inevitabile!). Vedremo caso per caso in seguito.

La variabile this

```
class pippo
public:
float calcola(float a, float b)
...
;
```

Un metodo (non statico) di una classe contiene un argomento formale nascosto. Nell'esempio il metodo `calcola` è come se avesse come firma

```
float calcola(pippo * ,float, float );
```

e, quando viene chiamato, al primo argomento viene associato come argomento attuale l'espressione this che è un puntatore all'oggetto corrente.

Questo è in effetti il meccanismo con cui il metodo accede ai membri dell'oggetto!

La variabile this

this è una variabile il cui nome è *riservato*, quindi nessun'altra variabile o funzione può chiamarsi così.

Il meccanismo di passaggio di membri a un metodo (non statico!) è il seguente:

```
double Point2D::xCoord() const{
return _x;//_x è un membro di Point2D }..
```

```
double Point2D::xCoord(Point2D const * this) const{
return this->_x;}
```

L'argomento in blue è in realtà nascosto. Tuttavia, nel corpo della funzione io avrei potuto scrivere `this->_x` al posto di `_x`.

Cos'è un operatore

Un'operatore è una funzione specialeche prende la forma seguente:

`Type operator` $\triangle$ `(Argomenti)`

`Argomenti` sono opzionali e $\triangle$ è il simbolo dell'operatore che può prendere uno dei valori seguenti:

```
  +    -    *    /    %    ^    &    |    ~
  !    =    <    >    +=   -=   *=   /=   %=
  ^=   &=   |=   <<   >>   >>=  <<=  ==   !=
  <=   >=   &&   ||   ++   --   ,    ->*  ->
  ()   []
```

Si può inoltre fare l'overloading di `new`, `delete`, `new[]` e `delete[]`. Esistono poi gli *operatori di conversione*. Un operatore può (in generale) essere implementato sia come una *funzione libera* che come *metodo* di una classe. Se è un metodo il primo argomento è implicitamente l'oggetto della classe.

Degli operatori elencati è possibile fornire una implementazione da parte dell' utente. Spesso si parla in questo caso di *operator overloading* -sovraccaricamento in Italiano-.

Il numero e tipo di argomenti dipendono dal tipo di operatore. In generale si distinguono operatori unari, operatori binari e operatori generici. Gli operatori unari si distinguono inoltre in prefix e postfix. Omettiamo i dettagli che risulteranno chiari negli esempi.

Gli operatori =, `[]` e `->` possono essere esclusivamente membri non statici di una classe.

Non è permesso fare il sovraccaricamento di `::` (scope resolution), `.` (member selection) e `.*` (member selection through pointer to function).

Regole generali

- Usate un operatore al posto di una funzione ordinaria SOLO quando il simbolo corrispondente ha un significato intuitivo nel contesto.
- Definite gli operatori per tipi complessi in modo che si comportino analogamente ai corrispondenti operatori su classi semplici: when in doubt, do as the `int`'s do.
- È altamente sconsigliato sovraccaricare operatori logici (`&&` e `||`).
- Il C++ da grande libertà su argomenti e tipi di ritorno di un operatore: usate però le regole comuni.

operator =

```
class Vtr {
int length;
double* ets;
public:
...
Vtr& operator=(const Vtr&);
```

L'operatore = è sempre implementato come metodo. Dati due oggetti `v` e `z` di tipo `Vtr` si ha

`v=z;` $\leftrightarrow$ `v.operator=(z);`

Il compilatore fornisce di default un *operatore di assegnazione automatico*, che fa la copia membro a membro (ma qui NON è quello che vogliamo).
Infatti

1. `v=z` deve modificare `v` in modo che contenga gli stessi valori di `z`.
2. Vogliamo inoltre che $v = z = y$ abbia il significato $z = y; v = z$, in altre parole che si traduca in $v = (z = y)$ (l'operatore di assegnazione ha associatività da destra a sinistra).
3. Vogliamo evitare preoblemi se si esegue `v=v`.

L'implementazione

Questa è una implementazione leggermente diversa da quella del libro (pag.213). Permetto infatti che l'operatore sia applicabile a due `Vtr` di lunghezza diversa.

```
Vtr& Vtr::operator=(const Vtr& z){
if (this !=&z){if(length!=z.length){delete[] ets;ets=new
double(z.length);}
length=z.length;
for(int i=0;i!=length;++i)ets[i]=z[i] }
return *this; }
```

L'operatore = è un esempio di operatore binario:

$$\texttt{a} \triangle \texttt{b} \leftrightarrow \texttt{a.operator} \triangle \texttt{(b)}$$

operator +=

Trattiamo prima l'operatore += dell'operatore + per ragioni che vedremo in seguito.

Voglio poter fare `a+=b` essendo `a` e `b` dei `Vtr`.
L'operatore += viene di norma implementato come metodo ed è un operatore binario. Dato che per le variabili primitive in cui è definito esso ritorna una referenza all'oggetto, vogliamo seguire la stessa convenzione anche per la nostra classe, la firma sarà dunque

```
Vtr& operator+=(const Vtr &);
```

implementazione

```
Vtr & Vtr::operator+=(const Vtr& v) {
if (length != v.length ) error(bad vector sizes);
for (int i = 0; i < length; i++) ets[i] += v[i];
return *this;}
```

Il comando `error` è simile ad `abort()`.

Altre implementazioni possibili

È ovviamente possibile prevedere altre definizioni dell'operatore, con argomento di tipo diverso, p.es

```
Vtr& operator+=(const double &);
```

con implementazione, per esempio,

```
Vtr & Vtr::operator+=(const double & a) {
for (int i = 0; i < length; i++) ets[i] += a;
return *this;}
```

Esso somma il valore a a tutti gli elementi di `Vtr`.
Ma è intuitivo?. Evitate implementazioni non intuitive degli operatori. In questo caso probabilmente un metodo di nome `addScalar` è meno ambiguo.

operator []

Voglio che `v[i]` ritorni un elemento di `v`. Tale operatore può essere esclusivamente un metodo. Nel corpo della classe avrò

```
inline double& operator[](int i);
```

e l'implementazione

```
double& Vtr::operator[](int i) {return ets[i];};
```

Ritorna una referenza per poter essere un lvalue: `v[i]=5;`. È opportuno metterlo *inline* dato che si prevede che verrà utilizzato spesso.

operator ()

L'operatore parentesi tonda è molto più flessibile di `[]` in quento *accetta argomenti in numero e tipo arbitrario*.
Posso anche averne più d'uno. Per esempio potrei stabilire che `v()` ritorna un puntatore a double sugli elementi del `Vcr v` e `v(i)` l'i-esimo elemento con convenzione Matlab/Fortran

```
double const * operator()() const;
double& operator()(int i);
```

```
double const * operator()() const{return ets;};
double& operator()(int i){return ets[i-1];};
```

operatore binario +

L'operatore + è meglio implementarlo come funzione esterna alla classe. Questa è una regola abbastanza generale per tutti gli operatori binari simmetrici.

Deve ritornare un oggetto temporaneo, in quanto non modifica nessuno dei suoi argomenti. Inoltre tale oggetto è opportuno sia const per evitare che l'istruzione a+b=c sia ammissibile!

```
class Vtr{...
friend Vtr const operator+(const Vtr&, const Vtr &);
...
}

Vtr const operator+(const Vtr& v1, const Vtr & v2) {
if (v1.length != v2.length ) error(bad vtor sizes);
Vtr sum(v1);
sum += v2;
return sum;
}
```

Per tutte le coppie di operatori $\triangle$ e $\triangle=$ implementare sempre $\triangle$ utilizzando $\triangle=$.

una considerazione

Non vi è modo di implementare correttamente l'operatore + (o -, * etc.) senza produrre un temporaneo. Questo per oggetti di grande taglia introduce un problema di efficienza.

Si può ovviare in più modi: 1) usare funzioni ordinarie, p. es.

```
void addVtr(const Vtr & a, const Vtr & b, Vtr & res);
```

o 2) ricorrere agli *expression templates* (tecnica di metaprogrammazione avanzata che illustreremo (se ci sarà tempo) più avanti.

L'operatore unario -

Se si vuole poter scrivere `z=-v+s`, dobbiamo fornire la regola che produce un oggetto temporaneo che contiene `-v`. Nel nostro caso si vuole che tale oggetto abbia gli elementi di `v` con il segno opposto.
Qui sfrutto il fatto che ho già definito il costruttore `Vtr(int l)` che produce un vettore nullo di lunghezza `l` e che ho già definito l'operatore `-`.
A differenza del libro qui scelgo di implementarlo come metodo della classe.

```
class Vtr{...
Vtr const operator-() const;
...
}

const Vtr Vtr::operator-() {
return Vtr(lentgh) - v; }
```

operator ++

Anche se non si applica a `Vcr` si vuole presentare il caso dell'operatore unario ++, che, analogamente a -, presenta sia una una versione `prefix` che una versione `postfix`.

Supponiamo allora di avere definito una classe `MyInt` di interi con numero arbitrario di cifre (possibile applicazione la crittografia) e di aver già definito `MyInt::operator+=(int const &)` e di volere ora definire anche gli operatori di incremento (e decremento). Sappiamo che per un intero `++i` e `i++` si comportano in modo molto diverso.

`++i` e `i++`

- `++i`: *update and fetch*. Aggiungi 1 a `i` e ritorna una referenza a `i` .
- `i++`: *fetch and update*. Ritorna il valore corrente di `i` e quindi aggiunge 1 a `i`.

Si vuole cha anche per `MyInt` si segua la stessa semantica. Si vuole inoltre impedire che `i++++` sia lecito (ambiguo!), mentre `++++i` sia permesso!.
Gli operatori di incremento e decremento si implementano di solito come metodi della classe (ma non è obbligatorio). La versione `postfix` prende un argomento intero fittizio, che serve solo al compilatore per generare una *signature* diversa.

Prefix:

```
MyInt & MyInt::operator++(){
*this +=1; //update
return *this; }//fetch
```

Postfix:

```
const MyInt MyInt::operator++(int){
MyInt tmp(*this);fetch
++(*this); //update
return tmp; }
```

Si noti come l' operatore postfix sia implementato in funzione dell'operatore prefix e come il secondo ritorni una variabile const.

operator «

Gli operatori di *stream* sono particolari. È sempre meglio implementarli come funzione esterna, *friend* della classe. Prendono in input una referenza a uno stream (`istream` o `ostream`) e ritornano la referenza allo stesso stream. In questo modo è possibile la *concatenazione* (`cout«a«b«endl`).

```
#include<iostream>
... class Vct{
public:
...
friend std::ostream& operator«(std::ostream&, const Vtr&);
friend std::istream& operator»(std::istream&, Vtr&);
```

Implementazione

```
std::ostream& operator«(std::ostream& s, const Vtr& v )
{
for (int i =0; i < v.length; ++i ) {
s « v[i] «  ;
if (i%10 == 9) s « std::endl;
}return s;
}

std::istream& operator»(std::istream& s, Vtr& v ) {
for (int i =0; i < v.length; ++i )s » v[i];
return s;
}
```

La conversione implicita

Il compilatore C++ permette di definire delle regole di conversione tra tipi usando sue meccanismi: la conversione per costruzione e gli operatori di conversione.
La conversione per costruzione si attiva ogniqualvolta esista un *costruttore* utilizzabile con un solo argomento.

conversione per costruzione

```
class Rational {
public:
Rational(int num= 0,int den= 1);
...}
Rational simplify(rational&);
```

L'istruzione

```
r=simplify(1);
```

si traduce automaticamente in

```
r=simplify(Rational(1));
```

operatori di conversione

Un operatore di conversione è un *metodo* di una classe della forma

```
operator Type() const;
```

e converte un oggetto della classe nel tipo voluto.

```
class Rational {
public:
operator double() const;

...}
```

Permette di fare

```
Rational r(1,10);
double d=r+10.9;
```

attenzione alle conversioni implicite

Le conversioni implicite sembrano molto comode, ma spesso possono dare risultati non voluti:

```
Rational r(1, 2);
cout « r;
```

Se non ho definito `operator«` per il tipo `Rational` il compilatore lo converte in un `double`... ma è quello che volevo?

costruttori espliciti

Per evitare che un costruttore possa essere usato per una conversione implicita si usa la parola chiave explicit. Il suo uso è *altamente consigliato* a meno che non si desideri effettivamente attivare la conversione implicita.

```
class Rational {
public:
explicit Rational(int num= 0,int den= 1);
...}
```

Metodo o funzione?

Ho una funzione che opera su oggetti di una classe. La implemento come funzione libera o membro della classe?

- se la funzione è `operator»` o `operator«` o se si vuole permettere una conversione di tipo nel suo primo argomento da sinistra, o se può essere implementato usando solo l'interfaccia pubblica della classe allorausate una funzione libera(nom membro della classe), eventualmente `friend` se necessario.
- se deve operare in modo polimorfico implementatela come funzione membro virtuale
- altrimenti implementatela come membro.