Programmazione ad oggetti

Appunti a cura di *Andrea Abriani (@aabriani), Davide Gagliardi (@davidegagliardi), Andrea Braghiroli*

Fattori qualità software:

Esterni

- Estendibilità: sistema amplia funzionalità che il software aveva in precedenza. Programmazione ad oggetti viene resa più semplice.
- Riusabilità: software e componenti possono essere riutilizzati per un insieme di requisiti che possono essere diversi da quelli precedenti.

Interni:

fattori percepibili al codice della sorgente

- Strutturazione: codice viene scritto in modo tale da astrarre delle singole parti. E' possibile un'individuazione di parti di software indipendenti dalle altre (es. Input/output può essere posizionato in zona differente).
- Modularità: collegata alla strutturazione. Va oltre la struttura e indica che può essere usata in modo diverso e con componenti può raggiungere scopi e requisiti differenti. (Il codice può essere utilizzato per scopi diversi).
- Comprensibilità: software strutturato in modo che corrisponda alla descrizione del dominio.

Caratteristiche della programmazione a oggetti

Key idea: connessione esplicita funzioni-dati (non vi sono più struct con metodi e costruttori)

Connessione esplicita fra funzioni e dati - Incapsulamento

In precedenza, vi era una procedura che lavorava su variabili globali. Ora invece, le funzioni sono connesse ai dati. Questo delimita ciò che le funzioni possono fare. Questo avviene tramite la procedura di incapsulamento. Questa procedura non avviene per la sicurezza dei dati bensì per la visibilità. Solo alcuni metodi e funzioni potranno accedere agli elementi di una determinata classe.

Concetto di classe

Le classi sono legate in gerachie. Una eredita l'altra per creare una struttura.

Si pone enfasi sulla rappresentazione. In caso di errori si andrà a realizzare una struttura sbagliata.
L'enfasi viene spostata da computazione a rappresentazione (più o meno astratta) sul quale il software deve agire

Enfasi sulla rappresentazione

Il software può essere visto come un insieme di scatole, nelle quali entrano ed escono dati.

Viene utilizzato l'UML (*unified modeling language*), uno standard che contiene un insieme di linguaggi grafici per realizzare un diagramma delle classi e degli oggetti, ottenendo una rappresentazione grafica della programmazione.

Oggetti in UML
All'interno dell'UML, la relazione oggetto-classe è simile a elemento-insieme

Classe	Insieme
Oggetto	Elemento

Gli oggetti risultano essere elementi del dominio con vita propria indipendente con identificatore e istanza.

La parte grafica in UML non è arbitraria ma unificata, ovvero vengono utilizzate convenzioni per rappresentare determinati elementi del programma.

Ereditarietà

L'ereditarietà consente di creare classificazioni gerarchiche. E' possibile creare una classe generale che definisce le caratteristiche comuni a una serie di oggetti correlati. La classe può, in seguito, essere ereditata da una o più classi, ognuna delle quali aggiunge alla classe solo elementi specifici.
Si definisce *classe base* la classe ereditata, mentre la classe che "riceve" l'eredità è detta *classe derivata*. Avviene mediante la seguente sintassi:

```
class NomeClasseDerivata : tipoEreditarietà NomeClasseBase{
  // contenuto della classe
  // il tipoEreditarietà può essere public, private, protected
};
```

Quando, ad esempio, il tipo di accesso alla classe base è `public`, tutti i membri `public` della classe base diverranno membri `public` della classe derivata, Lo stesso discorso varrà per i membri `protected`.
Per gli elementi di tipo `private` della classe base, questi non saranno accessibili da parte dei membri della classe derivata.

Una classe derivata può ricevere in eredità elementi di due o più classi base.

Di seguito alcuni esempi di ereditarietà:

```
class A : tipoEreditarietà B{
  public:
    void m4();
}

class B {
  private:
    void m1();
  public:
    void m2();
    void m3();
};

A seconda delle ereditarietà

  A a;
  a.m1();
  a.m2();
  a.m3();

  // Chiamo i metodi m1, m2, m3
```

Ereditarietà di tipo public `class A: public B{};`

- Quando è private in B diventa inaccessibile in A ed inaccessibile dall'esterno

```
main() {

  A a;
  a.m1(); // NO m1 è private in B (inaccessibile da classi derivate à private)
```

```
}

void A::m4(){
  m1(); // NO m1 è private in B e quindi inaccessibile da A
}
```

- Quando è public in B diventa public in A

```
main() {

  A a;
  a.m2(); //OK
}
void A::m4(){
   m2(); //OK accessibile anche per classi derivate
}
```

- Quando è protected in B diventa protected in A

```
main() {
  A a;
  a.m3(); // NO, non posso accedere dall'esterno
}

main() {
  B b;
  b.m3();  // NO
}
void B::m2(){
  m3(); // OK
}
void A::m4(){
  m3(); // OK, se in B qualcosa è protected -> resta inaccessibile dall'esterno,
        // ma accessibile sia da classe base che da classe derivata
}

IS-A -> ogni istanza di A è anche istanza di B -> EREDITARIETA'
```

Ereditarietà di tipo private `class A: private B{};`

- Quando è private in B diventa inaccessibile da A ed inaccessibile dall'esterno

```
main() {

  A a;
```

```
  a.m1(); //NO
}

void A::m4(){
  m1(); //NO *come nel caso precedente con ereditarietà public*
}
```

- Quando è public in B diventa private in A

```
main() {

A a;
a.m2(); //NO -> privatizzato per l'esterno uguali
}

void A::m4(){
  m2(); //OK -> accessibile anche per classi derivate
}
```

- Quando è protected in B diventa private in A

```
main() {
  A a;
  a.m3();  //NO
}

void A::m4(){
  m3(); //OK
}
```

Ereditarietà di tipo protected `class A: protected B{};`

- Quando è private in B diventa inaccessibile da A ed inaccessibile dall'esterno

```
main() {

  A a;
  a.m1(); //NO
}

void A::m4(){
  m1();  //NO
}
```

- Quando è public in B diventa public in A

```
main() {

  A a;
  a.m2(); //NO à esterno no
}

void A::m4(){
  m2(); //OK -> interno ok
}
```

- Quando è protected in B diventa protected in A

```
main() {
  A a;
  a.m3(); //NO
}

void A::m4(){
  m3(); //OK
}
```

		Base			
		Private		Public	
Derivata	Public	Inaccessibile dall'esterno	Inaccessibile dalla derivata	Inaccessibile dall'esterno	Access dalla derivat
	Private	Inaccessibile dall'esterno	Inaccessibile dalla derivata	Inaccessibile dall'esterno	Access dalla derivat
	Protected	Inaccessibile dall'esterno	Inaccessibile dalla derivata	Inaccessibile dall'esterno	Access dalla derivat

Le 2 righe `private` e `protected` sono uguali per accessibilità: ci sono conseguenze su classi derivate

	Private	Public	Protected
Public	Inaccessibile	Public	Protected
Private	Inaccessibile	Private	Private
Protected	Inaccessibile	Protected	Protected

Quando ha senso fare ereditarietà private/protected?
Ereditare in modo `private` implica la natura della classe base è nascosta dall'implementazione (ereditarietà per implementazione). In sostanza si effettua questo tipo di ereditarietà quando attributi e metodi sono utilizzati per l'implementazione.

La classe derivata diventa così diversa dalla classe sopra, ne deriva che non utilizzerà i metodi come la classe sopra (superclasse)

Quando ha senso fare ereditarietà public?

```
class A: public B {
    void m4();
};
A a;
B b;
a = b; // NO
b = a; // SI     (b è la classe base)
```

a contiene b, a è più grande

```
A& A::operator=(const A&a)
B& B::operator=(const B&b)  // OK per ereditarietà pubblica
```

Differenze valore / puntatore / riferimento

- Passaggio per valore: il valore di una variabile viene copiato in una struttura stack. Successivamente la funzione recupera il dato. Viene utilizzato questo tipo di passaggio quando la variabile è "piccola".
- Passaggio per indirizzo: viene copiato l'indirizzo della variabile in modo esplicito. Il passaggio per indirizzo viene utilizzato se la variabile è "grande".
- Passaggio per riferimento (esclusivo del c++): alla funzione viene passato l'indirizzo e non

il valore dell'argomento. Questo approccio richiede meno memoria rispetto alla chiamata per valore, e soprattutto consente di modificare il valore delle variabili che sono ad un livello di visibilità (scope) esterno alla funzione o al metodo.

In linea di massima, si tende a non fare la copia.

Operatore "++"

Può essere pre-fisso (`++i`) o post-fisso (`i++`).

Nel caso di incremento pre-fisso, la variabile verrà incrementata nello stesso ciclo:

```
int i = 5;
cout << "Il valore di i e' " <<  ++i;

//L'output del programma sarà "Il valore di i è 6"
```

Nel caso di operatore post fisso, la variabile verrà incrementata nel ciclo successivo:

```
int i = 5;
cout << "Il valore di i e' " <<  i++;

//L'output del programma sarà "Il valore di i è 5"
```

Se l'operatore viene applicato ad un puntatore, il valore della cella di memoria rimarrà il medesimo. Verrà però incrementato l'indirizzo di memoria contenuto dal puntatore.

Operatore "new"

L'operatore `new` alloca un puntatore di memoria, in questo modo l'allocazione avverrà in modo dinamico. Bisogna però preoccuparsi di utilizzare il comando `delete` , per non aumentare l'occupazione di memoria da parte del programma.

Costruttori

Quando viene costruita una classe vuota, viene fatta una definizione astratta. Proprio per questo non è scontato che il compilatore allochi memoria (come per il `typedef`).
L'allocazione di memoria avverrà una volta istanziata una classe. In questo modo verrano inseriti in memoria anche un puntatore `this` (che punta a dove la variabile è allocata) e un secondo puntatore verso i metodi di quella classe.

A livello di memoria, il compilatore punta i costruttori che ci sono sempre (default, 0 parametri, distruttore, costruttore di copia, operatore di assegnazione).

Header

Si tratta della dichiarazione all'inizio del file con cui si dichiara il nome che il compilatore usa per chiamare il metodo (che è l'unico nel programma).

Costruttore

Funzione membro che si occupa di:

- allocazione della memoria per tutti i suoi membri (dati e funzioni) nello heap o nello stack;
- inizializzazione dei dati membri

Caratteristiche del costruttore sono:

- nome uguale a quello della classe di appartenenza;
- non ha un valore di ritorno

Il costruttore può accettare uno o più argomenti. Nel caso di costruttore a zero parametri, prende il nome di costruttore di *default*. Nel caso fossero assenti i costruttori, il compilatore è in grado di generarne uno automaticamente (con i limiti del caso).

L'uso di un costruttore appositamente definito, invece, consente di inizializzare tutti i membri della classe assieme, di modo che l'istanziazione di un oggetto sia sempre consistente con il nostro modello di dati e indipendente dalle scelte del compilatore.

```
NomeClasse::NomeClasse(){
  cout << "Costruttore di default/zero parametri";
}
NomeClasse::NomeClasse(Nome _nome, Cognome _cognome){
  cout << "Costruttore specifico a due parametri";
  nome = _nome;
  cognome = _cognome;
}
```

Distruttore

Il distruttore di classe assolve il compito di rilasciare tutte le risorse associate ad un oggetto, quando esso non è più necessario.

```
NomeClasse::~NomeClasse(){
  cout << "Il contenuto e' stato distrutto";
}
```

Il distruttore è caratterizzato dal fatto che non ha valori di ritorno né parametri. Per questo motivo non può essere sovraccaricato.

Il suo compito primario dovrebbe essere sempre e solo quella di rimuovere un oggetto e tutte le sue dipendenze dallo stato del programma in maniera sicura e completa.
Viene invocato automaticamente per tutte le variabili, ogni volta che viene raggiunta la fine del loro ambito di visibilità, oppure nel caso di deallocazione tramite il comando `delete`.

Attributi statici operatori

Gli attributi statici sono attributi istanziati solo una volta che valgono per tutte le istanze della classe.
Se un'istanza cambia il contenuto, allora viene cambiata per tutte le istanze

Esempio:
`static int numIstanze`

- indica il n° di licenze disponibili nel programma.

Dal punto di vista funzionale, la variabile statica è come una variabile globale.

`const float miaConst`

- non cambia mai dopo l'inizializzazione (attributo costante -> `A():miaConst(1,1) {…}`)
- qui ho già inizializzato l'attributo

`static const int a = 1;`

- `static const int b;` private
- `const float A::b = 1.0` fuori dal main

`static const int a;`

- NON posso scriverlo; con `static` potrei mettere il valore dopo l'inizializzazione, solo che con `const` `a` deve esistere

Overload operatori come metodi

- [a = 3]

```
A& operator = (const int _k) {
    k = _k;
    cout << “operator = const int” << endl;
    return *this;
}
```

- [b = a]

```
A& operator = (const A& aa) {
    k = aa.k;
    cout << “operator = const A&” << endl;          NOTA: qualcuno ha già implement
    return *this;                                uguale tra int e float
}
```

- [a = a + b]

```
A& operator + (const A& aa) {
    A temp;                                 //creata classe temporanea
    temp.k = k + aa.k;                      //k = valore classe in cui sono dentro
    cout << “operator + const A&” << endl; //aa.k = valore classe passata
    return temp;                            //temp = il mio risultato
}

// A crea una nuova istanza. Prendo le 2 istanze e le sommo.
// Infine metto il valore nella nuova istanza
// temporanea -> la somma non modifica l’istanza corrente
```

- [c += a]

```
A& operator += (const A& aa) {
    k += aa.k;
    cout << “operator += const A&” << endl;
    return *this;
}
```

- [a == c]

```
A& operator == (const A& aa) {
    cout << “operator == const A&” << endl;     //NOTA: NO per classi diverse!!
    return k == aa.k;
}
```

`return *this == bb;` è un confronto infinito (ricorsivo) tra classi, quindi devo confrontare i valori

- [a != c]

```
A& operator != (const A& aa) {
    cout << “operator += const A&” << endl;
    return !(*this == aa);
}
```

- [++a;] pre-incremento

```
A& operator ++ () {
    cout << “operator ++A” << endl;
    k++;
  return *this;
}
```

- [++a;] post-incremento

```
A operator ++ (int) {
    A aa = *this;
    k++;
  cout << “operator A++” << endl;
  return aa;    //istanza-1 perché viene distrutto l’operatore di copia
}
```

il passaggio del parametro di tipo `int` è ciò che distingue il post-incremento dal pre-incremento.

Nel caso si voglia implementare [a == b]

```
friend bool operator == (const A& aa, const B& bb);     // dentro i 2 public delle
bool operator == (const A& aa, const B& bb) {           // funzione esterna alla cl
    return aa.val = bb.val;
}
```

Al contrario, per implementare [b == a] bisogna cambiare `A` con `B` ed `aa` con `bb` e viceversa.
[a == a] è un metodo interno.

Operatori in C++ e la loro ridefinizione (operator overloading)

- Operator `<<`
- Operator `=`
- Operator `++`
- Operator `==`

Gli operatori sono funzioni, quindi è possibile fare l'overloading degli operatori. Ciò vuol dire che alla stessa entità si possono dare significati diversi (capita per le funzioni, posso fare overloading).

Cosa sono gli operatori?

Sono operazioni che si trovano nelle espressioni dei vari tipi base (per esempio nell' `int` : gli operatori predefiniti).
L'operatore sempre definito in ogni classe è l'operatore `=`.
Qualsiasi classe noi definiamo, il comando `new` restituisce un puntatore.

Data l'espressione `a = b + c;`, il compilatore c++ cerca gli operatori/metodi per svolgere l'espressione

- nei tipi base: ci sono predefiniti
- nelle classi: li vado a cercare (l'operatore `=` c'è sempre, mentre il `+` lo va a cercare)

Nota bene: L'overloading degli operatori è un caso particolare dell'overloading di funzioni.

A livello codice, per alcuni operatori l'overloading è realizzabile in due modi:

- come metodo
- come funzione esterna

Al contrario, l'operator `=` ad esempio, è possibile implementarlo solo come metodo.

Consideriamo il seguente frame di codice

```
class A {
private:
  int i;
public:
    A(int _i) { i = _i; }
    A operator + (const A& _a) const {return A( i + _a.i )};
    friend A operator + (const A&, const A&);
    A operator + (const A& _a1, const A& _a2) {
    return A(_a1.i + _a2.i + i); //NON FUNZIONA, i è privato. Uso funzione esterna
  };
```

vediamo gli operatori che vengono chiamati nei prossimi esempi

```
A a1(1), A a2(2);
a1 = a2 + 3;
a1 operator = a2.operator + (A(3));     // chiamato costruttore ad un parametro
a1 = 3 + a2;                            // non c'è explicit quindi va
a1 operator = (operator + (A(3), a2));  // l'operatore è quello di intero,
                                        // non trova modo di fare intero con a
                                        // e quindi mi da errore
```

```
a1 = a1 + a2;
A a1(1);  A a2(2);
a1 operator = (a1 operator + (a2));
A a3;
a3 = a1 + a2;
```

```
int c;  c = 3 + 2;
A d;  d = 3 + 2;
d operator = (A(3 + 2));
```

```
a1 += a2;
class A {
  private:
    int i;
  public:
    A(int _i) {i = _i;}
    A& operator += (const A& _a){   // & è referenza
      i += _a.i;  return *this;     // NO const dopo, perché l'oggetto chiamato sar
    }
}
```

nel `return` c'è operazione di deferenziazione, quindi ritorna la stessa istanza, ma modificata (`lvalue` per fare assegnazione e quindi come fa l'operazione di assegnazione).

```
class A {
private:
  int i;
public:
  A(int _i) {i = _i;}
  bool operator < (const A& _a) const { return (i < _a.i); }

  A operator * (const A& _a) const {
        A temp = (*this);
        return temp *= _a;          //return (*this) * _a; NO!!!
  }

  A& operator *= (const A& _a) {    //efficienza uguale ma scrivo meno
        (*this) = (*this) * _a;
        return (*this);
  }
```

Standard Template Library (STL)

Una Standard Template Library è universalmente nota; è portabile (integrata con il linguaggio, quindi il compilatore deve compilare ed avere accanto quella libreria standard).

Di seguito elenchiamo una funzione che calcola il minimo tra due variabili, riguardo diversi tipi:

```
int min(_i int, _j int) {
    if (_i < _j) return _i;
    else return _j;
}
```

```
double min( _t double,  _s double);
```

```
class A {
private:
  int i;
public:
  bool operator < (const _a A) const;
};
```

```
bool A::operator <(const _a A) const {
  return i < _a.i;
}

A min (_a1 A, _a2 A);
A min (_a1 A, _a2 A) {
    if (_a1 < _a2)  return _a1;
    else return _a2;
}
B min (_b1 B, _b2 B) {
  if (_b1 < _b2)  return b1;
  else return b2;
}
```

Un template è utilizzato per definire opzioni, metodi e strutture parametrizzate.

Ad esempio, per fare un cerca e copia di funzioni posso generalizzare rispetto ad un parametro.

```
template <E> E min (E e1, E e2)   // header definizione template
template <E> E min (E e1, E e2){  // implementazione fuori main (nel main sarebbe s
  if (e1 < e2)  return e1;
    else return e2;
  }
```

È possibile definire una classe template avendo una classe personalizzata?

```
template <E> class list <E> {
  public:
    int size();
};
```

Allora con `#include <list>`

```
list <int> l;
l.push_front(3);
l.push_back(1);
```

E' necessario standardizzare la varie classi che ricevono.

Nella STL ci sono:

- container (contenitori)
- iterator (iteratori)
- algorithms

I container sono classi che contengono qualcosa:

- list
- vector
- set
- queue
- stack
- map

Uso i container per non implementare continuamente le stesse cose

```
class A {
private:
  list <B> lb;
public:
}
```

Gli algoritmi sono funzioni alle quali possono essere passati dei contenitori (iterator ai contenitori).

Fanno cose come minimo dalla lista

- efficiente
- testato, usato molte volte
- si passa ad una programmazione dove si compongono pezzi con algoritmi che ci sono

Definisco delle classi (organizzate in una gerarchia) che mi permettono l'accesso agli elementi del contenitore.
Gli iteratori sono un modo per accedere ai contenitori: non sono puntatori, ma ci assomigliano per alcune funzionalità.

```
list/set/vector <int>::iterator it;        //è un iteratore bidirezionale
map <int, float>::iterator iter;
```

Un esempio di implementazione è

```
list <int> l;    //se l è vuoto, l.begin() == l.end();   iteratore è oggetto della
l.push_front(3); l.push_back(2);
list <int>::iterator it;
for(it = l.begin(); it != l.end(); it++) {
  cout << *it;    // overloading di operatori per dereferenziare l'operatore -> * r
                  // puntato, è un operatore unario
}
```

Virtual

Consideriamo il seguente frame di codice:

```
class Personaggio {
private:
  int vita = 100;
public:
  virtual void stampa() = 0;  // =0 rende il metodo puramente virtuale
  virtual bool operator < (const Personaggio&);
};
//Personaggio p;    NON posso scriverlo, la classe ha un metodo puramente virtuale
```

Tramite `virtual`, posso decidere in che classe della gerarchia va fatto eseguire un determinato metodo. In poche parole, con il `virtual`, il compilatore non esegue il metodo della classe madre, ma quello delle classi derivate.
Con gli operatori, prima guardo di che tipo (classe) è il primo operatore (sempre che in quella classe ci sia implementato lo stesso metodo del `virtual`) e poi il secondo che deve essere lo stesso tipo del primo.

```
class Cavaliere: public Personaggio {
private:
  string nome;
  int forza;
public:
  Cavaliere (string n, int f);
  void stampa() {cout << … ;}
};

class Mago: public Personaggio {
private:
  string nome;
  int potere;
public:
```

```
  bool operator < (const Mago&);
  friend ostream& operator << (ostream& os, const Mago& m);
}

main() {
  list <Personaggio*> l;
  l.push_front(new Mago("Circe", 100));
  l.push_back(new Cavaliere("Odolfo", 100));
}
```

Allora, considerando la seguente implementazione

```
Personaggio* pp;
pp = new Cavaliere("Astolfo", 100);
pp->stampa();
```

- Senza `virtual`, e con implementazione di `stampa()` in `Personaggio`, avrei stampato quello in `Personaggio`
- Con `virtual`, vado a scegliere a runtime il metodo più giusto per stampare (`stampa()` in `Cavaliere`)

A conferma di ciò, si verifica il seguente caso

```
Personaggio* pp = new Mago( … );
pp -> stampa();
  //Mago::stampa(); perché ho virtual (con = 0 o senza)
  //Personaggio::stampa() non ho virtual
```

Altro caso `virtual`

```
class Personaggio {
private:
  int vita = 100;
public:
  virtual void Stampa() = 0;
  friend ostream& operator << (ostream& os, const Personaggio& p);
  friend virtual ostream& op(ostream& os); //non è virtuale puro perché implementat
}
ostream& Personaggio::op (ostream& os) {
  return os << vita;
}

ostream& operator << (ostream& os, const Personaggio& p){
```

```
}
```

Non si possono definire i metodi esterni come `virtual`, neanche i costruttori!