Programmazione Orientata agli Oggetti

prof. Riccardo Torlone
Università di Roma Tre

Paradigmi di programmazione

- Imperativo
  - un programma è composto di istruzioni che specificano operazioni (comandi) da eseguire;
- Funzionale
  - un programma è la specifica di una funzione (che a sua volta può contenere la specifica di altre funzioni)
- Logico
  - un programma è la descrizione delle proprietà dei risultati del programma, basata sulla logica matematica;
- Orientato agli oggetti
  - un programma è la specifica di un insieme di classi di oggetti, ognuna definita per mezzo di struttura e operazioni

Nota: non si tratta di una classificazione netta; ad esempio C++ è un linguaggio imperativo orientato agli oggetti.

Programmazione orientata agli oggetti

- **classificazione**
  - dati (oggetti) organizzati in classi con proprietà comuni
- **incapsulamento**
  - oggetto = dato + operazioni
- **information hiding**
  - ogni oggetto ha un'interfaccia (pubblica) e una implementazione (privata)
- **polimorfismo**
  - molteplici definizioni delle stesse funzioni (**overloading**)
  - Funzioni e classi parametriche rispetto a tipi di dato
- **ereditarietà**
  - definizione di nuove classi sulla base di classi definite in precedenza, aggiungendo o modificando caratteristiche
  - introduce altre proprietà interessanti (**late binding**)

Classificazione

- Processo di astrazione che consiste nella organizzazione dei dati di un'applicazione in classi di oggetti che hanno proprietà comuni ed esistenza autonoma (Dipartimento, Impiegato, Acquisto, Vendita,…)

- Una semplice **classe** in C++:

```
class NumeroComplesso {
public
    float Re; // parte reale
    float Im; // coeff dell'immaginario
    float Modulo() { return sqrt(Re * Re + Im * Im); }
}
```

- **Oggetto**: elemento di una classe (dato)

```
NumeroComplesso c;
```

Classificazione e Modularizzazione

La classificazione permette di realizzare un principio di base della programmazione: la modularizzazione.

- Modularizzazione:
  - Tecnica di programmazione volta a costruire programmi articolati in componenti indipendenti con iterazione ben definita
- Modulo:
  - Componente di programma che:
    - offre (esporta) servizi ad altri moduli, attraverso una opportuna interfaccia (“fornitore”);
    - utilizza (importa) servizi offerti da altri moduli (“clienti”);
    - ha una propria struttura interna, non visibile all’esterno (cfr. information hiding).

Benefici della modularizzazione

- Significa decomposizione — tecnica essenziale per affrontare problemi complessi
- E' più facile sviluppare applicazioni complesse da parte di gruppi di progetto (le realizzazioni dei moduli sono indipendenti l'una dall'altra)
- Aumenta la comprensibilità e la leggibilità dei programmi, con benefici su:
  - verifica di correttezza (è più facile isolare eventuali errori);
  - riusabilità
- *Il C++ offre altri strumenti per la modularizzazione (che approfondiremo più avanti):*
  - *funzioni (come nei linguaggi imperativi)*
  - *separazione di un'applicazione in più file*

Incapsulamento

- Gli oggetti sono descritti e organizzati insieme alle relative operazioni

```
class NumeroComplesso {
public
    float Re; // parte reale
    float Im; // coeff dell'immaginario
    float Modulo() { return sqrt(Re * Re + Im * Im); }
}
.....
NumeroComplesso c;
c.Re = 4.;
c.Im = 3.;
cout << c.Modulo(); // stampa il valore 5
cout << Modulo();  // errore: Modulo() si
```

Oggetti e messaggi

- il paradigma orientato agli oggetti viene spesso definito attraverso il modello “oggetto-messaggio”
- nel paradigma imperativo tradizionale, le funzioni si chiamano l'un l'altra con parametri di ingresso e uscita
- nel paradigma O-O, ogni dato (oggetto appartenente ad una classe) sa eseguire “su se stesso” le operazioni ammissibili, secondo le richieste che gli provengono dall'esterno
- l’atto di invocazione di una operazione di una classe (metodo) attraverso un oggetto della classe viene tradizionalmente chiamato “invio di messaggio ad un oggetto”

Information hiding

- Ogni oggetto ha:
  - un'interfaccia
  - una implementazione
- Per utilizzare l'oggetto si fa riferimento all'interfaccia;
- L’implementazione non è accessibile dagli utenti della classe;
- E’ possibile modificare l’implementazione senza dover cambiare l’interfaccia;
- Consente il disaccoppiamento tra moduli (importante nello sviluppo di applicazioni complesse);
- In C++ si realizza con separazione di una classe (che descrive l’oggetto) in:
  - parte pubblica (contenente metodi);
  - parte privata (contenente proprietà dell’oggetto).

Implementazione NumeroComplesso

```
class NumeroComplesso {
public
    float Modulo() { return sqrt(Re * Re + Im * Im); }
    float Fase() {
        if (Im >= 0) return acos(Re/Modulo());
        else return -acos(Re/Modulo());
    }
    float Reale() { return Re; }
    float Immaginaria() { return Im; }
private
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

Interfaccia

Implementazione

Oggetti della classe NumeroComplesso

```
void main() {
    NumeroComplesso c;
    cout << c.Reale(); // corretto
    cout << c.Modulo(); // corretto
    cout << c.Re; // scorretto
    cout << c.Im; // scorretto
}
```

Un’altra implementazione della stessa classe

```
class NumeroComplesso {
public
    float Modulo()      { return m; }
    float Fase()        { return phi; }
    float Reale()       { return m * cos(phi); }
    float Immaginaria()  { return m * sin(phi); }
private
    float m; // modulo
    float phi; // fase
};
```

Oggetti della nuova classe NumeroComplesso

```
void main() {
        NumeroComplesso c;
        cout << c.Reale(); // corretto
        cout << c.Modulo(); // corretto
        cout << c.m; // scorretto
        cout << c.phi; // scorretto
}
```

Uso della classe NumeroComplesso

```
int Quadrante(NumeroComplesso c) {
    if (c.Reale()==0 || c.Immaginaria()==0)
        return 0; // nell'origine o su uno degli assi
    if (c.Reale() > 0 && c.Immaginaria() > 0)
        return 1;
    if (c.Reale() < 0 && c.Immaginaria() > 0)
        return 2;
    if (c.Reale() < 0 && c.Immaginaria() < 0)
        return 3;
    if (c.Reale() > 0 && c.Immaginaria() < 0)
        return 4;
}
```

Polimorfismo

- Polimorfismo = capacità di assumere forme diverse (Webster's Dictionary)
- Nei linguaggi O-O esistono diverse forme di polimorfismo:
  - Overloading (sovrapposizione, sovraccarico): più definizioni per uno stesso nome di funzione
    - oltre alle funzioni definite nel programma, possono essere ridefiniti (o estesi) gli operatori del linguaggio (+,—,/,…)
  - Funzioni parametriche rispetto ad uno o più tipi
  - Classi parametriche rispetto ad uno o più tipi
    - realizzate in C++ mediante i *template*

Esempio di overloading: costruttori

```
#include <iostream.h>
class NumeroComplesso {
public:
    void stampa() { cout << Re << '+' << Im << 'i' << endl; }
    NumeroComplesso() { Re = 0.; Im = 0.; }
    NumeroComplesso(float r, float i) { Re = r; Im = i; }
    NumeroComplesso Somma(NumeroComplesso c) {
        NumeroComplesso c1;
        c1.Re = c.Re + Re;  c1.Im = c.Im + Im;
        return c1;
    }
private :
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

Uso della classe NumeroComplesso

```
void main() {
    NumeroComplesso c1(-3.2,2.0);
    NumeroComplesso c2(4.2,1.5);
    NumeroComplesso c3, c4;
    c1.stampa();
    c2.stampa();
    c3 = c2.Somma(c1);
    c3.stampa();
}
```

Ridefinizione dell'operatore +

```
#include <iostream.h>
class NumeroComplesso {
public:
    void stampa() { cout << Re << '+' << Im << 'i' << endl; }
    NumeroComplesso(float r, float i) { Re = r; Im = i; }
    NumeroComplesso() { Re = 0.; Im = 0.; }
    NumeroComplesso operator+(NumeroComplesso c) {
        NumeroComplesso c1;
        c1.Re = c.Re + Re; c1.Im = c.Im + Im;
        return c1;
    }
private :
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};
```

Uso della classe NumeroComplesso

```
void main() {
        NumeroComplesso c1 (-3.2,2.0);
        NumeroComplesso c2 (4.2,1.5);
        NumeroComplesso c3 , c4;
        c1.stampa();
        c2.stampa();
        c3 = c1 + c2;
        c3.stampa();
        c4 = c1.operator+(c2); //equivalente
        c4.stampa();
}
```

Polimorfismo parametrico: i template

- Nei linguaggi tradizionali, strutture simili che differiscano solo al livello di "tipo componente" richiedono definizioni diverse, anche se molto simili
- Nei linguaggi OO è possibile definire in maniera parametrica rispetto a un tipo:
  - le funzioni
  - le classi
- Vediamo alcuni esempi:
  - funzioni calcolo max
  - collezione di interi
  - collezione di caratteri

Funzioni per il calcolo dell'elemento più grande

```
int maxInt (int a, int b) {
    if (a>b) return a;
    else return b;
}

char maxChar (char a, char b) {
    if (a>b) return a;
    else return b;
}
```

Funzione max parametrica

```
#include <iostream.h>

template <class T>
T max (T a, T b) {
    if (a > b) return a;
    else return b;
}

main()  {
    int n1 = 4, n2 = 5;
    char c1='b', c2='g';
    cout  << "il piu' grande tra " << n1 << " e " << n2
          << " e' " << max (n1,n2) << endl ;
    cout  << "il piu' grande tra " << c1 << " e " << c2
          << " e' " << max (c1,c2) << endl;
}
```

Collezione di interi

```
#include <iostream.h>
const int N=100;
class collezioneInteri {
    int v[N];
    int nelem;
public:
    collezioneInteri() { nelem=0; }
    collezioneInteri(int n) { int i; for(i=0;i<N;i++) v[i]=n; nelem=0; }
    void aggiungi (int n) { v[nelem]=n; nelem++; }
    int togli() { int n=v[nelem]; nelem--; return n; }
    int numeroElementi() { return  nelem; }
    void stampa() {  // stampa tutti gli elementi della collezione
        int i; for(i=0; i<nelem; i++) cout << v[i]; cout << endl;
    }
};
```

Uso della classe collezioneInteri

```
main(){
    collezioneInteri ci(0); // collezione di interi
    ci.aggiungi(1); ci.aggiungi(2); ci.aggiungi(3);
    ci.stampa(); // 1 2 3
    int n;
    n = ci.togli();
    ci.stampa(); // 1 2
}
```

Collezione di caratteri

```
#include <iostream.h>
const int N=100;
class collezioneCaratteri {
    char v[N];
    int nelem;
public:
    collezioneCaratteri() { nelem=0; }
    collezioneCaratteri(int c) { int i; for(i=0;i<N;i++) v[i]=c; nelem=0; }
    void aggiungi (int c) { v[nelem]=c; nelem++; }
    char togli() { char c=v[nelem]; nelem--; return c; }
    int numeroElementi() { return  nelem; }
    void stampa() { // stampa tutti gli elementi della collezione
        int i; for(i=0;i<nelem;i++) cout << v[i]; cout << endl;
    }
};
```

Uso della classe collezioneCaratteri

```
main(){
    collezioneCaratteri cc('X'); // collezione di caratteri
    cc.aggiungi('A'); cc.aggiungi('B'); cc.aggiungi('C');
    cc.stampa(); // A B C
    char c;
    c = cc.togli();
    cc.stampa(); // A B
}
```

Collezione parametrica

```
#include <iostream.h>
const int N=100;
template <class T> class collezione {
    T v[N];
    int nelem;
public:
    collezione() { nelem=0; }
    collezione(T x) {int i; for(i=0;i<N;i++) v[i]=x; nelem=0; }
    void aggiungi (T x) { v[nelem]=x; nelem++; }
    T togli() {T x=v[nelem]; nelem--; return x; }
    int numeroElementi() { return  nelem; }
    void stampa() { // stampa tutti gli elementi della collezione
        int i; for(i=0;i<nelem;i++) cout << v[i]; cout << endl;
    }
};
```

Uso della classe collezione

```
main(){
    collezione<char> cc('X'); // collezione di caratteri
    cc.aggiungi('A'); cc.aggiungi('B'); cc.aggiungi('C');
    cc.stampa();
    char c;
    c = cc.togli();
    cc.stampa();

    collezione<int> ci(0); // collezione di interi
    ci.aggiungi(1); ci.aggiungi(2); ci.aggiungi(3);
    ci.stampa();
    int n = ci.togli();
    ci.stampa();
}
```

Ereditarietà (o derivazione)

- Meccanismo che permette di definire (derivare) nuove classi a partire da classi già definite aggiungendo elementi (dati e funzioni)
  - classe base e classe derivata
- Concetti simili sono definiti in vari contesti, con termini diversi: derivazione, sottotipo (subtyping), ereditarietà (inheritance), generalizzazione, sottoinsieme...
- Alcune regole generali:
  - un oggetto della classe derivata eredita tutte le proprietà della classe base
  - il tipo della classe derivata è "compatibile" con il tipo della classe base: gli oggetti della classe derivata sono utilizzabili dove è previsto che compaiano oggetti della classe base;
  - il tipo della classe base non è compatibile con il tipo della classe derivata.

Classe Persona

```
class Persona {
    char Nome[20];
    int Eta;
public:
    Persona(char n[20], int e) { strcpy(Nome,n); Eta=e; }
    void outDati(char n[20], int& e) { strcpy(n,Nome); e=Eta; }
};

void stampaDati(Persona p) {
    char n[20];
    int e;
    p.outDati(n,e);
    cout << n << " " << e << endl;
}
```

Classe Studente derivata da Persona

```
class Studente : public Persona {
    int Matricola; // eredita da Persona le proprietà Nome ed Età,
                   // il costruttore e il metodo outDati
public:
    Studente(char n[20], int e, int m):Persona(n,e) { Matricola=m; }
};

main( ) {
    Persona p("Giorgio Berti",25);
    Studente s("Marina Guali",26,33333);
    stampaDati(p);
    stampaDati(s);
}
```

Proprietà della derivazione

La derivazione ha diverse proprietà interessanti (che verranno approfondite):

- Da una classe base è possibile derivare diverse classi
- Una classe si può ottenere per derivazione di più classi base (**ereditarietà multipla**)
- Si possono costruire complesse gerarchie di derivazione
- E' possibile che una classe sia derivata in diverse maniere dalla stessa classe base (**ereditarietà condivisa**)
- E' possibile rimandare l'associazione tra invocazione di funzione e sua implementazione a tempo di esecuzione (**late binding**)
- Si possono costruire classi base astratte
- Si può combinare derivazione e polimorfismo (template)