Linux@evening

seminari di informatica libera

solira

Introduzione a C++ e alla programmazione OO

Nunzio Brugaletta

Di cosa si parla: Paradigma OOP e C++98

Si determini quali classi si desiderano;
si fornisca un insieme completo delle operazioni di ogni classe;
si renda esplicito ciò che hanno in comune mediante l'ereditarietà.

Sviluppato in origine con il nome **C con classi** nel 1983 da Bjarne Stroustrup come miglioramento del linguaggio C.

- Un C migliore
- Supporto a: dati astratti, OOP, programmazione generica
- Linguaggio multi-paradigma

Ultima standardizzazione 1998 (C++98)
in attesa di nuovo standard C++0x (C++Ax?)

Problema decomposto in unità piccole(funzioni)

Strutture di dati

```
struct libro {
  string titolo;
  string autore;
  string editore;
  float prezzo;
  bool presente;
};

libro l;
```

Funzioni di manipolazione delle strutture

```
bool prestitoOk(){
  bool esito=false;
  if (l.presente){
    l.presente = false;
    esito = true;
  };
  return esito;
};
```

Esempio: libro di una biblioteca e operazione di prestito

Le classi in c++

```
class clibro {
  public:

    // funzione inline

    void setLibro(libro l)
         {lib = l;};
    bool prestitoOk();
  private:
    libro lib;
};

bool clibro::prestitoOk(){
  bool esito=false;
  if (lib.presente){
    lib.presente = false;
    esito = true;
  };
  return esito;
};

// oggetto della classe

clibro lib1;
```

- **Incapsulamento**: una classe contiene I dati e le procedure di manipolazione (attributi e metodi)
- **Data Hiding**: i dati sono dichiarati nella parte privata della classe

Livelli di visibilità:

- **Private**: accessibilità solo dai membri della classe
- **Public**: accessibilità dall'esterno
- **Protected**: accessibilità dai membri della classe e dalle classi discendenti

Applicazione paradigma: definizione classe (1)

problema:

conoscere il totale della fattura di cui siano date le righe che la compongono, ognuna individuata da una determinata quantità e dal prezzo unitario dell'oggetto venduto

Dati:

riga fattura con quantità venduta e prezzo unitario

Operazioni sui dati:

generazione di una nuova riga.
Calcolo del totale della riga

```
class riga{
  public:
    void nuova(int, float);
    float totriga();
  private:
    int qv;
    float pu;
};

// metodo per inserimento di una nuova riga

void riga::nuova(int q, float p){
  qv = q;
  pu = p;
};

// metodo per il calcolo del totale della riga

float riga::totriga(){
  float tr;
  tr = (float) qv*pu;
  return tr;
};
```

Applicazione paradigma: definizione classe (2)

problema:

conoscere il totale della fattura di cui siano date le righe che la compongono, ognuna individuata da una determinata quantità e dal prezzo unitario dell'oggetto venduto

Dati:

fattura con totale

Operazioni sui dati:

inizializzazione totale (**costruttore**)
aggiornamento totale
output totale

```
class fattura{
  public:
    fattura(){totale=0.0;};
    void aggiorna(float t){totale += t;};
    float getTot(){return totale;};
  private:
    float totale;
};
```

Costruttore:

1. Ha lo stesso nome della classe e non ha tipo ritornato
2. Non può essere richiamato esplicitamente
3. Viene richiamato quando si instanzia un oggetto della classe
4. Può essere definito anche un **distruttore** (es. **~fattura**)

Applicazione paradigma: uso classe

```
// Calcolo del totale di una fattura date le righe di vendita

#include <iostream>
#include "c_riga"
#include "c_fattura"
using namespace std;

main(){
  int qvend,i;
  float pzunitt;
  riga r;
  fattura f;

  // elaborazione righe fattura

  for(i=1;;i++){
    cout << "Riga n. " << i << endl;
    cout << "Quantita\' oggetti venduti (0 per finire) ";
    cin  >> qvend;

    if(!qvend)
      break;

    cout << "Prezzo unitario ";
    cin  >> pzunit;

    r.nuova(qvend,pzunit);
    f.aggiorna(r.totriga());
  };

  // totale cercato

  cout << "\nTotale fattura " << f.getTot() << endl;
}
```

Livello di visibilità per gli oggetti delle librerie standard

r oggetto della classe riga
f oggetto della classe fattura

Invio di **messaggi** all'oggetto:

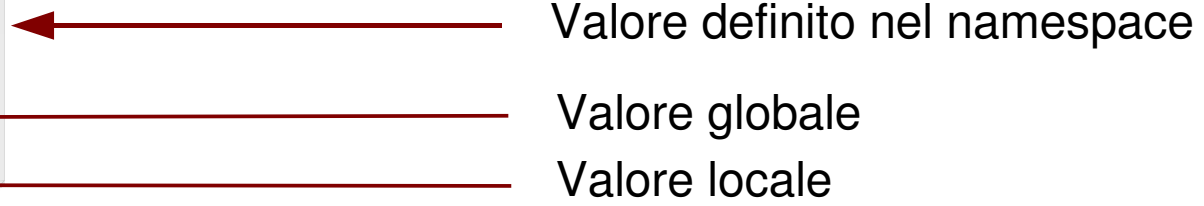
le variabili diventano un oggetto

si richiede dall'oggetto di esplicitare una sua competenza

Scope e namespace

Si possono definire diversi livelli di visibilità delle variabili

```
#include <iostream>
using namespace std;

namespace prova1{
  int a = 5;
};
namespace prova2{
  int a = 10;
};

int a = 15;

main(){
  int a = 20;
  cout << prova1::a << endl;
  cout << prova2::a << endl;

  cout << ::a << endl;

  cout << a << endl;
}
```



Classi Libreria c++: string

```
#include <iostream>
#include <string>
using namespace std;

main(){
  string cognnome, nome, cognome, nomecogn;
  int pos;

  cout << "Da una stringa con cognome e nome separati da spazio" << endl
       << "estrae cognome nome e inverte ordine\n\n";

  cout << "Cognome e nome separati da uno spazio ";
  getline(cin,cognnome);

  // ricerca spazio ed estrazione

  pos = cognnome.find(' ');
  if(pos>=0){
    cognome = cognnome.substr(0,pos);
    nome = cognnome.substr(pos+1);
  }

  nomecogn = nome+" --- "+cognome;
  cout << "Cognome: " << cognome << endl;
  cout << "Nome: " << nome << endl;
  cout << "Inverso: " << nomecogn << endl;
}
```

Oggetti della classe string

Due metodi della classe string

Classi Libreria C++: vector

Alcuni metodi della classe **vector**: `push_back`, `size`, `at`, `insert`, `erase` ...

```
#include <iostream>
#include <vector>
using namespace std;

main(){
  vector<int> numeri;
  int i,temp;
  int cosa,pos;

  cout << "Inserisce numero in vettore ordinato ed" << endl;
  cout << "Elimina elemento, conoscendone posizione \n\n";

  // Riempimento di un vettore contenete 10 elementi

  cout << "Inserimento elementi ordinati" << endl;
  for(i=0; i<10; i++){
    cout << "elemento " << i << " -->";
    cin  >> temp;

    numeri.push_back(temp);
  };

  cout << "Elemento da inserire ";
  cin >> cosa;

  // Ricerca posizione

  pos=-1;
  for(i=0; pos<0 && i<numeri.size(); i++){
    if(cosa<numeri.at(i))
      pos = i;
  }

  // Inserimento nella posizione trovata

  if(pos>=0)
    numeri.insert(numeri.begin()+pos,cosa);
  else
    numeri.push_back(cosa);

  // Vettore con nuovo elemento inserito

  cout << "\nNuovo vettore " << endl;
  for(i=0; i<numeri.size(); i++)
    cout << numeri.at(i) << "\t";
  cout << endl;

  // Eliminazione di un elemento dal vettore

  cout << "Da quale posizione togliere? ";
  cin >> pos;
  numeri.erase(numeri.begin()+pos);

  // Nuova visualizzazione

  cout << "\nNuovo vettore " << endl;
  for(i=0; i<numeri.size(); i++)
    cout << numeri.at(i) << "\t";
  cout << endl;
}
```

Libro biblioteca

Interfaccia classe

```
struct datilib{
  string titolo;
  string autore;
  string editore;
  float prezzo;
};

class libro {
  public:
    void setLibro(datilib);
    void getLibro(datilib&);
    bool getDisponibile(){return presente;};
    bool prestitoOk();
    bool restituitoOk();
  private:
    datilib libbib;
    bool presente;
};
```

Implementazione metodi classe

```
void libro::setLibro(datilib lset){
  libbib = lset;
  presente=true;
}

void libro::getLibro(datilib& lget){
  lget = libbib;
}

// Operazioni di prestito

bool libro::prestitoOk(){
  bool esito=false;
  if (getDisponibile()){
    presente = false;
    esito    = true;
  };
  return esito;
}

bool libro::restituitoOk(){
  bool esito=false;
  if (!getDisponibile()){
    presente = true;
    esito    = true;
  };
  return esito;
}
```

Classi aggregato

Oltre al libro c'è pure la libreria
con i propri attributi e metodi

Dati:

tabella con libri

Operazioni sui dati:

aggiungere un nuovo libro
estrazione dati da un libro
aggiornamento dati del libro (prestito)
conoscere quantità libri presenti

```
class libreria{
  public:
    int aggiungi(libro);
    bool estrai(int, libro&);
    bool aggiorna(int, libro);
    int dotazione(){return bib.size();};
  private:
    vector<libro> bib;
};

// Aggiunge un libro

int libreria::aggiungi(libro lib){
  bib.push_back(lib);
  return dotazione()-1;
};

// Estrae le informazioni su un libro

bool libreria::estrai(int quale,libro& lib){
  bool estrattoOK=false;

  if(quale>=0 && quale<dotazione()){
    lib = bib.at(quale);
    estrattoOK=true;
  }
  return estrattoOK;
};

// Aggiorna i dati di un libro esistente

bool libreria::aggiorna(int quale,libro lib){
  bool eseguito=false;

  if(quale>=0 && quale<dotazione()){
    bib.at(quale)=lib;
    eseguito = true;
  }
  return eseguito;
};
```

Interazioni fra oggetti

Esempio:

Registrazione di un prestito effettuato su un libro

Si prende un libro dalla libreria
Si registra nel libro l'operazione
Si aggiornano i dati nella libreria

```
// Prestito di un libro

void prestito(libreria& l){
  int quale,posultimo;
  libro lib;

  // Scelta libro per operazione

  posultimo = l.dotazione()-1;
  cout << "\nQuale libro (0," << posultimo << ")? ";
  cin >> quale;

  // Registrazione nuovo stato libro

  if (l.estrai(quale,lib)){
    if(lib.prestitoOk()){

      // Aggiornamento libro nella libreria

      l.aggiorna(quale,lib);
      cout << "\nPrestito registrato" << endl;
    }else
      cout << "\nLibro non presente" << endl;
  }else
    cout << "\nLibro inesistente" << endl;
};
```

ereditarieta'

solira

- Si può definire una nuova classe (classe discendente, ***subclass***) a partire da una classe esistente (classe antenata, ***superclass***).
- La classe C2 eredita dalla classe C1 se C2 **IS-A** C1 (C2 è un C1)
- La classe discendente ha tutte le proprietà della classe genitrice.
- La classe discendente ha le sue proprietà e metodi che specializzano gli oggetti della classe rispetto a quelli della classe antenata.
- Nella classe discendente si possono ridefinire i metodi ereditati dalla classe genitrice o adattarli alla propria specificità (***overloading*** dei metodi).
- Quando si richiama, per un oggetto, un metodo di cui è stato fatto l'overloading, ogni oggetto assume comportamenti in base alla propria specificità (***polimorfismo***).

Ereditarieta': esempio

Nella gestione della biblioteca è necessario, ora, registrare anche dati riguardanti il socio che prende in prestito il libro.

```
struct datisoc{
  string nome;
  string cognome;
  string via;
};

class libSocio : public libro {
  public:
    libSocio();
    void getSocio(datisoc& sget){sget = socbib;};
    bool prestitoOk(datisoc);
    bool restituitoOk();
  private:
    void setSocio(datisoc s1){socbib = s1;};
    datisoc socbib;
};
```

```
// Inizializzazione dati socio

libSocio::libSocio(){
  socbib.nome=" ", socbib.cognome=" ", socbib.via=" ";
}

// Prestito del libro al socio

bool libSocio::prestitoOk(datisoc s1){
  bool esito=false;

  if(libro::prestitoOk()){
    setSocio(s1);
    esito=true;
  }
  return esito;
}

// Restituzione libro dal socio

bool libSocio::restituitoOk(){
  bool esito=false;
  datisoc s1;

  if(libro::restituitoOk()){
    s1.nome=" ", s1.cognome=" ", s1.via=" ";
    setSocio(s1);
    esito=true;
  }
  return esito;
}
```

Esempio: commenti

```
class libSocio : public libro
```

La classe libSocio eredita i metodi pubblici della classe libro. Tutto ciò che era valido per oggetti di tipo libro vale anche per oggetti di tipo libSocio

```
bool libSocio::prestitoOk
```

Il metodo prestitoOk viene redifinito e specializzato per la classe ...

```
if(libro::prestitoOk()){
  setSocio(s1);
  ...
```

Il metodo è specializzato aggiungendo a quanto specificato prima, le nuove elaborazioni da svolgere (registrare i dati del socio che ha preso in prestito il libro)

Overloading degli operatori

```
class libro {
  public:
    void setLibro(datilib);
    void getLibro(datilib&);
    bool getDisponibile(){return presente;};
    bool prestitoOk();
    bool restituitoOk();
    friend ostream& operator<<(ostream& output, const libro&);
  private:
    datilib libbib;
    bool presente;
};

// Ridefinizione operatore << per oggetti della classe

ostream& operator<<(ostream& output, const libro& l){
  output << l.libbib.titolo  << ' '
         << l.libbib.autore  << ' '
         << l.libbib.editore << ' '
         << l.libbib.prezzo  << endl;
  if(l.presente)
    output << "Libro disponibile" << endl;
  else
    output << "Libro in prestito" << endl;
  return output;
};
...
libro l;
...
// Ora operazione legittima!!

cout << l;
```

Le funzioni ***friend*** anche se non sono metodi della classe hanno accesso a quanto dichiarato in private

Significato dell'operatore per oggetti della classe