Linguaggi di Programmazione

Ivano Salvo

OOP1: Introduzione agli Oggetti

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **3**, 13 ottobre 2020

Lezione 3a:

Introduzione agli Oggetti
e
primi passi in C++

Algorithms+Data Structures=Programs

Questo slogan di **Nicklaus Wirth** pone l'accento sull'importanza della **corretta progettazione** delle **strutture dati** nella programmazione.

Un tipo di dato è caratterizzato da un **insieme di valori** e da un insieme di **funzioni** che operano sui dati.

In C: con la `typedef` possiamo definire nuove strutture dati.

Tuttavia:

- Le funzioni vanno **definite a parte**
- Non c'è **nessun controllo** sul corretto utilizzo dei dati
- La responsabilità è tutta sulla **autodisciplina** del programmatore

Tipi Astratti di Dato

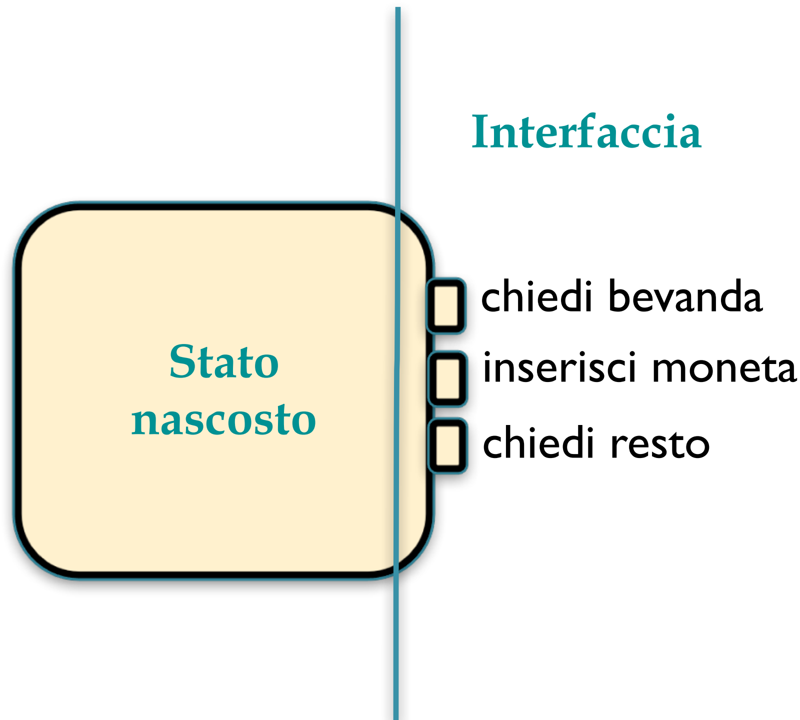
Incapsulamento + information hiding:

- La struttura dati e le funzioni che la manipola sono **raccolte in un'unica unità di programma**, detta **ADT** o **modulo** (**incapsulamento**)
- La struttura dati è manipolabile **solo dalle funzioni definite dentro l'ADT** o il modulo (**information hiding**)

Essenzialmente implementati solo in **ADA** e **Modula**, linguaggi di scarso successo. Sono stati superati subito dai Linguaggi Orientati agli Oggetti.

Oggetto come metafora

Potete immaginare un **automa** (ad es. la **macchina del caffè**) con uno **stato nascosto** (ad es. quantità di acqua, caffè, monete etc.) che si modifica a causa dell'interazione con l'esterno (pressione dei pulsanti, inserimento di monete) attraverso un'**interfaccia**:



Oggetto come metafora

❖ Il programmatore dovrà:

- Definire **la struttura dati** per memorizzare lo **stato** dell'oggetto
- Definire l'**interfaccia**, cioè le **interazioni** possibili con l'oggetto
- Questo viene, di norma, definito nelle **classi**.

❖ Di norma, lo stato è **acessibile solo attraverso l'interfaccia**. Le funzioni dell'interfaccia possono:

- **mostrare** informazioni (metodi **osservatori**)
- **calcolare** valori
- **modificare** lo stato dell'oggetto (metodi **mutatori**)
- **creare** nuovi oggetti (metodi **peoduttori**)

Iniziamo col C++: classi

L'esempio del corso sarà un'applicazione per **gestire una partita a scacchi** e verificare la **correttezza delle mosse**.

Cominciamo con i prototipi di un tipo per i **colori**:

```
class Colore{
   int c;

public:
   Colore();
   Colore(int x);
   Colore(const Colore& col);

   void cambiaColore();
   void print();
   bool bianco();
   bool nero();
   bool operator ==(Colore col);
}; //end class Colore
```

variabili di istanza

costruttori

metodi

Un colore è rappresentato da un intero. Ma questa rappresentazione è nota solo a chi scrive questa classe.

La parte dichiarata public è visibile fuori dalla classe

Classi e Tipi

Una volta definita una **classe**, possiamo usare il suo nome come un **qualsiasi tipo** e dichiarare variabili di quel tipo.

```
Colore c;
 /* definisce una variabile di tipo colore
  * e mette in esecuzione il costruttore
  * di default, quello senza parametri
  */
Colore d(0);
 /* mette in esecuzione il costruttore
  * con un parametro intero
  */
Colore* b = new Colore(1);
 /* mette in esecuzione il costruttore
  * con un parametro intero e restituisce un
  * pointer a un colore
  */
d.print() // invoca il metodo print su d
if (d==b) then // invoca il metodo == su d
```

Costruttori (1)

Il codice dei costruttori usualmente si scrivono in C++ **fuori dalla classe** (vedi sotto la sintassi).

Il codice va nel file `colore.cpp`, mentre la definizione della classe vista prima nel file `colore.h`.

I **costruttori** sono eseguiti **solo alla creazione** di un oggetto.

```
/* costruttore di default. Inizializza il
 * colore a nero
 */
Colore::Colore(){
   c=0;
}

/* Inizializza il colore in base al parametro */
Colore::Colore(int x){
    c=x;
}
```

Costruttori (2)

Un oggetto viene **creato quando viene “eseguita” la sua definizione**, oppure per effetto di **una new se il tipo della variabile è un pointer**.

I **costruttori hanno sempre il nome della classe**: posso scriverne più di uno con diverso numero o tipo di parametro (differenti segnature): **overloading**.

```
/* costruttore di default. Inizializza il
 * colore a nero
 */
Colore::Colore(){
   c=0;
}

/* Inizializza il colore in base al parametro */
Colore::Colore(int x){
   c=x;
}
```

Osservate che i due costruttori hanno lo stesso nome, ma diversi parametri: overloading

Metodi (1)

I **metodi** "assomigliano" alle funzioni, ma hanno un'importante differenza: essi vengono invocati **su un oggetto ricevente** (come premere un pulsante nell'interfaccia di un oggetto). Lo **stato privato** dell'oggetto ricevente è **visibile nei metodi della classe**

```
void Colore::print(){
   switch(c){
      case 0: printf("Nero\n"); break;
      case 1: printf("Bianco\n"); break;
      case 2: printf("Blue\n"); break;
      case 3: printf("Rosso\n"); break;
      case 4: printf("Verde\n"); break;
   default: printf("Incolore\n");
   }
}

b.print(); // stampa "Nero"
c.print(); // stampa "Bianco"
d->print(); // attenzione, d è un pointer
```

La variabile c è la variabile di istanza nella def della classe

Si chiamano i metodi su un oggetto usando la dot notation come per selezionare un sottocampo di una struct

Metodi (2)

Il metodo seguente serve a **invertire il colore**.

Nello stato che descrive la partita ci sarà una variabile `turno` di tipo `Colore` che mantiene informazione su chi ha la mossa.

Quando si inverte il colore, si eseguirà il codice:

```
turno.cambiaColore();
```

È importante che **non si acceda direttamente alla variabile c** per vari motivi: potremmo **distruggere l'integrità dei dati** oppure scrivere programmi difficili da capire perché **dipendono dalle codifiche**, piuttosto che dalla logica dei dati.

```
void Colore::cambiaColore(){
/* se il colore era nero (codifica 0)
 * diventa bianco (codifica 1) e viceversa
 */
   c = 1 – c;
}
```

Ridefinizione di operatori

Una simpatica caratteristica del C++ è la possibilità di **ridefinire operatori del linguaggio**, come =, ==, *, etc.

Pro: Questo rende omogeneo il trattamento dei tipi definiti dall'utente coi tipi predefiniti (`int`, `char`, …): **usare gli operatori coerentemente col significato usuale…**

Contro: può nascondere della complessità al programmatore. Ad esempio, su una lista, == non ha complessità $\mathcal{O}(1)$ ma magari proporzionale alla lunghezza della lista.

```
/* a parte la parola chiave 'operator'
 * è una definizione di metodo come le altre
 */
bool Colore::operator ==(Colore col){
   return c==col.c;
}
/* osservate che c (variabile di istanza)
 * del parametro col è accessibile come la
 * variabile c dell'oggetto ricevente
 */
```

Oggetti: Heap vs Stack (1)

Nella programmazione a **oggetti pura** (Java, Smalltalk) gli oggetti vengono **sempre** trattati con **variabili puntatore**!

C++ lascia la scelta al programmatore.

[**Personalmente**] ritengo che la scelta del C++ **generi una serie di pasticci** e costringa il programmatore a riflettere bene su cosa sia più appropriato fare.

```
Colore c;
Colore d(0);
 /* c e d sono oggetti di tipo Colore allocati
  * sullo stack, come una variabile int
  */

Colore* b = new Colore(1);
 /* b è allocato sull'Heap, come la memoria
  * allocata con malloc/calloc
  */
```

Oggetti: Heap vs Stack (2)

Ricordate che **tutto ciò che viene allocato sullo stack**, **viene deallocato quando finisce di eseguire la componente nella quale viene allocato** (tipicamente una **funzione** o un **metodo**).

Il **Vero Programmatore a Oggetti**, usualmente **pensa gli oggetti come persistenti**!

```
Colore f(){
   Colore c(1);
   return c;
 /* scorretto! c verrà deallocato quando f
  è finisce di eseguire */
}
Colore* f(){
   Colore* c = new Colore(1);
   return c;
 /* corretto! c vive sull'heap e ritorno un
  * pointer all'oggetto */
}
```

Lezione 2b:

Programmiamo un ADT in C++:

Numeri Razionali

Programmiamo l'ADT Razionali

La prima scelta da fare è la rappresentazione del tipo di dato.

Un **razionale** è chiaramente una **coppia di interi**.

Tuttavia, **molte coppie di interi rappresentano lo stesso razionale.** E molte coppie di interi **non rappresentano nessun razionale** (quelle con 0 come denominatore).

Usualmente è conveniente avere una **rappresentazione canonica**: converremo di rappresentare i razionali sempre ridotti ai minimi termini e il segno codificato nel numeratore. Questo sarà un **invariante di rappresentazione** (o di Tipo di Dato)

```
class Raz{
/* INV di ADT: MCD(num,den)=1 & den>0 */
   int num;
   int den;
…
}
```

Programmiamo l'ADT Razionali

A questo punto, potrebbero essere comode un metodo **`riduci`** che riduce un razionale ai minimi termini (forma canonica) e una funzione **`mcd`** che calcola il massimo comun divisore.

Il metodo **`riduci`** sarà **privato**: infatti **non appartiene all'interfaccia dei razionali**, ma serve come funzione di servizio.

La funzione **`mcd`** più probabilmente sarà una funzione esterna di una **libreria di utilità** matematiche.

Il fatto che le **variabili di istanza `num` e `den`** siano private, è essenziale per assicurarsi che gli invarianti di tipo di dato siano mantenuti: devo verificare **solo** che i **metodi produttori** e **mutatori** mantengano gli **invarianti**.

D'altro canto, potrò assumere gli **invarianti** di tipo di dato come **precondizione di ogni metodo** definito nella classe.

Completiamo l'interfaccia

```
class Raz{
    int num;
    int den;

    int mcd(int x, int y);
    void riduci();

public:
    Raz(int x, int y);
    Raz(int x); // costruisce un razionale intero
    Raz(); // costruisce il razionale 0

    bool operator==(Raz q);
    void per(Raz q);
    void piu(Raz q);
    void opposto();
    void inverti();
    void diviso(Raz q);
    void meno(Raz q);
    void print();
};
```

Le operazioni aritmetiche sono di tipo `void`*: implementiamo i razionali mutabili: i metodi modificano l'oggetto ricevente, invece di creare un nuovo numero razionale. Questa è una scelta critica da fare durante la progettazione*

Costruttori

```
Raz::Raz(int x, int y){
/* PREC: y!=0 */
   assert(y!=0);
   num=x;
   den=y;
   riduci();
}

Raz::Raz(int x){
   num=x;
   den=1;
}

Raz::Raz(){
   num=0;
   den=1;
}
```

*L'invocazione di **riduci** è necessaria per assicurare che l'invariante di tipo di dato valga alla creazione di un oggetto*

Questi oggetti sono soddisfano naturalmente l'invariante di tipo di dato

Alcuni metodi

```
void Raz::piu(Raz q){
   int d=den*q.den;
   num=d/den*num + d/q.den*q.num;
   den=d;
   riduci();
}
void Raz::inverti(){
   assert(num!=0);
   scambia(&num,&den);
}
void Raz::print(){
   printf("%d/%d ",num,den);
}
bool Raz::operator==(Raz r){
   return num==r.num && den==r.den;
}
```

Le variabili di istanza num *e* den *sono accessibili anche se appartengono all'oggetto parametro, perché di tipo Raz, la classe che sto definendo*

La funzione **==** *assume l'invariante di tipo di dato. Altrimenti sarebbe scorretta.*

Altri metodi

```
void Raz::diviso(Raz r){
/* notare che r è una copia
   r.inverti();
   per(r);
}
/* quindi anche:

void Raz::diviso(Raz r){
   r.inverti();
   this.per(r);
}
```

Un metodo può chiamare altri metodi: l'oggetto ricevente è implicitamente lo stesso (***this***)

Non ci sono side-effect su **r**, *in quanto, in accordo con la semantica call-by-value viene copiato alla chiamata della funzione*

Esempi d'uso e output

```
#include <stdio.h>
#include "raz.h"
int main(){
   Raz r(4,6); r.print();
   Raz *p = new Raz(3,6); p->print();
   p->piu(r); p->print();
   r.diviso(*p);
   p->print();
   r.print();
   Raz q(7,6);
   if (r == *p) printf("uguali\n");
       else printf("diversi\n");
   if (*p == q) printf("uguali\n");
       else printf("diversi\n");
}
```

```
Output:
2/3
1/2
7/6
7/6
4/7
diversi
uguali
```

Esercizi & Riflessioni

Provare a modificare l'implementazione dei razionali, definendoli come **tipo immutabile**: ogni operazione (`piu`, `meno`, `per`, `diviso`, etc.) deve generare un **nuovo razionale**.

Programmare usando solo **puntatori a oggetti** (sia nell'uso dei razionali, sia nei parametri delle funzioni `piu`, `meno`, `per`, `diviso`, etc.)

Verificare che **non ci sono side-effects**, anche se si passano puntatori a oggetti come parametri (gli oggetti non vengono quindi copiati).

Verificare invece che i side-effects ci sono **nella versione mutabile** vista nelle slides precedenti.

Provare a definire l'operatore = (**che tipo ha?**)

Lezione 3

That's all Folks…

Grazie per l'attenzione…

…Domande?