Laboratorio di programmazione

Lezione VIII

Tatiana Zolo

zolo@cs.unipr.it

PROGRAMMAZIONE A OGGETTI

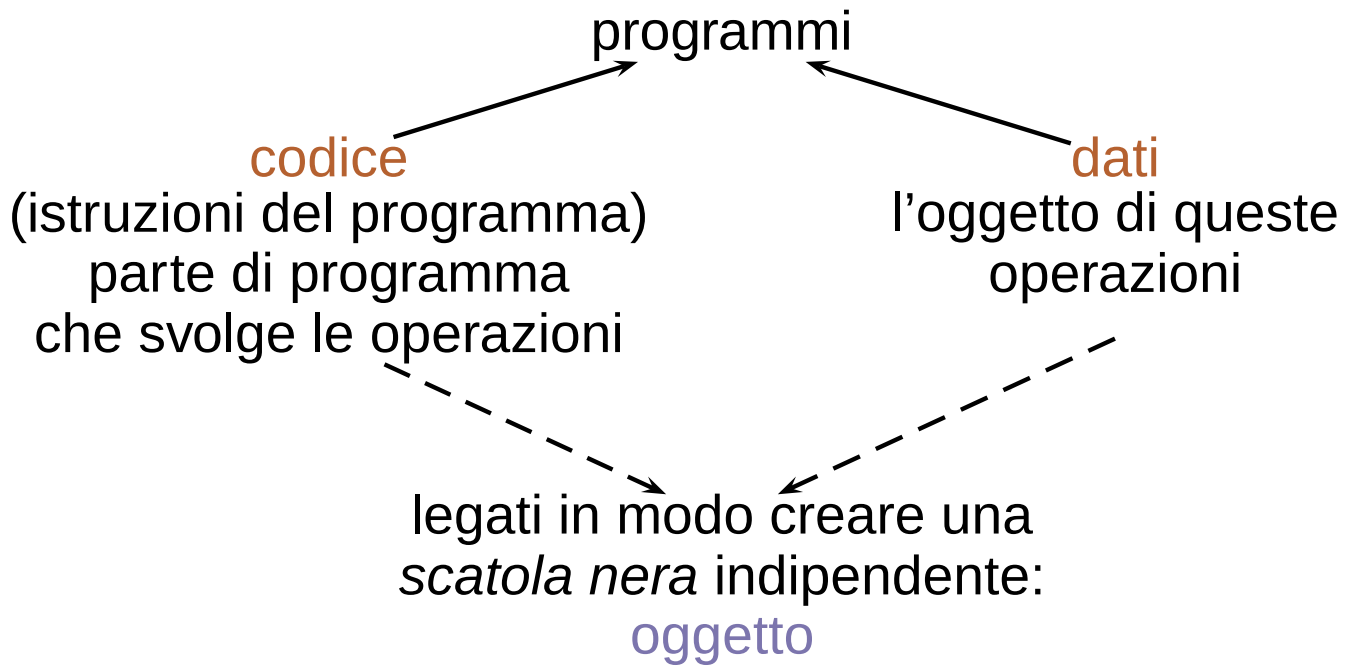
Quando si programma a oggetti

- ➜ si scompone il problema in sottogruppi di parti collegate che tengono conto sia del codice sia dei dati relativi a ciascun gruppo;
- ➜ i sottogruppi vengono organizzati in una struttura gerarchica;
- ➜ i sottogruppi vengono tradotti in unità indipendenti chiamate oggetti.

Tre caratteristiche comuni a tutti i linguaggi di programmazione a oggetti:

1. Incapsulamento o information hiding
2. Polimorfismo
3. Ereditarietà

1. Incapsulamento:



Codice e/o dati all'interno di un oggetto possono essere privati per quell'oggetto o pubblici.

2. Polimorfismo: “un’interfaccia, molti metodi”. È possibile creare un’interfaccia generica per un gruppo di attività collegate, per un insieme di azioni $\Longrightarrow$ riduzione della complessità.

2. Polimorfismo: “un’interfaccia, molti metodi”. È possibile creare un’interfaccia generica per un gruppo di attività collegate, per un insieme di azioni $\Longrightarrow$ riduzione della complessità.

3. Ereditarietà: processo mediante il quale un oggetto acquisisce le proprietà di un altro oggetto in base al concetto di classificazione gerarchica. Senza la classificazione gerarchica ogni oggetto dovrebbe definire esplicitamente tutte le sue caratteristiche. Con l’ereditarietà l’oggetto deve definire solo quelle caratteristiche che lo rendono unico: un oggetto può rappresentare un esempio specifico di una clsse più generale.

Definizione di Classe

Una definizione di classe C++ è costituita da due parti: testa della classe e corpo della classe

```
class NomeClasse {
public:
  // insieme pubblico delle operazioni
  // (metodi accessibili dall'esterno)
private:
  // implementazione privata: metodi e dati
  // accessibili solo ad altre parti della classe
} A, B;
```

Il nome della classe rappresenta un nuovo tipo di dato:

```
NomeClasse C;
```

Il corpo della classe definisce la lista dei membri della classe.

DATI MEMBRO

Sono dichiarati nello stesso modo in cui sono dichiarate le variabili.

```
class NomeClasse {
  int n;
  char c;
  char str[32];
};
```

Un membro non può essere inizializzato esplicitamente nel corpo della classe: si usano i costruttori!

```
class NomeClasse {
  int n = 0;      // errore!!
};
```

Per default i membri di una classe sono privati.

METODI O FUNZIONI MEMBRO

Possono essere specificati in due modi diversi:

1. definizione nel corpo della classe

```
class NomeClasse {
public:
  void nome_metodo(lista parametri) {
    // corpo del metodo
  }
};
```

METODI O FUNZIONI MEMBRO

2. definizione all'esterno del corpo della classe usando l'operatore di scope resolution ::

```
class NomeClasse {
public:
  void nome_metodo(lista parametri);
};

void NomeClasse::nome_metodo(lista parametri) {
  // corpo del metodo
}
```

METODI O FUNZIONI MEMBRO

Le funzioni membro differiscono dalle funzioni ordinarie per le seguenti proprietà.

1. Sono dichiarate nel campo d'azione della classe, quindi il nome di una funzione non è visibile all'esterno dello scope della sua classe.

   Ci si riferisce ad una funzione membro usando l'operatore punto:

   ```
   NomeClasse A;
   A.nome_metodo();
   ```

2. I metodi hanno privilegi di accesso completo ai membri sia pubblici sia privati della classe mentre, in generale, le funzioni ordinarie hanno accesso soltanto ai membri pubblici.

COSTRUTTORI E DISTRUTTORI

- Costruttore: permette agli oggetti di inizializzare se stessi al momento della creazione.

  Funzione membro speciale con stesso nome della classe; non ha tipo di ritorno.

  È applicato automaticamente dal compilatore ad ogni oggetto prima che questo venga usato.
  - Costruttore di default: invocato senza che l'utente specifichi un argomento.
  - Costruttore con parametri.

Costruttori e Distruttori

- Costruttore: permette agli oggetti di inizializzare se stessi al momento della creazione.

  Funzione membro speciale con stesso nome della classe; non ha tipo di ritorno.

  È applicato automaticamente dal compilatore ad ogni oggetto prima che questo venga usato.
  - Costruttore di default: invocato senza che l'utente specifichi un argomento.
  - Costruttore con parametri.
- Distruttore: spesso è necessario che un oggetto esegua un'azione o una serie di azioni quando viene distrutto (es. deallocazione della memoria precedentemente allocata).

  Funzione membro speciale con stesso nome della classe preceduto da una tilde ~; non ha tipo di ritorno.

  Viene invocato ogni qualvolta un oggetto esce dal campo d'azione.

Costruttori e Distruttori

```
class Tempo {
public:
  Tempo();
  Tempo(int H, int M, int S);
  ~Tempo();
private:
  int secondi;
};

Tempo::Tempo() {
  secondi = 0;
}
```

COSTRUTTORI E DISTRUTTORI

```
Tempo::Tempo(int H, int M, int S) {
  if (H < 0 || H > 24 || M < 0 || M > 60 || S < 0 || S > 60)
    secondi = 0;              // e' una scelta!
  else
    secondi = H*3600 + M*60 + S;
}

Tempo::~Tempo() {
  // eventuali operazioni in chiusura, in questo caso nessuna.
}
```

Nel main:

```
Tempo T1;   // invocazione del costruttore di default
Tempo T2(21,0,0);
T1 = Tempo(12,55,49);
```

OVERLOADING DI OPERATORI

```
Tempo T1;
Tempo T2(2,15,0);
```

Dati due oggetti `Tempo` voglio poter scrivere `T1+T2` (e non `somma(T1, T2)` dove `somma()` è una funzione...).

```
Tempo operator+(Tempo T) {
  Tempo tmp;
  tmp.secondi = (T.secondi + secondi) % 86400;
  return tmp;
}
```

Ora posso scrivere

```
T1 = T1 + T2;
```

che equivale a scrivere

```
T1 = T1.operator+(T2);
```

OGGETTI CONST

Oggetti costanti: ogni tentativo di modificare quel valore dal programma provoca un errore a tempo di compilazione:

```
const int n = 100;
const Tempo T(10,30,0);

// non modifica T
void funzione1(const Tempo& T);

// eventuali modifiche di T locali al metodo
void funzione2(Tempo T);

// eventuali modifiche di T si applicano anche
// all'oggetto originale
void funzione3(Tempo& T);
```

METODI CONST

Per rispettare la caratteristica `const` di un oggetto, il compilatore deve distinguere tra metodi insicuri (che tendono a modificare l'oggetto) e metodi sicuri (che non tentano di modificarlo) che sono detti metodi costanti.

```
// Puo' modificare l'oggetto di invocazione
void metodo1(const Tempo& T);

// Non puo' modificare l'oggetto di invocazione
void metodo2(const Tempo& T) const;
```

È bene dichiarare `const` i metodi che non modificano i dati della classe.

ESERCIZI

1. Costruire la classe Razionale con due dati privati che rappresentano numeratore e denominatore (non dimenticate costruttore/i e distruttore!).

   Scrivere un programma che chieda all'utente di inserire due numeri razionali (1° modo: inserendo numeratore e denominatore di ogni numero separatamente; 2° modo: nella forma n/d facendo l'overloading dell'operatore $>>$) e il simbolo di operazione che si vuole eseguire (somma, sottrazione, moltiplicazione o divisione). Queste operazioni verranno eseguite utilizzando l'overloading di operatori ($+$, $-$, $*$, $/$).

   Restituire i risultati (1° modo: stampando numeratore e denominatore di ogni numero separatamente; 2° modo: nella forma n/d facendo l'overloading dell'operatore $<<$) semplificati, cioè, le frazioni devono essere ridotte ai minimi termini.

   Ripetere l'esercizio fintanto che il risultato dell'operazione risulta positivo (overloading dell'operatore $>$ o dell'operatore $<$). (`cl_razionale_operazioni.cpp`.)

ESERCIZI

2. Definire la classe Tempo con un unico dato privato, sec, che è l'equivalente in secondi di una tripla (ore, minuti e secondi).
   Definire i seguenti metodi (oltre costruttori e distruttore):
   - ➜ "ore()": restituisce l'ora (a partire da sec);
   - ➜ "minuti()": restituisce i minuti (a partire da sec);
   - ➜ "secondi()": restituisce i secondi (a partire da sec);
   - ➜ "operator+()": somma di due ore del giorno, quindi di due oggetti "Tempo" (il risultato deve essere sempre un oggetto "Tempo");
   - ➜ "operator==()": confronta due oggetti "Tempo".
   - ➜ quello che volete...

   Esempio: il dato privato sec contiene 7812. Allora i primi 3 metodi restituiranno rispettivamente 2, 10, 12.

   (`tempo.cpp`.)