PROGRAMMAZIONE AVANZATA

2 Classi e oggetti

Programmazione orientata agli oggetti

Una **classe** rappresenta una famiglia omogenea di entità che:

- hanno la stessa rappresentazione in memoria
- supportano le stesse operazioni

Esempio

- i numeri razionali sono rappresentati da una coppia di numeri interi, di cui il secondo è positivo
- i numeri razionali possono essere sommati, sottratti, moltiplicati, ecc.

Un **oggetto** rappresenta una singola entità della classe ed è caratterizzato da:

- la classe di appartenenza
- un identificatore (nome/indirizzo in memoria)
- una regione di memoria che contiene la sua rappresentazione

Principio di incapsulamento

Ogni classe ha una parte **privata** e una **pubblica**:

- La parte **privata** è accessibile/utilizzabile solo dall'interno della classe.
- La parte **pubblica** è accessibile/utilizzabile ovunque nel programma.

Perché l'incapsulamento è utile?

1. Facilita il riuso e la modifica del codice: la parte privata di una classe può cambiare indipendentemente dal resto del programma.
2. Impedisce accessi diretti alla parte privata di una classe, prevenendo errori involontari o maliziosi.

Esempio di accesso diretto

```
rat a = rational(1, 2);
a.den = 0;              // accesso diretto al campo den
```

La classe `rat`

```
class rat {
private:
   ...

public:
   ...
};
```

Note

- La parola chiave `class` introduce una nuova **classe** e un nuovo **tipo**
- Il nome `rat` può essere usato ovunque può comparire un tipo
- La classe contiene **membri** (per ora omessi e sostituiti da `...`) che includono **campi** (dati) e **metodi** (operazioni)
- I membri **privati** sono accessibili solo dall'interno della classe
- I membri **pubblici** sono accessibili da ogni parte del programma
- Possono esserci più blocchi pubblici e privati, in qualunque ordine

Definizione di campi

```
class rat {
private:
  int num;
  int den; // den > 0

public:
  ...
};
```

Note

- I **campi** specificano la rappresentazione degli oggetti di tipo `rat`
- Qui abbiamo due campi di tipo `int` (numeratore, denominatore)
- In generale, i campi possono essere in numero e di tipo arbitrario
- Il commento accanto a `den` specifica un **invariante di classe**: ci aspettiamo che nel ciclo di vita di un oggetto di tipo `rat` la condizione `den > 0` sia sempre verificata, ma è responsabilità del programmatore fare in modo che questa proprietà sia vera

Costruttore di una classe

```
class rat {
  ...
public:
  rat(int, int = 1); // dichiarazione del costruttore
  ...
};

rat::rat(int a, int b) { // definizione del costruttore
  if (b > 0) {
    num = a;
    ... // stesso codice della funzione rational
}
```

- Il **costruttore** realizza un'operazione speciale eseguita al momento della **creazione** di un oggetto di tipo `rat`
- La **dichiarazione** viene fatta all'interno della classe
- La **definizione** può essere fatta all'interno o (come qui) all'esterno
- Il nome del costruttore **coincide** con quello della classe
- La sintassi `rat ::` indica che definiamo il costruttore della classe `rat`
- Il costruttore **non ha un tipo di ritorno**

Ancora sul costruttore

```
rat::rat(int a, int b) {
  if (b > 0) {
    num = a;
    den = b;
  } else if (b < 0) {
    num = -a;
    den = -b;
  } else throw std::domain_error("division by zero");
}
```

Note

- La funzione del costruttore è quello di preparare l'oggetto al primo utilizzo, inizializzando opportunamente i campi della classe
- È responsabilità del costruttore garantire che eventuali invarianti di classe siano soddisfatti (o segnalare un **errore** se ciò è impossibile)
- Il costruttore si riferisce ai **campi** `num` e `den` dell'oggetto che sta inizializzando

Esempio: creazione di una istanza

```
int main() {
  rat a(1, 2); // crea il numero razionale 1 / 2
  rat b(3);    // crea il numero razionale 3 / 1
  ...
}
```

- a e b sono **oggetti** di tipo `rat` (anche: sono **istanze** della classe `rat`)
- Il costruttore di `rat` viene eseguito ogni volta che si istanzia `rat`
- Ogni istanza ha la sua copia (privata) dei campi `num` e `den`

Ogni tentativo di accedere ai campi `num` e `den` dall'esterno di `rat` viene segnalato come errore dal compilatore:

```
void test() {
  rat a(1, 2);
  std::cout << a.num << std::endl; // ERRORE!
}
```

Non possiamo fare nulla con `rat` se non aggiungiamo operazioni **pubbliche**!

Metodi

```
class rat {
  ...
public:
  int get_num() const { return num; }
  int get_den() const { return den; }
};
```

Note

- I **metodi** sono le operazioni eseguibili su istanze di una classe
- Qui **dichiariamo** e **definiamo** i metodi `get_num` e `get_den`, per leggere numeratore e denominatore di un `rat`
- Il qualificatore `const` indica che queste operazioni **non modificano** l'istanza su cui vengono invocate (si limitano a leggere dati)
- Metodi semplici (come `get_num` e `get_den`) possono essere definiti internamente alla classe
- Metodi complessi (es. `rat::rat`) sono solitamente definiti all'esterno

Invocazione di un metodo

```
void test() {
  rat a(1, 2);
  std::cout << a.get_num() << std::endl; // stampa 1
  std::cout << a.get_den() << std::endl; // stampa 2
}
```

Note

- **Invochiamo** il metodo `get_num` su a per ottenere il numeratore di a
- **Invochiamo** il metodo `get_den` su a per ottenere il denominatore di a
- L'oggetto che riceve la richiesta di eseguire una certa operazione (es. `get_num`) è detto **oggetto ricevente** (è quello a sinistra di `.`)
- Il metodo `get_num`/`get_den` accede al campo `num`/`den` dell'oggetto ricevente su cui è stato invocato

Sintassi generale di una invocazione di metodo

```
oggetto ricevente . nome metodo ( argomenti )
```

Metodi di accesso controllato

```
class rat {
  ...
public:
  int get_num() const { return num; }
  int get_den() const { return den; }
};
```

`get_num` e `get_den` sono **metodi di accesso controllato**

- i campi num/den sono privati ⇒ inaccessibili nel resto del programma
- i metodi `get_num`/`get_den` sono pubblici ⇒ usabili da chiunque
- i metodi `get_num`/`get_den` permettono di **leggere** num/den
- `private`+`public`+metodi di accesso = controllo fine su chi può fare cosa

Esempio

```
rat a(1, 2);
std::cout << a.get_num(); // posso leggere num
a.get_num() = 3;          // NON POSSO scrivere num
```

Metodi esterni

```
class rat {
  ...
  rat add(const rat&) const;
  rat neg() const;
};

rat rat::add(const rat& b) const {
  return rat(num * b.den + b.num * den, den * b.den);
}

rat rat::neg() const {
  return rat(-num, den);
}
```

- **Dichiariamo** (internamente) e **definiamo** (esternamente) `add` e `neg`
- Per riferirsi ai campi di un oggetto diverso da quello ricevente (es. `b`) occorre specificare l'oggetto in questione (es. `b.num`)
- È possibile accedere al campo privato `num` dell'oggetto `b` poiché:
  1. `rat::add` è un metodo della classe `rat`
  2. `b` è una istanza di `rat`

Metodi che invocano altri metodi

- Abbiamo già realizzato `add` e `neg`
- Per non duplicare codice, calcoliamo `a-b` come `a+(-b)`

```
class rat {
  ...
  rat sub(const rat&) const;
};

rat rat::sub(const rat& b) const {
  return add(b.neg());
}
```

Una spiegazione complicata per un concetto semplice

- Per invocare un metodo `m` sull'oggetto ricevente l'invocazione di un altro metodo, è sufficiente scrivere `m()` senza ricevente
- `add( ... )` invoca `add` sull'oggetto ricevente l'invocazione di `sub`
- Volendo essere **espliciti** si può scrivere `this→add(b.neg())`
- `this` è un puntatore all'oggetto ricevente (non lo useremo mai, o quasi)

Overloading di operatori

```
rat operator+(const rat& a, const rat& b) {
  return a.add(b);
}

std::ostream& operator<<(std::ostream& os, const rat& a) {
  if (a.get_den() == 1) os << a.get_num();
  else os << a.get_num() << " / " << a.get_den();
  return os;
}
```

- L'**overloading** ci permette di definire il significato degli operatori + e << (ed eventualmente altri) quando li usiamo con istanze di `rat`
- Queste sono funzioni, **non** metodi di `rat`

Esempio

```
int main() {
  rat a(1, 2);
  rat b(3, 4);
  std::cout << a + b << std::endl; // stampa 10 / 8
}
```

Metodi privati

Supponiamo di voler dotare la classe `rat` di un secondo costruttore, sulla falsariga di quanto fatto nella libreria di funzioni.

```
class rat {
   ...
public:
  rat(int, int = 1);
  rat(double, int);
};

rat::rat(int a, int b) {
  if (b > 0) {
    num = a;
    den = b;
     ...
}

rat::rat(double a, int n) {
  int m = std::pow(10, n);
   ... ? ...
}
```

- Per realizzare il secondo costruttore vorremmo poter usare il codice già scritto per il primo, come fatto per la funzione `rat`
- Purtroppo, in C++ non è sempre facile chiamare un costruttore da un altro
- Idea: **fattorizzare** in un metodo ausiliario la parte comune dei due costruttori
- Tale metodo ausiliario, in quanto tale, è di interesse esclusivamente per la realizzazione della classe, ma non per i suoi utilizzatori. Può essere dichiarato **privato**.

Costruttori con inizializzatore privato

```
class rat {
private:
  int num;
  int den;
  // init è privato
  void init(int, int);

public:
  rat(int, int = 1);
  rat(double, int);
  ...
};

rat::rat(int a, int b)
{
  init(a, b);
}
```

```
void rat::init(int a, int b) {
  if (b > 0) {
    num = a;
    den = b;
  } else if (b < 0) {
    num = -a;
    den = -b;
  } else throw ...
}
```

```
rat::rat(double a, int n)
{
  int m = std::pow(10, n);
  init((int) (a * m), m);
}
```

Ciclo di vita di un oggetto

```
...
{
  // qui f non esiste ancora
  ...
  rat f(1, 2);
  // da qui in avanti è possibile usare f
  ...
  // qui f viene "distrutto"
}
// qui f non esiste più
...
```

Note

- Un oggetto esiste dal punto in cui è dichiarato fino alla fine del blocco in cui è dichiarato
- È possibile creare oggetti che “sopravvivono” anche dopo che l’esecuzione continua oltre il blocco in cui sono stati creati. Questo meccanismo prende il nome di **allocazione dinamica della memoria** (non avremo tempo per illustrarlo in questo corso).

Esercizi

1. Completare la classe `rat` realizzando metodi (esterni) corrispondenti a tutte le operazioni considerate nella libreria di funzioni.
2. Definire gli operatori overloaded $=$, $\neq$, $<$, $>$, $\leqslant$, $\geqslant$ per la classe `rat`.
3. Definire una classe `Complex` per rappresentare numeri complessi e dotarla di metodi per leggere parte reale e immaginaria e per eseguire le seguenti operazioni: somma, sottrazione, negazione, moltiplicazione, divisione, modulo, reciproco, coniugato. Definire operatori overloaded `+`, `-`, `*`, `/`, `<<` per numeri complessi.
4. Si supponga di dover definire una classe `Time` per modellare un'ora del giorno e che la classe debba fornire i seguenti metodi

   ```
   int  Time::get_hour() const; // ora
   int  Time::get_mins() const; // minuti
   int  Time::get_secs() const; // secondi
   Time Time::elapse(int s) const;
   ```

   in cui l'ultimo metodo fa avanzare l'ora di `s` secondi. Quale potrebbe essere una rappresentazione ragionevole per questa classe?