30. Alcune definizioni (avanzate)

Corso di Informatica

Corso di Laurea in Matematica (D.M. 270/04) - A.A. 2020/2021
Angelo Cardellicchio
angelo.cardellicchio@uniba.it
16/12/2020

Outline

- La parola riservata **`this`**
- Overloading degli operatori
- Allocazione dinamica delle risorse

Esercizio – Parte 1

- Creiamo una classe **Vettore** contenuta all'interno del namespace **Geometria**. La classe è caratterizzata da due attributi, ovvero **modulo** ed **angolo** (quest'ultimo relativo all'asse delle ascisse). Definiamo inoltre i metodi **getter** e **setter** per ciascuno di questi attributi, oltre al **costruttore di default** ed ad un **costruttore parametrizzato** che accetta modulo ed angolo. Consideriamo argomenti di tipo **`double`**.

15 minuti

La parola riservata `this`

- Definisce uno speciale puntatore che punta ad una **specifica istanza** della classe
- L'inizializzazione è automatica, e viene passato in maniera **implicita** ad ogni funzione membro
- Un uso tipico è nei metodi setter, ma è anche importante nella copia di oggetti

- *Usando* **`this`** *ci si assicura di riferirsi all'attributo* **`nome`** *di quella specifica istanza di* **`PersonaBase`***!*
- ***La notazione da usare è quella dei puntatori***

```
// persona.cpp
void setNome(string nome) {
    this->nome = nome;
}
```

Esercizio – Parte 2

- Modifichiamo la classe **`Vettore`** in modo che i getter ed i setter utilizzino la parola chiave **`this`**, e testiamone il funzionamento nel **`main`**.

5 minuti

Overloading degli operatori (1)

- Il C++ considera gli operatori applicati ai tipi non primitivi come delle **funzioni**
  - *Ad esempio, come fare a sommare il modulo di due vettori?*
  - *Possibile risposta: usando la regola del parallelogramma…*
  - *…mediante l'overloading dell'operatore +!*
- Scendiamo nei dettagli. Il compilatore C++ tratta gli operatori sui tipi primitivi come segue:

```
a X_bin b;   ⟶   a.operatorX_bin(b);

X_un a;      ⟶   a.operatorX_un();
```

Overloading degli operatori (2)

- Tornando all'esempio dei due vettori, possiamo usare la regola del parallelogramma.
- Definiamo quindi un apposito overload per l'operatore **+**, usando la seguente sintassi.

```
// geometria.h
Vettore operator+(const Vettore& op);

// geometria.cpp
Vettore Vettore::operator+(const Vettore& op) {
    double res_angolo =
           abs(angolo - op.angolo);
    double norma = (modulo * modulo)
           + (op.modulo * op.modulo) -
           (2 * modulo * op.modulo *
                      cos(angolo));
    return Vettore(sqrt(norma),
           (angolo + op.angolo)/2);
}
```

```
// main.cpp
Vettore v1(10.0, 60.0);
Vettore v2(5.0, 30.0);
Vettore v3 = v1 + v2;
```

Overloading degli operatori (3)

- Nel caso visto in precedenza, l'operatore **+** viene messo in overload mediante una funzione membro
- Immaginiamo di voler sommare un valore al modulo del vettore, e di non voler usare, per qualche ragione, il metodo **`setModulo()`**
- Possiamo definire un altro overload dell'operatore **+**?

```
// geometria.h
Vettore operator+(double modulo);
// geometria.cpp
Vettore operator+(double modulo) {
    return Vettore(this->modulo + modulo, this->angolo);
```

Overloading degli operatori (4)

- Dobbiamo verificare la proprietà di **commutatività** della somma
- Ciò significa che entrambe le seguenti devono essere valide:

```
Vettore v1(10.0, 60.0);
Vettore v2 = v1 + 5;
Vettore v3 = 5 + v1;
```

- Se proviamo ad eseguire la seconda istruzione, avremo un errore.
- Per capire il perché, dobbiamo ricordare come vengono interpretati gli operatori quando li definiamo in overload come funzioni membro.

Overloading degli operatori (5)

```
a X_bin b;   ⟶   a.operatorX_bin(b);

X_un a;      ⟶   a.operatorX_un();
```

```
Vettore v1(10.0, 60.0);
Vettore v2 = v1 + 5;     // Ok, v1.operator+(5) è definito
Vettore v3 = 5 + v1;     // Errore, 5.operator+(v1) non è definito!
```

- Per questo motivo, abbiamo la possibilità di definire degli operatori in overload **esternamente** alla classe

Overloading degli operatori (6)

- Definendo un overload di un operatore esternamente alla classe, il compilatore interpreta la funzione come segue

```
a X_bin b;   ⟶   operatorX_bin(a, b);

X_un a;      ⟶   operatorX_un(a);
```

- Per gli operatori unari, basta definire una funzione con argomento un riferimento ad un'istanza della classe considerata
- Per gli operatori binari, sono possibili due approcci

Overloading degli operatori (7)

- Il primo, e più semplice, è quello di usare due overload, uno per ogni possibile commutazione

```
// geometria.h
Vettore operator+(Vettore& right, double left);
Vettore operator+(double right, Vettore& left);


// geometria.cpp
Vettore Geometria::operator+(Vettore& right, double left)
{
    return Vettore(right.getModulo() + left, right.getAngolo());
}
Vettore Geometria::operator+(double right, Vettore& left)
{
    return Vettore(left.getModulo() + right, left.getAngolo());
}
```

Overloading degli operatori (8)

- Il primo, più raffinato, consiste nel creare un ulteriore costruttore, che accetti soltanto il modulo, ed usarlo nell'operatore in overload

```
// geometria.h
Vettore::Vettore(double modulo);
Vettore operator+(Vettore& right, Vettore& left);
// geometria.cpp
Vettore::Vettore(double modulo) {
    setModulo(modulo);
    setAngolo(0);
}
Vettore operator+(Vettore& right, Vettore& left) {
    return Vettore(right.getModulo() + left.getModulo());
}
```

Overloading degli operatori (9)

- Possiamo anche effettuare l'overloading degli operatori di inserimento ed estrazione dallo stream (ovvero gli operatori `<<` e `>>`)
- In questo caso, ricordiamo che bisogna restituire un riferimento allo stream (ovvero, un valore di tipo **`ostream&`** per lo stream di output, od **`istream&`** per lo stream di input)

```
// geometria.h
ostream& operator<<(ostream& output, Vettore& v);
istream& operator>>(istream& input, Vettore& v);
```

Overloading degli operatori (9)

- L'idea è quella di definire dei metodi che possano semplificare l'acquisizione e la visualizzazione dei parametri necessaria definire un **`Vettore`**

```
// geometria.cpp
ostream& Geometria::operator<<(ostream& output, Vettore& vettore)
{
    cout << "Modulo: " << vettore.getModulo() << " Angolo: " << vettore.getAngolo();
    return output;
}
istream& Geometria::operator>>(istream& input, Vettore& vettore)
{
    double modulo;
    double angolo;
    input >> modulo >> angolo;
    vettore.setModulo(modulo);
    vettore.setAngolo(angolo);
    return input;
}
```

Esercizio – Parte 3

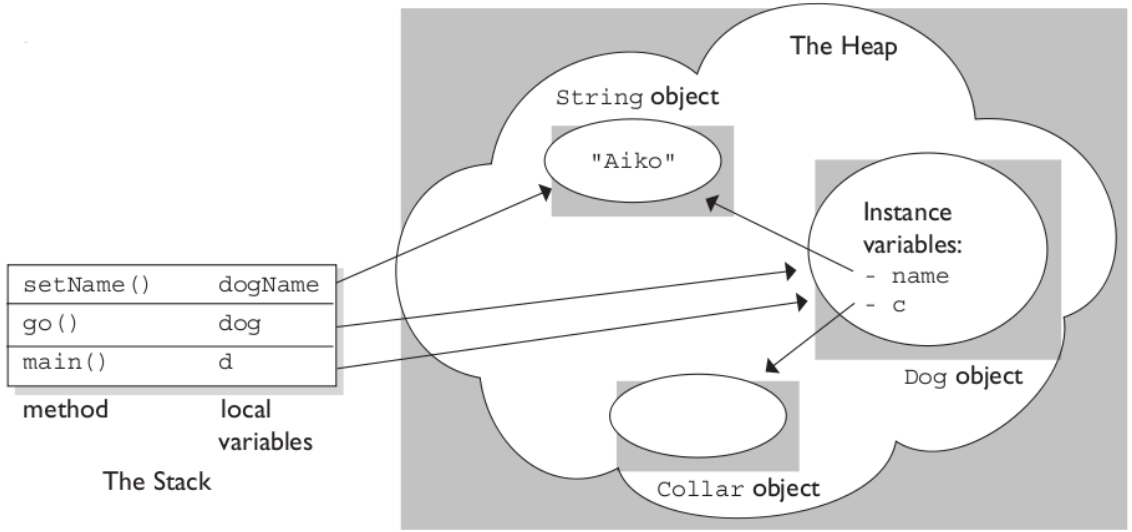
- Modifichiamo la classe **`Vettore`** in modo da inserire i seguenti overload su operatori:
  - *Overload su operatore di inserimento da stream*
  - *Overload su operatore di estrazione da stream*
  - *Overload dell'operatore + per la somma di due vettori*
- Verifichiamone il funzionamento nel **`main`**

15 minuti

Allocazione dinamica delle risorse (1)

- Abbiamo visto che esistono diverse zone di memoria
- Quelle che dobbiamo necessariamente considerare sono lo **stack** e l'**heap**
- Lo **stack** usa una strategia LIFO, ed è estremamente veloce, ma di dimensioni limitate; l'**heap** invece ha dimensioni maggiori, ma non forza alcun 'pattern' di accesso ai dati, per cui è più complesso (e lento!)
- Lo stack viene usato per l'allocazione **statica** delle variabili (quella che abbiamo usato finora), mentre l'heap per l'allocazione **dinamica**
- L'allocazione dinamica della memoria richiede strategie ben definite, sia nel C (funzioni **`malloc`**, **`calloc`** e **`free`**) sia nel C++ (operatori **`new`** e **`delete`**)

Allocazione dinamica delle risorse (2)



Fonte

Allocazione dinamica delle risorse (3)

- Ad esempio, nel seguente codice allochiamo le variabili in maniera statica, agendo sullo stack
- Notiamo che allocazione e deallocazione della memoria sono gestite in maniera **automatica**

```
int main()
{
    int a = 0;              // stack: {a}
    {
        int b = a++;        // stack: {b} {a}
    }                       // stack: {a}
    return 0;
}
```

Allocazione dinamica delle risorse (4)

- Nel seguente codice si usa l'allocazione dinamica mediante gli operatori **new** e **delete**
- **Si agisce su puntatori**

```
int main()
{
    int * puntatore;          // Dichiaro un puntatore
    puntatore = new int;      // Alloco memoria nello heap il puntatore con l'operatore new
    delete p;                 // Dealloco la memoria riservata al puntatore mediante l'operatore delete
    return 0;
}
```

Allocazione dinamica delle risorse (5)

- L'uso dei puntatori nell'allocazione dinamica delle risorse è necessario per mantenere il riferimento all'oggetto; senza, non sapremmo come accedervi!
- Ci si potrebbe chiedere **perché** si usa la separazione della memoria in stack ed heap. La risposta è semplice: lo stack è utile, soprattutto nelle funzioni ricorsive, ma pone serie limitazioni all'accesso ai dati, in quanto li vincola ad una strategia LIFO
- Lo heap permette di svincolarsi da questa limitazione
- È quindi possibile fare a meno dello stack? **Sì, ma le performance ne risentirebbero**
- È quindi possibile fare a meno dell'heap? **No**

Domande?

42