PROGRAMMAZIONE AVANZATA

7 Classi generiche

Classi generiche

Com'è fatta la classe `Vector`?

La versione con cui abbiamo lavorato fino ad ora

```
class Vector {
private:
  std::vector<int> data;          // notare int

public:
  Vector(int);
  int operator[](int) const;      // notare il primo int
  int& operator[](int);           // notare il primo int
  ...
  int mul(const Vector&) const; // notare int
};
```

Problema

Se avessimo bisogno di vettori i cui elementi hanno tipo `float`, `double`, `rat`, `nat`, ... dovremmo **riscrivere** classi sostanzialmente identiche tranne per il tipo degli elementi del vettore

La classe `Vector` generica

```
template <typename T> class Vector {      // ∀T Vector è ...
private:
  std::vector<T> data;                    // notare T

public:
  Vector(int);                            // notare int
  T operator[](int) const;                // notare T e int
  T& operator[](int);                     // notare T e int
  Vector<T> add(const Vector<T>&) const;  // notare T
  T mul(const Vector<T>&) const;          // notare T
};
```

Note

- La classe è **quantificata universalmente** sul tipo `T` degli elementi
- Si usa `T` laddove serve indicare il tipo degli elementi del vettore
- Si continua a usare `int` (o un altro tipo) laddove serve indicare valori proprio di quel tipo (ad esempio, l'indice di un elemento)
- I metodi `add` e `mul` fanno riferimento allo **stesso** `T` della classe, specificando che è possibile calcolare somma/prodotto di **vettori omogenei** (cioè i cui elementi hanno lo stesso tipo `T`)

Definizione di metodi generici

```
template <typename T>
Vector<T>::Vector(int n) {
  if (n < 0)
    throw std::domain_error("invalid vector size");
  data.resize(n, T());
}

template <typename T>
T Vector<T>::operator[](int i) const {
  return data.at(i);
}
```

Note

- Davanti a ogni definizione di metodo di una classe generica occorre **ripetere** la dichiarazione `template` così come usata per la classe
- Prima di `::` occorre indicare il nome della classe **completo dei parametri di tipo** (es. `Vector<T>::Vector` significa "questo è il costruttore della classe `Vector<T>`")
- Si può usare `T()` per creare un **valore di default** di tipo `T`. Quando `T` è un tipo numerico (come `int`), il valore di default è 0

Definizione di add e <<

```
template <typename T>
Vector<T> Vector<T>::add(const Vector<T>& v) const {
  if (size() != v.size()) ... // come prima
  Vector<T> res(size());
  for (int i = 0; i < size(); i++)
    res[i] = data[i] + v[i];
  return res;
}

template <typename T>
std::ostream& operator<<(std::ostream& os,
                         const Vector<T>& v) {
  os << "[";
  for (int i = 0; i < v.size(); i++) os << " " << v[i];
  os << " ]";
  return os;
}
```

- Si usa l'operatore + per **sommare** valori di tipo T
- Si usa l'operatore << per **stampare** valori di tipo T
- Il tipo T in `Vector<T>` è **vincolato**, deve fornire le operazioni + e <<

Utilizzo del template `Vector`

```
Vector<int>   v(2);     // definisce un vettore di int
Vector<float> w(2);     // definisce un vettore di float
v[0] = 0;
v[1] = 1;
w[0] = 0.5;             // OK: w ha elementi di tipo float
w[1] = 2;               // OK: int viene promosso a float
std::cout << v;         // OK: stampa vettore di int
std::cout << w;         // OK: stampa vettore di float
std::cout << v.add(w); // NO: w non ha tipo Vector<int>
```

Note

- Possiamo **istanziare** il parametro di tipo `T` di `Vector` con ogni tipo che supporti le operazioni usate nella classe `Vector<T>` ovvero `+`, `*`, `<<`.
- Se definiamo una classe che supporta tali operazioni (es. `rat`), possiamo istanziare `T` con tale classe (es. `Vector<rat>`)
- Il compilatore controlla la **corrispondenza** tra parametri di tipo. Ad esempio, rifiuta l'invocazione `v.add(w)` in quanto `v` ha tipo `Vector<int>` e `w` ha tipo `Vector<float>`.

Vettori con lunghezza esplicita

```
template <typename T, int N>
class Vector {
private:
  std::vector<T> data;

public:
  Vector();
  T operator[](int) const;
  T& operator[](int);
  Vector<T,N> add(const Vector<T,N>&) const;
  T mul(const Vector<T,N>&) const;
};
```

Note

- `typename T` indica che `T` deve essere un **tipo** (es. `int` o `rat`)
- `int N` indica che `N` deve essere un **valore** di tipo `int` (es. `5`)
- usiamo il parametro `N` per indicare la **lunghezza** del vettore **nel suo tipo**
- Esempio: `Vector<float,5>` è un vettore di 5 elementi di tipo `float`

Costruttore e operatore [ ]

```
template <typename T, int N>
Vector<T,N>::Vector() {
  if (N < 0)
    throw std::domain_error("invalid vector size");
  data.resize(N, T());
}

template <typename T, int N>
T Vector<T,N>::operator[](int i) const {
  return data.at(i);
}
```

Note

- Si usa `N` laddove serve indicare la lunghezza del vettore
- Occorre comunque controllare che `N` sia non negativo

add con lunghezza esplicita

```
template <typename T, int N>
Vector<T,N> Vector<T,N>::add(const Vector<T,N>& v) const {
  Vector<T,N> res;
  for (int i = 0; i < N; i++)
    res[i] = data[i] + v[i];
  return res;
}
```

Note

- Usando lo stesso N indichiamo che la somma è possibile solo tra vettori aventi la **stessa lunghezza**
- A differenza delle precedenti versioni, non serve più controllare **durante l'esecuzione** che la lunghezza di v coincida con quella dell'oggetto ricevente
- Il compilatore controlla che le lunghezze coincidano **prima dell'esecuzione**

Funzioni generiche

```
template <typename T, int N, int M>
Vector<T,N+M> append(const Vector<T,N>& a,
                     const Vector<T,M>& b) {
  Vector<T,N+M> c;
  for (int i = 0; i < N; i++) c[i] = a[i];
  for (int i = 0; i < M; i++) c[i + N] = b[i];
  return c;
}
```

Note

- append è una **funzione generica** con 3 parametri e 2 argomenti
  - T è il tipo degli elementi dei vettori da concatenare
  - N è la lunghezza del primo vettore
  - M è la lunghezza del secondo vettore
- La **concatenazione** di due vettori a e b produce un nuovo vettore la cui lunghezza è la somma delle lunghezze di a e b
- È possibile eseguire semplici operazioni aritmetiche sui parametri

Esercizi

1. Definire i metodi `mul` e `sub` per la classe `Vector<T,N>` che calcolino il prodotto scalare e la differenza di due vettori. Si aggiungono dei vincoli al parametro di tipo T della classe `Vector`?
2. Definire una funzione generica `zip` che, dati un `Vector<T,N>` ed un `Vector<S,N>`, produca un `Vector<std::pair<T,S>,N>` i cui elementi sono le coppie degli elementi corrispondenti dei due vettori.
3. Definire la funzione generica `unzip` inversa di `zip` che, dato un vettore di coppie `Vector<std::pair<T,S>,N>`, produca una coppia di vettori `std::pair<Vector<T,N>,Vector<S,N> >`. Fare attenzione a lasciare uno spazio tra due > adiacenti!
4. **Impegnativo**: definire una classe `Matrix<T,R,C>` analoga a `Matrix` vista in precedenza parametrizzandola rispetto al tipo T degli elementi ed al numero di righe R e di colonne C.