Appunti Algoritmi e Strutture Dati

Luca Seggiani

15 Maggio 2024

Funzioni modello con più parametri

Si possono definire funzioni modello (template) con più tipi generici (o meglio, tipi modello):

```
template<class tipo_1, class tipo_2>
tipo_1 max(tipo_1 x, tipo_2 y) {
  return (x > y) ? x : y;
}

void main() {
  int b = 2; double c = 6;
  cout << max(3, b); //restituisce un int (tipo_1 = int)
  cout << max(3, c); //restituisce comunque un int (tipo_1 =
   int)
}
```

oppure, in un'altro esempio:

```
template<class tipo_1, class tipo_2, class tipo_3>
tipo_1 max(tipo_2 x, tipo_3 y) {
  return (x > y) ? x : y;
}

void main() {
  cout << max(3, 0.3); //ERRORE: cosa devo restituire?
  cout << max<int>(3, 0.3); //OK!
}
```

Dove la sintassi che abbiamo appena usato è quella dei **parametri espliciti**.

Parametri non-tipo

Come parametri dei template, si possono inserire anche parametri non-tipo. Un parametro non-tipo è semplicemente una variabile, il cui valore deve essere obbligatoriamente definito in una chiamata esplicita della funzione templatizzata. Può essere utile usare template con parametri non tipo, in quanto la funzione viene effettivamente ridefinita per ogni valore dei parametri non tipo definiti nella chiamata:

```
template<int n, double m>
void func(int x = n) {
  int y = m;
  int array[n];
  ...
}

void main() {
  func<1, 2>(8);
  func<2, 3>(6); //e' una funzione oggetto diversa dalla
   precedente
  ...
}
```

Questo meccanismo può essere utile come alternativa alle funzioni statiche, visto che ci permette di creare più "istanze" di funzione per diversi parametri non-tipo.

Parametri tipo e non-tipo

Si possono usare parametri tipo e parametri non-tipo contestualmente alla stessa definizione di funzione modello:

```
template<int a, class T>
int func(T x) {
  return x > n;
}

void main() {
  gt<2>(3); //risoluzione di T implicita
  gt<2, double>(3) //risoluzione di T esplicita
  //il valore di a va comunque specificato in ogni caso!
}
```

Funzioni modello con variabili statiche

Si possono usare variabili statiche all'interno delle funzioni modello: basta ricordare che ogni istanza della funzione (anche quelle su più tipi) ha la sua copia delle variabili statiche:

```
template<class tipo>
tipo maxT(tipo x, tipo y) {
  static int a = 0; a++;
  return (x >y) ? x : a;
}

void main() {
  maxT(2, 3); //1
  maxT(3, 5); //2
  maxT(0.1, 0.2); //1 - e' un altra variabile
}
```

Dichiarazione e definizione di tempalte

Non si può compilare una funzione modello se ne non se ne conoscono le chiamate (quali istanze andremo a realizzare?). Questo vale anche se definizione e chiamate si trovano su unità di compilazione diverse: non potremo compilare le unità. Se si definisce una funzione (o classe) templatizzata in un modulo, l'implementazione va per questo motivo inserita nell'header.

Classi template

Anche le classi possono essere definite come template: in questo caso è obbligatorio usare sempre chiamate esplicite.

1 Derivazionie (Ereditarietà)

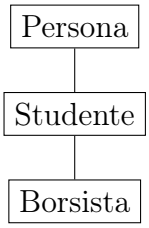
Il meccanismo della derivazione (o ereditarietà) permette di definire classi via via più specializzate sulla base di classi definite precedentemente. Nello specifico, si parla di classe **base** e classe **derivata**: la classe derivata mantiene un'insieme di caratteristiche comuni dalla classe base, senza che ciò comporti una duplicazione del codice, e offrendo la possibilità di adattare e specializzare la classe derivata rispetto alla classe base.

Gerarchia di classi

Possiamo definire una gerarchia per le classi, modellizzato graficamente da un'albero. Salendo l'albero (avvicinandosi alla radice), si parla di generalizzazione, spostandosi in senso opposto si parla di specializzazione. Facciamo un'esempio:

```
class Persona {
public:
  char nome[20];
  int eta;
}
class Studente : public Persona {
//tutti gli attributi di Persona sono gia' inclusi
pubic:
  int esami;
  int matricola;
}
class Borsista : public Studente {
//come sopra
public:
  int borsa;
  int durata;
}
```

La gerarchia sarà siffatta:



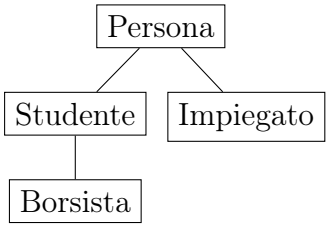
Potremo accedere, ad esempio da una classe di tipo borsista, a tutti gli attributi di studente:

```
Borsista b, *bp;
b.eta = 23;
bp->nome = "Francantonio";
...
```

Posso inoltre introdurre biforcazioni nella gerarchia, ad esempio come:

```
class Impiegato  : public Persona {
  int livello;
  int stipendio;
}
```

che cambia la nostra gerarchia in:



Conversioni implicite di classi derivate

Per le classi (e i puntatori a classe) derivati, si può convertire da una classe a un'altra più alta nella gerarchia (con perdita di informazione per le classi, nessuna perdita per i loro puntatori che rappresentano solo punti di accesso), ma non viceversa: si richiederebbe di creare informazione dal nulla!