

Ereditarietà e polimorfismo, ADT code

Lezione 17 di Programmazione 2

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2007-8

Indice

1. Ereditarietà e polimorfismo, ADT code
2. ereditarietà e derivazione di classi
3. funzioni virtuali e classi astratte
4. classe astratta per l'ADT union-find
5. implementazione dell'ADT union-find
6. l'ADT coda FIFO: interfaccia
7. implementazione della coda con un array
8. implementazione della coda con una lista
9. esercizi

ereditarietà e derivazione di classi

in C++, come in tutti i linguaggi di programmazione ad oggetti, è possibile organizzare le definizioni di classi in **gerarchie di ereditarietà**, dove ogni classe **derivata** eredita membri pubblici, **tranne i costruttori**, e **protected** dalle classi che la precedono nella gerarchia

l'ereditarietà si dice:

singola se la struttura gerarchica è ad albero

multipla se la struttura gerarchica è un grafo diretto aciclico (DAG)

la **derivazione immediata** di una classe **D**, detta **derivata**, da una classe **B**, detta **di base**, si codifica con la sintassi:

```
class D : public B { <definizioni di membri> };
```

in cui la specifica **public** mantiene inalterata la visibilità dei membri ereditati

è altrimenti possibile specificare **protected** oppure **private**, per i corrispondenti abbassamenti di visibilità dei membri ereditati

funzioni virtuali e classi astratte

in una classe derivata si può **ridefinire** il corpo di funzioni membro ereditate

se **D** eredita **f()** da **B** ma ne ridefinisce il corpo, allora l'invocazione **a.f()** esegue la definizione data in **D** se **a** è di tipo **D**; tuttavia, se **p** è un riferimento di tipo **B***, allora dopo l'assegnamento **p = &a**; l'invocazione **(*p).f()** esegue la definizione data in **B**, sebbene l'oggetto su cui si invoca **f()** sia "lo stesso", a meno che ...

... se una funzione è dichiarata **virtual**, allora le sue invocazioni su oggetti seguono la definizione data nel tipo dell'oggetto piuttosto che in quello determinato dal riferimento

questo fenomeno è detto **binding dinamico**

permette il **polimorfismo** dell'invocazione di funzioni

le funzioni virtuali **possono** essere ridefinite in classi derivate, tuttavia hanno in genere una definizione (di default) nella classe di base, a meno che ...

... una **funzione virtuale pura** non ha definizione nella classe di base, detta **astratta**, e **deve** essere definita nelle **classi concrete derivate** da essa

una classe è astratta se ha almeno una funzione virtuale pura, altrimenti è concreta
sintassi: **virtual <tipo> <nome_funzione> (<lista_tipi>) = 0;**

una classe astratta **non ha istanze**

un costruttore non può essere virtuale... **perché?**

classe astratta per l'ADT union-find

uf_interface.h : interfaccia astratta dell'ADT union-find

```
template <typename Elem>
class uf_interface {
public:
    virtual bool find(Elem, Elem) = 0;
    virtual void unite(Elem, Elem) = 0; };
```

uf.h : derivazione con aggiunta di un puntatore, accessibile alle classi derivate (la classe rimane astratta)

```
#include "uf_interface.h"
template <typename Elem>
class uf : public uf_interface<Elem> { protected: Elem* id; };
```

UFI.h : derivazione concreta dell'interfaccia per il tipo parametro **int**

```
#include "uf.h"
class UFI : public uf<int> {
public:
    UFI(int);
    bool find(int, int);
    void unite(int, int);
private:
    #include "UFI_private.h" };
```

implementazione dell'ADT union-find

UFI_private.h : strutture dati e funzioni private per l'algoritmo di quick-union pesata

```
int *sz;
int find(int x) { while (x != id[x]) x = id[x]; return x; };
```

UFI.cpp : implementazione per l'algoritmo di quick-union pesata

```
#include "UFI.h"
UFI::UFI(int N)
{ id = new int[N]; sz = new int[N];
  for (int i = 0; i < N; i++) { id[i] = i; sz[i] = 1; }
};

bool UFI::find(int p, int q)
{ return (find(p) == find(q)); };

void UFI::unite(int p, int q)
{ int i = find(p), j = find(q);
  if (i == j) return;
  if (sz[i] < sz[j])
    { id[i] = j; sz[j] += sz[i]; }
  else { id[j] = i; sz[i] += sz[j]; }
};
```

I'ADT coda FIFO: interfaccia

un'altra importante classe contenitore è la *coda*, ingl. *queue* : sequenza finita, di lunghezza variabile, di dati omogenei, ad accesso sequenziale con disciplina **FIFO**

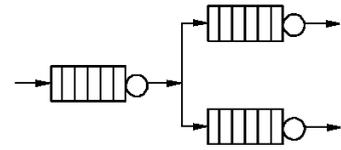
la coda è una struttura dati fondamentale:

nella **comunicazione**: canali di trasmissione

nella **concorrenza**:

gestione FIFO di richieste di servizio

reti di code, ingl. queueing networks



queue.h: come nel caso dell'interfaccia della pila, per esplorare implementazioni diverse dell'ADT con diverse strutture dati, usiamo una direttiva `#include` per la parte privata:

```
template <typename Elem> class Queue {
public:
    Queue();           // costruttore (capacità della coda illimitata)
    Queue(int);       // costruttore (parametro: capacità della coda)
    bool empty() const; // query: restituisce true sse la pila è vuota
    void put(Elem);   // inserimento di un elemento
    Elem get();       // rimozione di un elemento (restituito)
private:
    #include "queue-private.h"
};
```

implementazione della coda con un array

strutture dati private *queue-private.h*:

```
Elem *q; // puntatore alla struttura dati su cui si implementa la coda
int N;   // capacità della coda + 1
int head, tail; // indici degli estremi della coda
```

implementazione delle operazioni *queue.c++* : per il costruttore si richiede il parametro

```
#include "queue.h"
template <typename Elem>
Queue<Elem>::Queue(int maxN)
{ q = new Elem[maxN+1];
  N = maxN+1; head = N; tail = 0; }
template <typename Elem>
bool Queue<Elem>::empty() const
{ return head % N == tail; }
template <typename Elem>
void Queue<Elem>::put(Elem elem)
{ q[tail++] = elem; tail = tail % N; }
template <typename Elem>
Elem Queue<Elem>::get()
{ head = head % N; return q[head++]; }
```

implementazione della coda con una lista

strutture dati private `queue-private.h`:

```
struct lista { Elem elem; lista* next;
              lista(Elem e) {elem = e; next = 0; } };
typedef lista *plink; plink head, tail;
```

implementazione delle operazioni `queue.c++`: le due varianti del costruttore sono simili

```
#include "queue.h"
template <typename Elem> Queue<Elem>::Queue() { head = 0; }
template <typename Elem> Queue<Elem>::Queue(int) { head = 0; }
template <typename Elem>
bool Queue<Elem>::empty() const
{ return head == 0; }
template <typename Elem>
void Queue<Elem>::put(Elem elem)
{ plink t = tail; tail = new lista(elem);
  if (head == 0) head = tail; else t->next = tail; }
template <typename Elem>
Elem Queue<Elem>::get()
{ Elem e = head->elem; plink t = head->next;
  delete head; head = t; return e; }
```

esercizi

1. codificare l'implementazione dell'ADT union-find per altri algoritmi fra quelli visti nella prima lezione, ad es. quick-find e quick-union
2. usando l'interfaccia `UFI.h` dell'ADT union-find, codificare un programma che accetti da linea di comando un intero N e generi quindi una sequenza casuale di coppie di interi non negativi minori di N finché il grafo (non orientato) così costruito abbia una sola componente connessa, producendo infine in output il numero M di coppie generate; valutare quindi empiricamente le prestazioni, per valori crescenti di N , delle diverse versioni di tale programma ottenute usando diverse implementazioni dell'ADT in questione, ovvero quella vista qui per l'algoritmo quick union pesata e quelle prodotte nell'esercizio precedente
3. esercizio 4.22 del testo (Sedgwick, 2003), a p.158, adattato al caso della coda: modificare l'interfaccia della coda sostituendo `empty` con `count`, che restituisce il numero di elementi nella coda, e implementarla usando sia array che liste concatenate
4. esercizi 4.38 e 4.39 (modificato) del testo (Sedgwick, 2003), a p.174: modificare le due implementazioni della coda viste sopra in modo che si invochi una funzione membro `error()` quando il client invoca una `get` su una coda vuota o una `put` su una coda piena