C++

Alcuni cenni sulla progettazione di classi e librerie

Paolo Cignoni
cignoni@iei.pi.cnr.it
3/3/98

Classi astratte e concrete

- Le classi concrete che rappresentano un concetto relativamente semplice con tutte le operazioni essenziali per il supporto di quel concetto (e.g. il tipo **complex**)
  - matching concetto-implementazione
  - non mirate ad essere base classes

Classi astratte e concrete

- Le classi astratte sono espressione di concetti generali da cui classi più specifiche possono essere derivate
  - può essere usata solo come *base class* da cui derivare altre classi
  - non si può creare alcun oggetto di una classe astratta

Classi astratte esempio

```
class shape {            // classe astratta
// ..
public:
    virtual void rotate (int) = 0;
    virtual void draw () = 0;
};
// classe derivata concreta
class circle : public shape {
     int radius;
public:
     void rotate(int){} //overrides shape::rotate
     void draw();
}
```


Virtual Function

- Una virtual function è una funzione membro che ci si aspetta che sia ridefinita in una classe derivata.

Virtual Function

- Una chiamata ad una *virtual function* è risolta secondo il tipo soggiacente dell'oggetto per cui è chiamata.
- Una chiamata ad una *nonvirtual* function è risolta secondo il tipo del puntatore da cui è chiamata.

Virtual Functions

```
#include <iostream.h>

class Base {
public:
   void print() {cout << "base class\n"; }
   virtual void virtprint() {cout << "base (virtual)\n";
   }
};

class Derived : public Base{
public:
   void print(){cout << "derived class\n"; }
   void virtprint(){cout << "derived (virtual)\n"; }
};

void main() {
   Base b;
   Derived d;
   Base *bp = &d;

   bp->print();
   bp->virtprint();
}
```

Pure virtual

- le pure *virtual functions* sono **dichiarate,** ma **non definite** nella base classe, dovranno essere definite nelle classi derivate.
- una virtual function è resa pura aggiungendo un "`= 0`" alla def.

```
virtual void draw () = 0;
```

8

Classe astratta

- classe astratta: classe che contiene almeno una funzione *pure virtual*
- *e' vietato dichiarare una variabile del tipo di una classe astratta!*

9

A tree class

- Implementazione di un albero binario in c++
- Due classi
  - una classe astratta per i nodi
  - una classe per l'albero

10

I nodi

- Alcune cose da decidere
  - memorizzazione dati
  - ordine tra gli oggetti
  - processing sugli elementi

11

Memorizzazione dati

- Quando si aggiunge un oggetto all'albero, ci sono due possibilità:
  - si mette l'oggetto stesso (un suo riferim.)
  - si copia l'oggetto nell'albero
- entrambi gli approcci hanno pros e cons.
  - si sceglie il secondo (è + difficile da impl.;)

12

Classe Node

```
class Node {
public:
    virtual ~Node(); // destructor
    virtual Node* duplicate() const = 0; // duplicator

    // comparison of 2 objects
    virtual int compare(Node const *other) const = 0;

    // function to do whatever is needed to the node
    virtual void process() = 0;

    // called when object was already in the tree
    virtual void already_stored();
};
```

13

Memorizzazione Dati

- Il fare la copia dell'oggetto da memorizzare nell'albero e' compito della classe stessa
- Quando si deriva un particolare tipo di nodo si deve definire la funzione, definita pure virtual, duplicate, che restituisca un puntatore all'oggetto copiato.

```
virtual Node *duplicate() const = 0;
```

14

Confronto elementi

- Il confronto tra due nodi deve essere implementata per ogni classe derivata, quindi si definisce un'altra funzione pure virtual:

```
virtual int compare(Node const *other) const = 0;
```

15

Processing Nodi

- Di solito su di un albero si fa un qualche genere di processing che viene fatto per ogni nodo. Si aggiunge un'altra funzione da implementare che effettui tali operazioni durante il traversal dell'albero

```
virtual void process() = 0;
```

16

Gestione *Doppioni*

- Vogliamo che nell'albero ogni oggetto sia rappresentato una sola volta, può essere utile fornire una funzione virtuale per trattare i casi in cui si incontrano doppioni:

```
virtual void already_stored();
```

- Nota: la classe non è pure virtual

17

Classe Node

```
class Node {
public:
    virtual ~Node(); // destructor
    virtual Node* duplicate() const = 0; // duplicator

    // comparison of 2 objects
    virtual int compare(Node const *other) const = 0;

    // function to do whatever is needed to the node
    virtual void process() = 0;

    // called when object was already in the tree
    virtual void already_stored();
};
```

18

La classe Tree

- la classe tree è responsabile della gestione degli oggetti derivati da Node.
- La struttura dell'albero e' mantenuta con i soliti puntatori a figlio sinistro destro e all'informazioni (Node):

```
....
     // data
        Tree *left,
             *right;
        Node *info;
...
```

19

Costruttore Tree

- Il costruttore è semplice:

```
• // default constructor: initializes to 0
  Tree::Tree() { left = right = 0; info = 0; }
```

20

Aggiungere un elemento

- Si possono presentare due situazioni:
  - il campo info del nodo corrente è uno 0-pointer, allora si aggiunge un duplicato dell'oggetto al nodo corrente
  - Se l'albero è parzialmente pieno si sceglie se metterlo nel sottoalbero sinistro o destro a seconda del valore restituito dalla funzione compare.

21

Aggiungere un elemento

```
void Tree::add(Node *what){
   if (! info) info = what->duplicate();
   else {
         int cmp = info->compare(what);
         if (cmp < 0){
             if (! left){
                 left = new Tree;
                 left->info = what->duplicate();
             }
             else left->add(what);
        }
        else {
              if (cmp > 0){
                    if (! right){
                        right = new Tree;
                        right->info = what->duplicate();
                    }
                    else right->add(what);
              }
              else info->already_stored();
        }
    }
}
```

22

Gestione della memoria

- Poiché la classe tree contiene al suo interno puntatori occorre fare attenzione a non causare problemi di allocazione:
- occorre implementare
  - distruttore
  - overloading assegnamento
  - un costruttore di copia

23

Distruttore

- il distruttore della classe dovrà usare una funzione ricorsiva che discenda tutto l'albero:

```
Tree::~Tree() {
    destroy();
}
void Tree::destroy() {
    delete info;
    if (left) delete left;
    if (right) delete right;
}
```

24

Copiare un oggetto

- Fare una copia dell'albero significa copiarne tutta la struttura (e quindi discenderlo ricorsivamente)
- L'operazione di default è invece la copia membro a membro

25

Copiare un oggetto

- Due operazioni possono causare la copia di un'istanza
  - Assegnamento
    - a=b
  - Inizializzazione
    - dichiarazione di un nuovo oggetto
    - arg. passati per valore alle funzioni
    - oggetti ritornati dalle funzioni per valore

26

Copiare un oggetto

- Definiremo in dettaglio più avanti la seguente funzione, membro della classe tree

```
void copy(Tree const &other);
```

che, dato un albero, lo copia nella istanza corrente.

27

Overloading assegnamento

- Per ovviare al primo problema si ricorre all'overloading dell'operatore "**=**"

```
Tree const &Tree::operator=(Tree const &other){
    if (this != &other) {
        destroy();
        copy(other);
    }
    return (*this);
}
```

28

Copy Constructor

- In C++, un costruttore con un parametro il cui tipo è un riferimento ad un'altra istanza della classe.
- Il copy constructor è usato quando un'istanza è creata a partire da un'altra istanza.
- Se una classe è dichiarata senza un copy constructor il compilatore ne genera uno automaticamente

29

Copy Constructor

```
class Tree {
...
// Constructor
    Tree();
// Copy Constructor
    Tree(Tree const &other);
...

Tree::Tree(Tree const &other) {
    copy(other);
}
```

30

Copiare un albero

```
// copy constructor:
//       initializes to contents of other object
Tree::Tree(Tree const &other) {
    copy(other);
}

// private member function for copying a tree
void Tree::copy(Tree const &other) {
  info = other.info ? other.info->duplicate() :0;
  left = other.left ? new Tree(*other.left) :0;
 right = other.right ? new Tree(*other.right) :0;
}
```

31

Attraversamento di un albero

- L'attraversamento di un albero viene effettuato chiamando per ogni nodo la funzione process

```
void Tree::preorder_walk() {
    if (info) info->process();
    if (left) left->preorder_walk();
    if (right) right->preorder_walk();
}
```

32

Attraversamento di un albero

```
void Tree::inorder_walk() {
    if (left)  left->inorder_walk();
    if (info) info->process();
    if (right) right->inorder_walk();
}
void Tree::postorder_walk() {
    if (left)  left->postorder_walk();
    if (right) right->postorder_walk();
    if (info) info->process();
}
```

33

The Tree Class

```
class Tree {
public: // destructor, constructors
    ~Tree();
    Tree();
    Tree(Tree const &other);
// assignment
    Tree const &operator=(Tree const &other);
    void add(Node *what);
// processing
    void preorder_walk();
    void inorder_walk();
    void postorder_walk();
private:
// primitives
    void copy(Tree const &other);
    void destroy();
// data
    Tree *left, *right;
    Node *info;
};
```

34

Utilizzo dell'esempio

- Problema: contare le occorrenze di ogni parola in un file testo:
  - le parole lette vengono aggiunte all'albero
  - ogni volta che si trova un duplicato si incrementa un contatore relativo alla parola

35

Classe StrNode

- La classe StrNode, derivata dalla classe Node, implementa le funzioni virtuali (pure e non) che erano dichiarate in Node:

36

Classe Node

```
class Node {
public:
    virtual ~Node(); // destructor
    virtual Node* duplicate() const = 0; // duplicator

    // comparison of 2 objects
    virtual int compare(Node const *other) const = 0;

    // function to do whatever is needed to the node
    virtual void process() = 0;

    // called when object was already in the tree
    virtual void already_stored();
};
```

37

Classe StrNode

```
class Strnode: public Node    {
public:
// destructor, constructors
  ~Strnode();
  Strnode();
  Strnode(Strnode const &other);
  Strnode(char const *s);
// assignment
  Strnode const &operator=(Strnode const &other);

// functions required by Node protocol
  Node* duplicate() const;
  int compare(Node const *other) const;
  void process();
  void already_stored();

private:
// data
   char *str;
   int times;
};
```

38

Classe StrNode

- Ogni oggetto della classe rappresenta una stringa:

```
...
private:
// data
   char *str;
   int times;
...
```

- Il contatore **`times`** viene incrementato ogni volta che si incontra un doppione:

```
void Strnode::already_stored() {
      times++;
}
```

39

StrNode: Gestione Memoria

- Tre costruttori:

```
Strnode::Strnode() { str = 0; times = 0;    }

Strnode::Strnode(Strnode const &other) {
  str = strdupnew(other.str);
  times = other.times;
}

Strnode::Strnode(char const *s) {
  str = strdupnew(s);
  times = 1;
}
```

40

Gestione Memoria

- Nota: la funzione **`strdup()`** (*C*) utilizza al suo interno la **`malloc()`** e quindi la memoria utilizzata per tenere le stringhe andrebbe liberata con **`free()`** si preferisce quindi ridefinirla nel modo seguente, utilizzando l'operatore **`new`** del *C++*:

```
char *strdupnew(char const *str)    {
  return (strcpy(new char [strlen(str)+1], str));
}
// distruttore
Strnode::~Strnode(){ delete str; }
```

41

StrNode Duplicati e Overloading =

```
Strnode const &Strnode::operator=(Strnode
  const &other) {
  if (this != &other) {
     delete str;
     str = strdupnew(other.str);
     times = other.times;
  }
    return (*this);
}

Node *Strnode::duplicate() const {
  return (new Strnode(*this));
}
```

42

StrNode: Compare e Process

```
int Strnode::compare(Node const *other) const {
  Strnode *otherp = (Strnode *) other;
    if (str && otherp->str)
            return (strcmp(str, otherp->str));
  if (! str && ! otherp->str)
      return (0);
  return ((int) otherp->str - (int) str );
}

void Strnode::process(){
  if (str) printf("%4d\t%s\n", times, str);
}
```


Contare le parole

- ogni parola letta dal file viene aggiunta all'albero,
- la funzione `already_stored()` è chiamata automaticamente dalla classe Tree ogni volta che si aggiunge una parola già presente nell'albero
- alla fine del file si attraversa l'albero, stampando in ordine alfabetico le parole e il numero di occorrenze

Contare le parole

```
void countfile(FILE *inf, char const *name) {
  Tree  tree;
  char  buf [255];
  while (1)  {
      fscanf(inf, " %s", buf);
      if (feof(inf)) break;
      Strnode *word = new Strnode(buf);
      tree.add(word);
      delete word;
       }
  tree.inorder_walk();
}
```


Materiale di riferimento

- C++ Annotations
  - http://omega.di.unipi.it/IUM/IUP.html
  - ftp://ftp.icce.rug.nl/pub/http
- Mailing List
  - ium@di.unipi.it
  - mail a
    - listserv@di.unipi.it
  - contenuto
    - subscribe ium indirizzo