C++

Alcuni cenni sulla progettazione
di classi e librerie
Template

Paolo Cignoni
cignoni@iei.pi.cnr.it
9/3/98

Breve sunto: la classe tree

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

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();
};
```


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;
};
```


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;
};
```


Esercizio

- Aggiungere alla classe Tree un metodo che permetta di togliere oggetti da un albero.

Togliere un elemento da un albero

- Due fasi:
  - Trovare il sottoalbero che ha come come radice l'elemento cercato
  - Modificare il sottoalbero trovato togliendo la radice
- La funzione restituisce true se trova e cancella l'elemento

```
bool Tree::del(Node *what);
```

7

Trovare un elemento

```
bool Tree::del(Node *what){
  if (!info) return false;
  int cmp = info->compare(what);
  if (cmp < 0){
      if (!left) return false;
      else return left->del(what);
  }
  else {
      if (cmp > 0){
            if (! right) return false;
            else return right->del(what);
      }
      else {
      // cmp == 0 Abbiamo trovato l'elemento
      // Si deve modificare l'albero
      // eliminando la radice
      }
  }
}
```

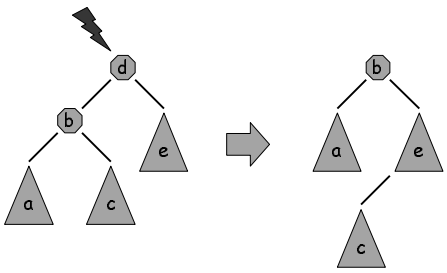
8

Togliere la radice di un albero

- In generale per essere occorrerebbe anche cercare di mantenere bilanciato l'albero (anche nella add()!)
- Qui ci limitiamo a mantenere la consistenza della struttura facendo *galleggiare* il sottoalbero sinistro

9

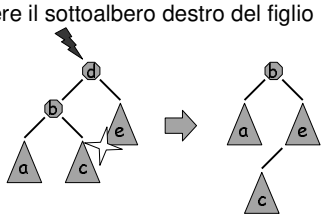
Eliminare la radice dall'albero



10

Eliminare la radice caso generale

- Occorre una funzione che dato un albero restituisca il sottoalbero piu' a sinistra che non abbia figli sinistri, cioè quello a cui appendere il sottoalbero destro del figlio sinistro



11

Trovare il primo elemento

- Dato l'ordinamento inerente dell'albero questo significa trovare l'elemento minimo:

```
Tree *Tree::findfirst(){
  if(left) return left->findfirst();
  else return this;
}
```

- Nota: vale sempre

```
findfirst()->left = 0;
```

12

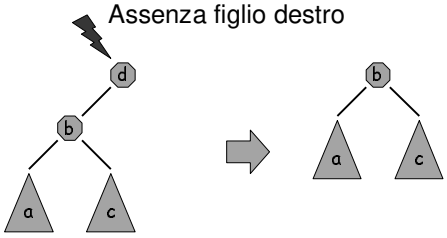
Eliminare la radice

- Casi particolari:
  - non esiste il figlio destro o sinistro
  - non esistono entrambi i figli

13

Eliminare la radice dall'albero

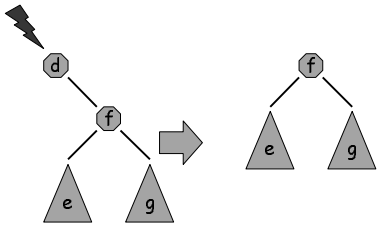
Assenza figlio destro



14

Eliminare la radice dall'albero

Assenza figlio sinistro



15

Eliminare la radice

- Si aggiunge un membro privato **`pop()`** alla classe che tolga la radice all'albero corrente e restituisca il nodo che era contenuto in essa:

```
Node *Tree::pop();
```

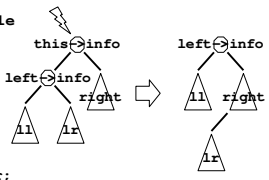
- la funzione **`del()`** quindi diventa:

```
...
     else {// cmp == 0 trovato l'elemento
          delete pop();
          return true;
     }
```

16

Eliminare la radice

```
Node *Tree::pop(){
Node *tmp=info;
  if (left){ // Caso generale
       Tree *ll=left->left;
       Tree *lr=left->right;
       Node *li=left->info;

       delete left;

       info = li;
       left = ll;
       if (!right) right = lr;
       else (right->findfirst())->left = lr;
  }
...
```



ERRORE!!!

17

Eliminare la radice

```
...
     delete left;
...
```

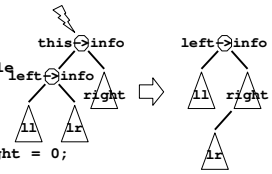
- L'operatore delete invoca il distruttore della classe prima di deallocarne la memoria (se il puntatore non è nullo), in questo caso invoca il distruttore dell'albero che distrugge tutto l'albero:

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

18

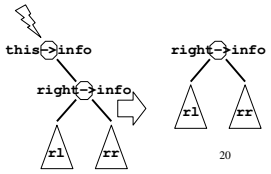
Eliminare la radice

```
Node *Tree::pop(){
Node *tmp=info;
  if (left){ // Caso generale
      Tree *ll=left->left;
      Tree *lr=left->right;
      Node *li=left->info;

      left->left = left->right = 0;
      left->info = 0;
      delete left;

      info = li;
      left = ll;
      if (!right) right = lr;
      else (right->findfirst())->left = lr;
  }
...
```



Eliminare la radice
altri casi

```
Node *Tree::pop(){
Node *tmp=info;
  if (left){ // Caso generale
  ...
  } else if (right) {// solo figlio destro
      Tree *rl = right->left;
      Tree *rr = right->right;
      Node *ri = right->info;

      right->left = right->right = 0;
      right->info = 0;
      delete right;
      info = ri;
      right = rr;
      left  = rl;
  }
  else info = 0;
  return tmp;
}
```



Alcune note sulla findfirst

- La funzione **`findfirst`** non modifica nessun elemento dell'albero, quindi potrebbe essere dichiarata come una funzione membro costante:

```
class Tree {
  ...
  Tree *findfirst() const;
  ...

Tree *Tree::findfirst() const{
  if(left) return left->findfirst();
  else return this;
}
```

Ma questo genera un errore in compilazione; perché?

```
`return' : no conversion from
   'const class Tree *const ' to 'class Tree *'
```


Const...

- Quando modifica una dichiarazione di dati, const specifica che l'oggetto o la variabile non è modificabile
- Quando segue la lista dei parametri di una funzione membro, const specifica che la funzione non modifica l'oggetto per cui è invocata.

```
const int i = 5;

i = 10; // Error
i++;    // Error
```


Const e puntatori

```
char *const aptr = mybuf;  // Constant pointer
// non posso cambiare il valore (indirizzo)
// del puntatore ma posso cambiare l'oggetto puntato
*aptr = 'a';       // Legal
aptr = yourbuf;    // Error
```

- un puntatore ad una variabile const può essere assegnato solo ad un puntatore che è dichiarato const

```
const char *bptr = mybuf;// Pointer to constant data
// non posso cambiare il contenuto della locazione
// puntata.
*bptr = 'a';       // Error
bptr = yourbuf;    // Legal
```

- può essere usato per impedire ad una funzione di cambiare un parametro passato per puntatore

Tipo di this

- In una funzione membro (non statica) la keyword **`this`** denota un puntatore all'oggetto per cui la funzione è chiamata
- Il tipo di **`this`** in una classe X è normalmente:

```
X *const
```

puntatore costante: posso cambiare il contenuto dell'oggetto puntato, ma non il valore (indirizzo) del puntatore.

Tipo di this

- Se invece la funzione membro è dichiarata costante il tipo di `this` è:

```
const X * const
```

puntatore costante ad un oggetto costante (non posso cambiare né il puntatore, né l'oggetto puntato)

25

findfirst

- La funzione `findfirst` non modifica nessun oggetto (e quindi al suo interno this è giustamente un puntatore ad un albero costante) ma noi vogliamo modificare l'oggetto puntato dal puntatore restituito dalla funzione:

```
Tree *Tree::findfirst() const{
  if(left) return left->findfirst();
  else return (Tree *)this;
}
```

26

Template

- I Template sono un meccanismo che permette di definire funzioni e classi basate su argomenti e oggetti dal tipo non specificato

```
template class <T> swap(T &a, T &b);

template class <T> class List {...};
```

- Queste funzioni e oggetti generici diventano codice completo una volta che le loro definizioni sono usate con oggetti reali

27

Esempio di template di funzione

Scambio tra due oggetti generici:

```
template <class T> void swap(T &a, T &b){
  T tmp = a;
  a = b;
  b = tmp;
}

int main(){
  int a = 3, b = 16;
  double d = 3.14, e = 2.17;
  swap(a, b);
  swap(d, e);
  // swap (d, a); errore in compilazione!

  return (0);
}
```

28

Template Osservazioni

- Notare che la definizione di una funzione template è simile ad una macro, nel senso che la funzione template *non è ancora codice*, ma lo diventerà una volta che essa viene usata
- Il fatto che il compilatore generi codice concreto solo una volta che una funzione è usata ha come conseguenza che una template function non può essere mai raccolta in una libreria a run time
- Un template dovrebbe essere considerato come una sorta di dichiarazione e fornito in un file da includere.

29

Array

- Come primo esempio si descrive una classe array,

```
template<class T> class Array {...
```

- generica, fatta per contenere un vettore di oggetti di tipo T

30

Array

- Come dati interni la classe mantiene un vettore di oggetti di tipo T e il numero di oggetti in esso contenuti

```
template<class T> class Array {
...
          // data
print n;
   T *data;
...
};
```


Array

- L'interfaccia con cui si accede alla classe array deve essere la stessa di un normale vettore ([]) per accedere agli elementi di un vettore.
  - overloading []
- La gestione della memoria deve essere fatta in maniera trasparente
  - distruttori e costruttori

Overloading []

- La keyword **operator** definisce una funzione che specifica il comportamento di un simbolo-operatore quando applicato alle istanze di una classe.

```
template<class T> class Array {
public:
...
      T &operator[](int index);
...
}
```

- Permette quindi l'utilizzo di un oggetto della classe array come un normale vettore

```
Array<int>  intarr;
intarr[i] = 1234;
```


Overloading []

- Si aggiunge anche il controllo che l'indice passato sia nel giusto intervallo

```
template <class T>
  T &Array<T>::operator[](int index) {
  if (index < 0 || index >= n) {
      fprintf(stderr, "Array: index out of bounds"
            "must be between 0 and %d\n "
            "requested was: %d\n", n – 1, index);
      exit(1);        }

  return (data[index]);
}
```


Costruttori e Distruttori

- Il costruttore si deve occupare di allocare la memoria per il vettore, si introduce una funzione privata ausiliaria di inizializzazione

```
template<class T> class Array {
public:      // constructors, destructors and such
   virtual ~Array(void)  { delete [] data; }
   Array(int sz = 10)    { init(sz); }
   Array(Array<T> const &other);
...
// initializer
   void init(int sz);
}
```


Inizializzazione del vettore

```
template <class T> void Array<T>::init(int sz) {
   if (sz < 1) {
       fprintf(stderr, "Array: of size < 1\n");
       exit(1);
     }
   n = sz;
   data = new T[n];
}
```

- Nota: l'operatore new invoca automaticamente il relativo costruttore per ogni oggetto del vettore

Sintassi delete

- l'operatore delete può essere usato in due modi a seconda che si debba disallocare un oggetto od un vettore di oggetti:

```
// Allocate a user-defined object and an object
//  of type double on the free store using the
//  new operator.
    UDType *UDObject = new UDType;
    double *dObject = new double;
    ...
// Delete the two objects.
    delete UDObject;
    delete dObject;
    ...
// Allocate an array of 5 user-defined objects on
// the free store using the new operator.
    UDType (*UDArr)[7] = new UDType[5][7];
    ...
// Use the array syntax to delete the array
    delete [] UDArr;
```


Array

```
template<class T> class Array {
public:      // constructors, destructors and such
  virtual ~Array(void)  { delete [] data; }
  Array(int sz = 10)    { init(sz); }
  Array(Array<T> const &other);
  Array<T> const &operator=(Array<T> const &other);
           // interface
  int size() const;
  T &operator[](int index);
private:
           // data
  int n;
  T *data;
           // initializer
  void init(int sz);
};
```


Costruttore di Copia

```
template <class T>
  Array<T>::Array(Array<T> const &other) {
  n = other.n;
  data = new T[n];
  for (register int i = 0; i < n; i++)
           data[i] = other.data[i];
}
```

- Il copy constructor è usato quando un'istanza è creata a partire da un'altra istanza

Overloading Assegnamento

```
template <class T>
  Array<T> const &Array<T>::operator=
  (Array<T> const &other) {
  if (this != &other) {
      delete []data;
      n = other.n;
      data = new T[n];
      for (register int i = 0; i < n; i++)
            data[i] = other.data[i];
      }
  return (*this);
}
```


Uso della classe Array<T>

```
int main(){
    Array<int>  intarr;
    for (int i = 0; i < intarr.size(); i++)
        intarr[i] = i << 2;

    Array<double> doublearr;
        for (i = 0; i < doublearr.size(); i++)
        doublearr[i] = PI * (i + 1);

    for (i = 0; i < intarr.size(); i++)
        printf("intarr[%d]   : %d\n"
                "doublearr[%d]: %g\n",
                i, intarr[i],
                i, doublearr[i]);
    return (0);
}
```


Esercizi

- Gestire il resize di un vettore
- Fornire un metodo per aggiungere elementi ad un vettore che non faccia il resize del vettore tutte le volte
- Cancellazione e inserzione di un elemento da un vettore

- Utilizzare questa struttura per definire una classe VTree con la stessa interfaccia della classe Tree ma che memorizzi l'albero binario in un vettore (radice in v[1], nodo in v[i] ha i figli in v[i*2] e v[i*2+1])