Linguaggi di Programmazione

Ivano Salvo

OOP5: Iteratori e uso di templates
FP6: perle di lazyness

Corso di Laurea in **Matematica**

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 7, 10 novembre 2020

Lezione 7a

Definizione e uso di Iteratori

Iteratori

Spesso è utile **percorrere tutti gli elementi** contenuti in una struttura dati (pensate alle belle computazioni col map in Haskell!). Questo non è possibile accedendo alla struttura dati, in quanto **l'utilizzatore non ha accesso allo stato nascosto**!

Il progettista della struttura dati offre dei metodi, detti **iteratori** per snocciolare tutti gli elementi della struttura dati. L'idea è quella di permettere una **forma astratta di iterazione** simile a quella che si trova nei cari vecchi libri di algoritmi…

forall $x \in S$ **do** …

Un **iteratore** in realtà **è un oggetto**, costruitto da una classe annidata dentro la struttura dati (**inner class**).

Essendo una classe annidata, ha accesso alla struttura dati e quindi può accedere allo stato, ad esempio usando un puntatore per scorrere gli elementi della struttura dati.

Definizione di Iteratori (1)

Definiamo un iteratore per gli stack come **classe annidata** dentro la classe `Stack` (**inner class**).

Per un motivo a me misterioso, in C++ occorre definire la classe `Stack:iterator` **amica** (**friend**) della classe `Stack`: questo **rende accessibile** lo stato di `Stack` (friend è un modo per **bypassare le regole di visibilità**).

Ci sono poi due metodi che creano iteratori, il primo posizionandosi all'inizio, l'altro alla fine.

```
template<class T>
class Stack{
   ...
   class iterator;
   friend iterator;
   ...
   iterator begin();
   iterator end();
   ...
} /* file Stack.h */
```

Definizione di Iteratori (2)

Sempre dentro la classe `Stack` definiamo **l'interfaccia** della classe `iterator` con le sue **variabili di istanza**.

Tradizionalmente si ridefiniscono gli operatori **++** (andare avanti), **--** (tornare indietro), ***** (elemento corrente – analogia coi pointers) e **==**.

```
template<class T>
class Stack{
   ...
class iterator{
      Stack<T> * myStack; /* pointer allo stack */
      int i;
   public:
      iterator(Stack<T> *);
      void operator++();
      void operator--();
      T operator*();
      bool operator==(iterator); };
   ...
} /* file Stack.h */
```

Definizione di Iteratori (3)

Cominciamo dai metodi `begin()` e `end()` della classe `Stack`: questi **creano un nuovo iteratore** chiamando i costruttori.

Osservate che passiamo come parametro un riferimento all'oggetto di tipo `Stack` ricevente. Inoltre passiamo l'indice dell'inizio dello Stack (`0`) e di fine (`t`).

tipo di ritorno

```
/* file Stack.cpp */

/* metodi per generare iteratori */
template<class T>
typename Stack<T>::iterator Stack<T>::begin(){
   return Stack<T>::iterator(this,0);
}

template<class T>
typename Stack<T>::iterator Stack<T>::end(){
   return Stack<T>::iterator(this, t);
}
```

costruttore di iterator

Definizione di Iteratori (4)

Vediamo il codice dei **costruttori** dell'iteratore invocati da `begin()` ed `end()`: prendono un riferimento allo stack a cui appartengono e sistemano il loro puntatore `i` all'interno della struttura dati.

```
/* file Stack.cpp */

/* costruttori di iteratori */
template<classT>
Stack<T>::iterator::iterator(Stack<T> * s){
   myStack=s;
   i=0;
}
template<class T>
Stack<T>::iterator::iterator(Stack<T>* s, int n){
   myStack=s;
   i=n;
}
```

nome del costruttore

salva un riferimento allo stack

Definizione di Iteratori (5)

Operatori ++, --, * e ==.

```
/* file Stack.cpp */
/* avanzare/indietreggiare sullo Stack */
template<class T>
void Stack<T>::iterator::operator++(){++i;}

template<class T>
void Stack<T>::iterator::operator--(){--i;}

/* torna l'elemento puntato dall'iteratore*/
template<class T>
T Stack<T>::iterator::operator*(){
   return myStack->s[i];
}
template<class T>
bool Stack<T>::iterator::operator==
                    (Stack<T>::iterator it){
   return i==it.i;
}
```

Uso di Iteratori

Vediamo infine come usando gli iteratori si possa scrivere un ciclo `for` che scorre tutti gli elementi dello Stack, in analogia al costrutto **forall** $x \in S$ **do** …

Certo, tutto è un po' **prolisso** e **macchinoso**…

È necessario conoscere questi meccanismi perché tutte le classi di strutture dati (nella **Standard Template Library**) sono definite in questo modo.

** è l'operatore di* ***iterator****, poi però ottengo un* ***Casella******

```
int main(){
   ...
   Stack<Casella*> s();
   ...
   for (Stack<Casella*>::iterator it=s.begin();
        it!=s.end();
        it++) *it->print();
   /* notare che *it è un Casella* ! */
}
```

Lezione 7b:

Uso di templates nell'applicazione Scacchi.

La Standard Template Library (STL)

Standard Template Library

Come tradizione di tutti i Linguaggi Orientati agli Oggetti (a partire dalle **Collection Classes** di **SmallTalk**) anche C++ ha una ricca famiglia di librerie di strutture dati: la **Standard Template Library** (**STL**).

Come dice il nome, questa libreria è costituita da strutture dati (chiamate anche **containers**) definite come template, quindi **generiche** rispetto ai dati che andranno a contenere.

Tipicamente offrono:

- **Costruttori**
- **Modificatori** (inserimento/eliminazioni)
- **Iteratori**
- Metodi per **copiare**/**trasferire** il contenuto in un'altra struttura dati
- **Operazioni** (o **algoritmi**) che ordinano, rovesciano, filtrano etc. gli elementi contenuti.

Elenco STL

array: normali vettori con qualche metodo/alias supplettivo

deque: sorta di "vettore dinamico" con inserimento in testa/coda, interno

forward_list: liste concatenate (possono essere "iterate" solo in avanti

list: liste **doppiamente** concatenate (possono essere "iterate" **sia in avanti che indietro**)

map: array **associativi**, di solito implementati con alberi di ricerca (richiedono un ordine sulle chiavi)

queue: code con inserzioni/rimozioni con disciplina FIFO;

set: insiemi (non replicazioni di elementi)

stack: pile con inserzioni/rimozioni con disciplina LIFO;

unordered_map: array **associativi**, di solito implementati con tabelle hash (non richiedono un ordine sulle chiavi)

unordered_set:

vector: vettori "estensibili". Vengono riscalati quando necessario

Esempio: Classe Partita in Scacchi

Come già detto, la classe `Partita` tiene una serie di informazioni supplettive per memorizzare lo stato del gioco oltre alla scacchiera e al turno.

```
/* informazione ridondante */
   Colore avv; /* colore di chi non muove */
   map<Colore, Pezzo *> re; /* rifer. ai Re */
/* due insiemi dei pezzi sulla scacchiera */
   map<Colore, set<Pezzo*> > pezzi;
/* possibilità di arroccare */
   map<Colore, bool*> arrocco;
/* numero di mosse senza prese e mosse di pedone */
   int mosseNeutre;
/* sequenza di mosse giocate */
   int nMosse;
   vector<Mossa> mosse;
/* casella in cui si potrebbe avere una presa
 * en-passant*/
   Casella ep;
...
```

Esempio: verifica scacco al Re

Un metodo ottimizzato per verificare se un Re è sotto scacco potrebbe, partendo dalla posizione del Re **cercare lungo diagonali, righe, colonne** e mosse di cavallo possibili **pezzi avversari che lo minacciano**.

Questo modo di procedere è senz'altro efficiente, ma ad esempio **poco flessibile** rispetto a future estensioni (esempio: introduzione di nuovi pezzi o regole).

Nella nostra applicazione **questa efficienza**, tutto sommato **non è necessaria.** Possiamo scrivere del codice molto semplice che semplicemente **analizza tutte le mosse di tutti i pezzi avversari** del re in questione!

Questo modo di procedere è particolarmente utile in fase di prototipazione del software: **si produce rapidamente un prototipo funzionamente** e poi, eventualmente, si vanno a **raffinare le procedure critiche** da un punto di vista dell'efficienza.

Verifica scacco al Re: codice

Si prende la posizione del Re r.

Si itera sull'insieme dei pezzi avversari (mantenuto nell'array associativo `pezzi`, che a ogni colore restituisce l'insieme dei pezzi di tale colore)

Si controlla **se esiste un pezzo avversario che può muovere** verso la posizione del re.

Se non esiste si torna `false`.

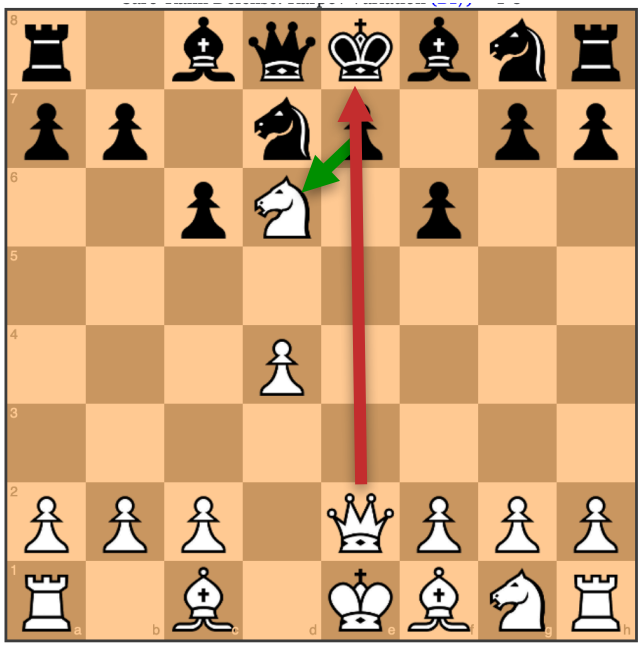
sintassi stile array per le map

```
bool Partita::reSottoScacco(Colore t){
   Casella r = re[t]->posizione();
   Colore avv = t.cambiaColore();
   for (set<Pezzo*>::iterator it = pezzi[avv].begin();
      it!=pezzi[a].end(); ++it){
         if ((*it)->puoiMuovere(r,s)==LEGALE)
            return true;
   }
   return false;
}
```

Verifica dello scacco matto: problemi

La verifica dello scacco matto **può essere molto complicata** se si cerca di implementarla in modo "**chirurgico**": ci sono molte mosse che possono salvare il re (mosse di re, presa del pezzo offendente, coperture… ma ci sono possibili side-effects!)

È tuttavia possibile scrivere con relativa facilità un metodo **brute force** che **prova tutte le mosse possibili** del colore sotto scacco e verifica se almeno una di esse rimuove la situazione di scacco.



e7xd6 non salva dallo scacco matto

perché in tal caso il Re rimane esposto allo scacco di regina!

E ci sono situazioni anche più complesse.

Verifica dello scacco matto: codice

Usando la precedente funzione `reSottoScacco`, e il trucco di eseguire/ritirare **ogni mossa possibile**…

```
bool Partita::scaccoMatto(Colore t){
   Casella r = re[t]->posizione();
   Colore avv = t.cambiaColore();
   /* iteriamo su tutti i pezzi amici */
   for (auto it = pezzi[t].begin();
             it!=pezzi[a].end(); ++it){
   /* iteriamo su tutte le mosse di ciascun pezzo */
      set<Casella *> pm = (*it)->allMoves(s);
      for (auto cit = pm.begin();
             cit!=pm.end(); ++cit){
         Mossa m = eseguiMossa(*it->posizione, **cit);
         bool rss = reSottoScacco(t);
         ritiraMossa(m);
         if (!rss) return false;
      }
   }
   return true;
}
```

è possibile evitare di scrivere tipi e lasciarli inferire al compilatore

metodo che genera tutte le mosse di un pezzo

metodo allMoves in Pezzo

Anche questo si **può implementare in `Pezzo`** e si specializza **grazie alla specializzazione di `puoiMuovere`**.

Itera su tutte le caselle della scacchiera.

Eventualmente, si può riscrivere il codice di allMoves su ogni singolo pezzo, evitando di esaminare tutte le caselle.

```
set<Casella *> Pezzo::allMoves(Scacchiera s){
   /* si crea l'insieme vuoto */
   set<Casella *> sm;
   for(int i=0; i<RIG; i++)
      for(int j=0; j<COL; j++){
         Casella *a = new Casella(i,j);
         if (puoiMuovere(*a,s)==LEGALE)
            /* si inserisce la casella nel Set */
                  sm.insert(a);
      }
   return sm;
}
```

sarebbe stato meglio definire un iteratore in scacchiera che snocciolava tutte le sue caselle

metodo allMoves in Cavallo

Ecco una possibile ridefinizione di questo metodo in Cavallo.

Ovviamente questo codice analizza **solo le 8 possibili mosse del cavallo** e ha da ritenersi molto più efficiente del precedente.

```
set<Casella *> Cavallo::allMoves(Scacchiera s){
   set<Casella *> sm;
   for(int i=0; i<8; i++)
      Casella *a = new Casella(pos.x+CavDir[i],
               pos.y+CavDirY[i]);
      if puoiMuovere(pos, a) sm.insert(a);
   }
   return sm;
}
```

Lezione 7c:

Perle di Laziness

Hamming Numbers (1)

Problema: generare tutti i composti di 2, 3 e 5 in ordine crescente.

Formalmente: $H = \{ 2^i \cdot 3^j \cdot 5^k \mid i, j, k \geq 0 \}$

Cominciamo con una soluzione che filtra tutti i numeri che non sono composti di 2, 3 e 5 (in verità **generalizziamo a un qualsiasi insieme di generatori**).

Questa soluzione è estremamente inefficiente, perché gli Hamming numbers hanno **densità 0**! ($\lim_{n\to\infty} H_n/n = 0$, dove H_n è il numero di elementi di una successione minori di n).

```
 -- funzione che controlla se un numero n è composto
 -- solo dei numeri in una lista gs
compositeOf   gs   1   = true
compositeOf   []   n   = false
compositeOf gs@(g:tgs) n
   | n `mod` g == 0   = compositeOf gs (n `div` g)
   | otherwise        = compositeOf tgs n

hamming gs = filter (compositOf gs) [1..]
```

Hamming Numbers (2)

Pensiamo a una definizione induttiva dell'insieme degli Hamming H. Vediamo il caso di `soli' 2 numeri p ed q:

$$H = \{1\} \cup \{p \cdot n \mid n \in H\} \cup \{q \cdot n \mid n \in H\}$$

che è una **proprietà di chiusura** e praticamente si può scrivere in Haskell, ricordando che lavoriamo con liste ordinate, e vogliamo **evitare duplicati** e **produrre una lista** ordinata.

Osservate che si consuma al più un elemento per ogni lista per produrre un nuovo elemento della lista risultato!

```
 -- definiamo prima union (merge senza duplicati)
union xs@(x:txs) ys@(y:tys)
      | x < y       = x:union txs ys
      | x > y       = y:union xs tys
      | otherwise   = x:union txs tys

hamming p q = hs where
   hs = 1:union (map (p*) hs)(map (q*) hs)
```

Hamming Numbers (3)

Problema: Proviamo a generare tutti i composti di un insieme G di generatori. Come prima avremo:

$$H = \{1\} \cup \bigcup_{g \in G} \{g \cdot n \mid n \in H\}$$

che non è diverso da prima, ricordando che possiamo `foldare' la funzione union su una lista di liste e mappare l'operazione (g*) dentro la stessa lista di liste…

Nota: `foldr1` evita di dover dare un valore iniziale.

Attenzione invece che `foldl` non termina mai su stream!

```
union xs@(x:txs) ys@(y:tys)
      | x < y       = x:union txs ys
      | x > y       = y:union xs tys
      | otherwise   = x:union txs tys

hamming gs = hs where
   hs = 1:foldr union [] (map (\x->map (x*) hs) gs)
-- hs = 1:foldr1 union (map (\x->map (x*) hs) gs)
```

Lezione 7

That's all Folks…

Grazie per l'attenzione…

…Domande?