Algoritmi e Strutture Dati

Nicola Di Mauro

Dipartimento di Informatica
Università degli Studi di Bari “Aldo Moro”

Code in C++

Esempio di Coda

```
F   F
I   F I
R   F I R
S   F I R S
*   FI R S
T   I R S T
*   IR S T
I   R S T I
N   R S T I N
*   RS T I N
*   ST I N
*   TI N
F   I N F
I   I N F I
*   IN F I
R   N F I R
S   N F I R S
*   NF I R S
*   FI R S
*   IR S
T   R S T
*   RS T
O   S T O
U   S T O U
T   S T O U T
*   ST O U T
*   TO U T
*   OU T
*   UT
*   T
```

Questa sequenza mostra il risultato di una serie di operazioni indicate nella colonna di sinistra (dall'alto verso il basso), dove una lettere indica un inCoda e un asterisco indica un fuoriCoda. Ogni riga mostra l'operazione, la lettera restituita dalla fuoriCoda e il contenuto della coda ordinato dall'elemento inserito meno di recente a quello inserito più di recente, da sinistra a destra.

Coda: rappresentazione con vettore
Interfaccia

```
#ifndef _CODAVT_
#define _CODAVT_

#include <iostream>
#include <assert.h>

template < class tipoelem >
class Coda {
public:
   Coda(int);
   ~Coda();
   void creaCoda();
   bool codaVuota();
   void fuoriCoda();
   void inCoda(tipoelem);
private:
   tipoelem *elementi;
   int testa, lung, maxlung;
};
```

Coda: rappresentazione con vettore
Costruttore e distruttore

```
template <class tipoelem>
Coda<tipoelem>::Coda(int n){
   maxlung = n;
   creaCoda();
};

template <class tipoelem>
Coda<tipoelem>::~Coda(){
   delete[] elementi;
}
```

Coda: rappresentazione con vettore
creaCoda

```
template <class tipoelem>
void Coda<tipoelem>::creaCoda(){
    elementi = new tipoelem[maxlung];
    testa = 0;
    lung = 0;
}
```

Coda: rappresentazione con vettore
CodaVuota

```
template <class tipoelem>
bool Coda<tipoelem>::codaVuota(){
    return (lung == 0);
}
```

Coda: rappresentazione con vettore
leggiCoda

```
template <class tipoelem>
tipoelem Coda<tipoelem>::leggiCoda(){
    assert (!codaVuota());
    return (elementi[testa]);
}
```

Coda: rappresentazione con vettore
fuoriCoda

```
template <class tipoelem>
void Coda<tipoelem>::fuoriCoda(){
    assert(!codaVuota());
    testa = (testa + 1) % maxlung;
    lung--;
}
```

Coda: rappresentazione con vettore
inCoda

```
template <class tipoelem>
void Coda<tipoelem>::inCoda(tipoelem a){
    assert(lung != maxlung);
    elementi[(testa+lung) % maxlung] = a;
    lung++;
}
```

Coda: rappresentazione con vettore

```
#ifndef _CODAVT_
#define _CODAVT_

#include <iostream>
#include <assert.h>

template < class tipoelem >
class Coda {
public:
   Coda(int);
   ~Coda();
   void creaCoda();
   bool codaVuota();
   void fuoriCoda();
   void inCoda(tipoelem);
private:
   tipoelem *elementi;
   int testa, lung, maxlung;
};

template <class tipoelem>
Coda<tipoelem>::Coda(int n){
   maxlung = n;
   creaCoda();
};

template <class tipoelem>
Coda<tipoelem>::~Coda(){
   delete[] elementi;
}
```

```
template <class tipoelem>
void Coda<tipoelem>::creaCoda(){
    elementi = new tipoelem[maxlung];
    testa = 0;
    lung = 0;
}

template <class tipoelem>
bool Coda<tipoelem>::codaVuota(){
    return (lung == 0);
}

template <class tipoelem>
tipoelem Coda<tipoelem>::leggiCoda(){
    assert (!codaVuota());
    return (elementi[testa]);
}

template <class tipoelem>
void Coda<tipoelem>::fuoriCoda(){
    assert(!codaVuota());
    testa = (testa + 1) % maxlung;
    lung--;
}

template <class tipoelem>
void Coda<tipoelem>::inCoda(tipoelem a){
    assert(lung != maxlung);
    elementi[(testa+lung) % maxlung] = a;
    lung++;
}
#endif  /* _CODAVT_H_ */
```

codavt.h

Esercizio 1

Coda senza duplicati

- politica “ignora il nuovo elemento”
  - un elemento non va inserito nella coda se già presente
- politica “dimentica il vecchio elemento”
  - si aggiunge sempre il nuovo elemento ma si rimuove un eventuale duplicato

```
/* ignora il nuovo /
template <class tipoelem>
void Coda<tipoelem>::inCoda(tipoelem a){
    assert(lung != maxlung);
    if (!this.presente(a)) {
       elementi[(testa+lung) % maxlung] = a;
       lung++;
    }
}

/* dimentica il vecchio */
template <class tipoelem>
void Coda<tipoelem>::inCoda(tipoelem a){
    assert(lung != maxlung);
    if (this.presente(a)) {
       this.rimuovi(a);
       elementi[(testa+lung) % maxlung] = a;
       lung++;
    }
}
```

Da realizzare: presente(tipoelem) e rimuovi(tipoelem)

Esercizio 2

Simulare una situazione in cui si assegnano in modo casuale gli utenti in attesa di servizio a una di M possibili code. Quindi, scegliamo una coda a caso e, se non è vuota, eseguiamo il servizio richiesto dall'utente. Ogni volta stampiamo l'utente aggiunto, quello servito, e il contenuto delle cose.

75 in 74 out
0: 58 59 60 67 68 73
1:
2: 64 66 72
3: 75

76 in
0: 58 59 60 67 68 73
1:
2: 64 66 72
3: 75 76

77 in 58 out
0: 59 60 67 68 73
1: 77
2: 64 66 72
3: 75 76

```
#include “codavt.h”
static const int M = 4;
int main(int argc, char *argv[]){
   int N = atoi(argv[1]);
   Coda<int> code[M];
   for (int i=0; i<N; i++, cout << endl){
      int in = rand() % M, out = rand() % M;
      code[in].inCoda(i);
      cout << i << “ in ”;
      if (!code[out].codaVuota())
        cout << code[out].leggiCoda() << “ out “;
      cout << endl;
      for (int k=0; k<M; k++, cout << endl)
         stampaCoda(code[k]);
   }
}
```

Da realizzare: stampaCoda(Coda)

Le eccezioni

- Le eccezioni vengono usate per segnalare il verificarsi di errori
  - la valutazione dell'espressione a+b*c+b/c con c=0
- E' possibile scrivere programmi C++ che lanciano eccezioni al verificarsi di eventi eccezionali

```
int abc(int a, int b, int c)
{
    if (a <= 0 || b <= 0 || c <= 0)
        throw “All parameters should be > 0”;
    return a + b * c;
}
```

Lancia un'eccezione di tipo char*

Eccezioni /2

- Il C++ ha una gerarchia di classi per le eccezioni con radice la classe **exception**
  - divide by zero, illegal parameter value, ...
- Le eccezioni che potrebbero essere lanciate da un pezzo di codice possono essere gestite racchiudendo il codice in un blocco **try**
  - il blocco try è seguito da uno o più blocchi **catch**
    - ogni blocco catch cattura un tipo specifico di eccezione
      - catch (char* e) {}
        - cattura eccezioni di tipo char*
      - catch (bad_alloc e) {}
        - cattura eccezioni di tipo bad_alloc

Eccezioni /3

```
int main()
{
    try { cout << abc(2,0,4) << endl; }
    catch (char* e)
    {
        cout << “I parametri della funzione abc erano 2,0, e 4” << endl;
        cout << “E' stata lanciata un'eccezione” << endl;
        cout << e << endl;
        return 1;
    }
    retrun 0;
}
```

Definizione classe illegalParameterValue

```
class illegalParameterValue
{
   public:
      illegalParameterValue() :
          message(“Illegal parameter value”) {}
      illegalParameterValue(char* theMessage)
         {message = theMessage;}
      void outputMessage() {cout << message << endl;}
   private:
      char * message;
}
```

classe illegalParameterValue

```
int abc(int a, int b, int c)
{
    if (a <= 0 || b <= 0 || c <= 0)
        throw illegalParameterValue(All parameters should be > 0”);
    return a + b * c;
}
```

```
int main()
{
    try { cout << abc(2,0,4) << endl; }
    catch (illegalParameterValue e)
    {
        cout << “I parametri della funzione abc erano 2,0, e 4” << endl;
        cout << “Lanciata l'eccezione illegalParameterValue” << endl;
        e.outputMessage();
        return 1;
    }
    retrun 0;
}
```

Coda: raffinamento con eccezioni

```
template <class tipoelem>
tipoelem Coda<tipoelem>::leggiCoda(){
    if (lung == 0)
       throw queueEmpty();
    return (elementi[testa]);
}


// in questo metodo si potrebbe usare l'arrayDoubling
template <class tipoelem>
void Coda<tipoelem>::inCoda(tipoelem a){
    if (lung != maxlung)
       throw queueFull();
    elementi[(testa+lung) % maxlung] = a;
    lung++;
}
```