Inoltre, dobbiamo definire `operator<<` per il template di classe `QueueItem`.

```
template <class T>
ostream& operator<<(ostream& os,const QueueItem<T>& qi){
  os << qi.info; // amicizia con QueueItem!
                 // NB: richiede operator<< sul tipo T
  return os;
}
```

Dobbiamo quindi dichiarare la funzione esterna `operator<<` di `QueueItem` come amica associata della classe `QueueItem`.

```
template <class T>
class QueueItem {
  friend ostream& operator<< <T>(ostream&,const QueueItem<T>&);
  ...
};
```

Campi dati statici in template di classe

Un campo dati statico intero che funzioni da contatore globale degli oggetti **`QueueItem`** presenti nelle liste di tutti gli oggetti di una certa istanza di **`Queue`**.

```
template <class Tipo>
class Queue {
  private:
  static int contatore;
  ...
};
```

L'inizializzazione esterna alla classe ha la seguente forma:

```
template <class T>
int Queue<T>::contatore = 0;
```

Un campo dati statico è istanziato e quindi inizializzato dalla definizione del template **soltanto se viene effettivamente usato**. La mera definizione di un campo dati statico non provoca allocazione di memoria.


```
template<int I> // parametro valore
class C {
static int numero;
public:
  C() {++numero;}
  C(const C& x) {++numero;}
  static void stampa_numero();
};

// inizializzazione parametrica del campo statico
template<int I>
int C<I>::numero = I;

template<int I>
void C<I>::stampa_numero()
{ cout << "Valore statico: " << numero << endl; }

int main() {
  C<1> uno;
  C<2> due_a;
  C<2> due_b(due_a);
  C<1>::stampa_numero(); // stampa: 2
  C<2>::stampa_numero(); // stampa: 4
}
```


```
class A {
public:
  A(int x=0) {cout << x << "A() ";}
};

template<class T>
class C {
public:
  static A s;
};
template<class T>
A C<T>::s=A(); // stampa 0A()

int main() {
  C<double> c;
  C<int> d;
  C<int>::s = A(2);
}
// stampa: 0A() 2A()
// NOTA BENE: non stampa 0A() 0A() 2A() !!
```

Friend e classi annidate

C++ standard 2003 TR1 (2005)

standard $11.7.1

"A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed"

and the usual access rules specify that:

"A member of a class can also access all the names to which the class has access..."

Classi annidate in template di classe

Annidiamo nella parte privata del template di classe `Queue` la definizione del template di classe `QueueItem`.

```
template <class T>
class Queue {
private:
  // template di classe annidato associato
  class QueueItem {
  public:
    QueueItem(const T& val);
    T info;
    QueueItem* next;
  };
  ...
};
```

`QueueItem<T>` è un cosiddetto **tipo implicito**, in quanto non è un tipo completamente definito ma dipende implicitamente dai parametri di tipo di `Queue<T>`

Tipi e template impliciti in template di classe

```
template <class T>
class C {
public:
  class X {};          // template di classe annidata associato
                       // potrebbe usare il parametro di tipo T

  class D {            // template di classe annidata associato
  public:              // usa il parametro di tipo T
    T x;
  };

  template <class U>
  class E {            // template di classe annidata
  public:              // non associato
    T x;               // usa T del template contenitore
    U y;               // usa il suo parametro di tipo U
    void fun1() {return;}
  };

  template <class U>
  void fun2() {        // template di metodo di istanza
    T x; U y; return; // non associato
  }
};
```

Tipi e template impliciti in template di classe

```
template <class T>
  // C<T>::D è un uso di un tipo che
  // effettivamente dipende dal parametro T
void templateFun(typename C<T>::D d) {

  // C<T>::X è un uso di un tipo che
  // comunque dipende dal parametro T
  typename C<T>::X x;

  // C<T>::D è un uso di un tipo che
  // effettivamente dipende dal parametro T
  typename C<T>::D d2 = d;

  // (1) E<int> è un uso del template di classe
  //     annidata che dipende dal parametro T
  // (2) C<T>::E<int> è un uso di un tipo
  //     che dipende dal parametro T
  typename C<T>::template E<int> e;
  e.fun1();

  // c.fun2<int> è un uso del template di funzione
  // che dipende dal parametro T
  C<T> c;
  c.template fun2<int>();
}
```


Comune di
ANVERSA degli ABRUZZI
VIA
SOPPORTO

Forward declarations for templates (check your g++)

```
// dichiarazione incompleta (alias forward declaration)
template<class T> class Queue;

// dichiarazione del template di funzione operator<< <T>
template<class T> std::ostream& operator<<(std::ostream& os,const Queue<T>&);

// dichiarazione incompleta non permessa
// template<class T> class Queue<T>::QueueItem;

// dichiarazione non necessaria
// template<class T> ostream& operator<<(ostream& os,const typename Queue<T>::QueueItem&);

template<class T> class Queue {
  // NON agisce da dichiarazione di template di funzione
  friend std::ostream& operator<< <T> (std::ostream& os, const Queue<T>&);
private:
  class QueueItem {
    friend class Queue<T>;
    friend std::ostream& operator<< <T> (std::ostream& os, const Queue<T>&);
    // agisce da dichiarazione di template di funzione
    friend std::ostream& operator<< <T> (std::ostream& os, const typename Queue<T>::QueueItem&);
  private:
    T info;
    QueueItem* next;
  };
  QueueItem *primo, *ultimo;
public:
  // ...
};

// definizione del template di funzione operator<< <T>
template<class T> std::ostream& operator<< (std::ostream& os, const Queue<T>& q) {
  // ...
  return os;
}

// definizione del template di funzione operator<< <T>
template<class T> std::ostream& operator<< (std::ostream& os, const typename Queue<T>::QueueItem& qi) {
  // ...
  return os;
}
```


macrolibrarsi.it
Gli
ESERCIZI DI
RICARICA
di Paramhansa Yogananda
macrolibrarsi.it

Definire un template di classe `albero<T>` i cui oggetti rappresentano un **albero 3-ario** in cui i nodi memorizzano dei valori di tipo `T` ed hanno 3 figli (invece dei 2 figli di un usuale albero binario). Il template `albero<T>` deve soddisfare i seguenti vincoli:

1. Deve essere disponibile un costruttore di default che costruisce l'albero vuoto.
2. Gestione della memoria senza condivisione.
3. Metodo `void insert(const T&)`: in una chiamata `a.insert(t)`, inserisce nell'albero `a` una nuova radice che memorizza il valore `t`, ed avente come figli 3 copie di `a`.
4. Metodo `bool search(const T&)`: in una chiamata `a.search(t)` ritorna true se il valore `t` occorre nell'albero `a`, altrimenti ritorna false.
5. Overloading dell'operatore di uguaglianza.
6. Overloading dell'operatore di output.

```
/*
Definire un template di classe albero<T> i cui oggetti rappresentano
un albero 3-ario ove i nodi memorizzano dei valori di tipo T ed hanno
3 figli (invece dei 2 figli di un usuale albero binario). Il template
albero<T> deve soddisfare i seguenti vincoli:
1. Deve essere disponibile un costruttore di default che costruisce l'albero vuoto.
2. Gestione della memoria senza condivisione.
3. Metodo void insert(const T&): a.insert(t) inserisce nell'albero a una nuova radice che memorizza il
valore t ed avente come figli 3 copie di a
4. Metodo bool search(const T&): a.search(t) ritorna true se t occorre nell'albero a, altrimenti
ritorna false.
5. Overloading dell'operatore di uguaglianza.
6. Overloading dell'operatore di output.

*/

template <class T>
class albero {
private:
  // classe annidata associata
  class nodo {
  public:
    T info;
    nodo *sx, *cx, *dx;
    nodo(const T& x, nodo* s =0, nodo* c =0, nodo* d =0):
      info(x), sx(s), cx(c), dx(d) {}
  };

  nodo* root;

  // copia profonda ricorsiva
  static nodo* copia(nodo* r) {
    if(!r) return nullptr;
    // albero non vuoto
    return new nodo(r->info, copia(r->sx), copia(r->cx), copia(r->dx));
  }
  // distruzione profonda ricorsiva
  static void distruggi(nodo* r) {
    if(r) {
      distruggi(r->sx); distruggi(r->cx); distruggi(r->dx);
      delete r;
    }
  }

public:
  albero(): root(nullptr) {}
  albero(const albero& a): root(copia(a.root)) {}
  albero& operator=(const albero& a) {
    if(this != &a) {
      if(root) distruggi(root);
      root = copia(a.root);
    }
    return *this;
  }
  ~albero() {if(root) distruggi(root);}

  void insert(const T& x) {
    root = new nodo(x,copia(root), copia(root), root);
  }
};

int main() {
  albero<char> t1, t2, t3;
  t1.insert('b');
  t1.insert('a');
  t2.insert('a');
  t3 = t1;
  t3.insert('c');
}
```