

Template di classe



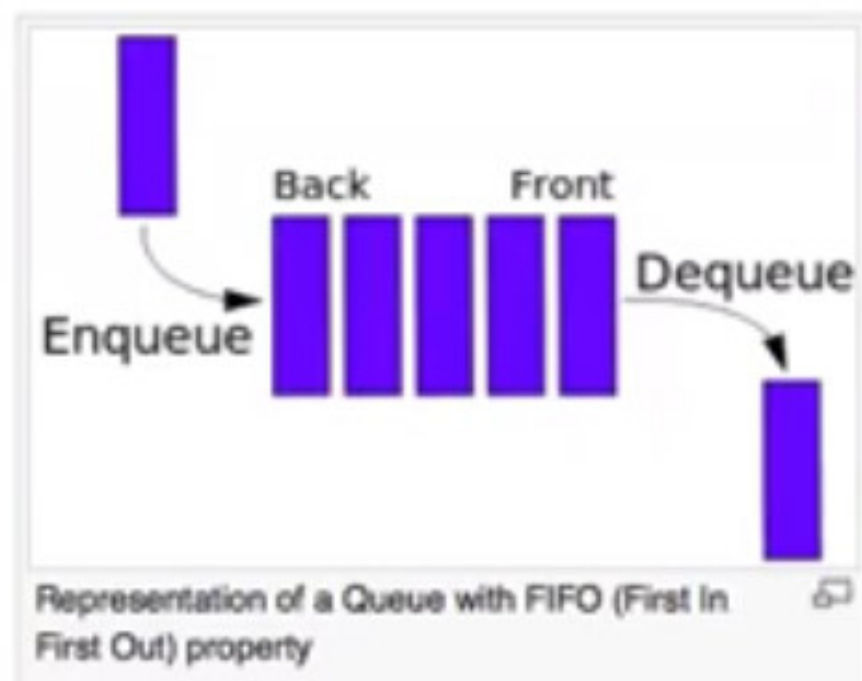
*Píovego
pre-covíd*

Queue (abstract data type)

From Wikipedia, the free encyclopedia

In [computer science](#), a **queue** (/ˈkjuː/ ***KEW***) is a particular kind of [abstract data type](#) or [collection](#) in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue*, and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a [First-In-First-Out \(FIFO\) data structure](#). In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a *peek* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a [linear data structure](#), or more abstractly a sequential collection.

Queues provide services in [computer science](#), [transport](#), and [operations research](#) where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a [buffer](#).



Se si vogliono usare sia code di interi che code di stringhe, si devono scrivere due definizioni distinte della classe e con due nomi diversi.

```
class QueueInt {  
public:  
    Queue();  
    ~Queue();  
    bool empty() const;  
    void add(const int&);  
    int remove();  
private:  
    ...  
};
```

```
class QueueString {  
public:  
    Queue();  
    ~Queue();  
    bool empty() const;  
    void add(const string&);  
    string remove();  
private:  
    ...  
};
```


Template di classe

Queue<T>

```
template <class T>
class Queue {
public:
    Queue();
    ~Queue();
    bool empty() const;
    void add(const T&);
    T remove();
private:
    ...
};
```

```
Queue<int> qi;
Queue<bolletta> qb;
Queue<string> qs;
```

Template di classe

- Parametri di tipo
- Parametri valore
- Parametri tipo/valore con **possibili valori di default**
- Solo **istanziazione esplicita**

```
template <class Tipo = int, int size = 1024>
class Buffer {
...
};
```

```
Buffer<> ib;           // Buffer<int,1024>

Buffer<string> sb;     // Buffer<string,1024>

Buffer<string,500> sbs; // Buffer<string,500>
```


Completiamo il template `Queue<T>`



```

template <class T>
class QueueItem {           // per ora classe esterna
public:                     // per ora tutto public
    QueueItem(const T&);
    T info;
    QueueItem* next;
};

template <class T>
class Queue {
public:
    Queue();                // Queue e non Queue<T>
    ~Queue();               // distruzione profonda
    bool empty() const;
    void add(const T&);
    T remove();
private:
    QueueItem<T>* primo;    // QueueItem<T>
    QueueItem<T>* ultimo;  // e non QueueItem
};

```

Nella dichiarazione o definizione di un template (di classe o di funzione) possono comparire sia nomi di istanze di template di classe sia nomi di template di classe.

```
template <class T>
int fun(Queue<T>& qT, Queue<string> qs);
// Queue<T> template di classe associato
// Queue<string> istanza di template di classe
```


Attenzione



Nota Bene: il compilatore genera una istanza di un template (di classe o funzione) **solo quando è necessario**.

Ad esempio, il compilatore non genera l'istanza `Queue<int>` quando incontra le due seguenti occorrenze del nome dell'istanza:

```
// basta una dichiarazione incompleta di template
template <class T> class Queue;

void Stampa(const Queue<int>& q) {
    Queue<int>* pqi = const_cast< Queue<int>* >(&q);
    ...
}
```

Invece il compilatore è costretto a generare l'istanza del template di classe con

```
template <class T> class Queue {
    // definizione della classe necessaria
};

void Stampa(Queue<int> q) {
    Queue<int> qi; // si genera l'istanza Queue<int>
                  // per la costruzione di default
}
```


Il compilatore è pure costretto a generare l'istanza `Queue<int>` con

```
template <class T> class Queue {  
    // definizione  
};  
  
void Stampa(const Queue<int>& q) {  
    Queue<int>* pqi = const_cast< Queue<int>* >(&q);  
    pqi++;  
    ...  
}
```

perché l'istanza serve per calcolare la quantità `sizeof(Queue<int>)` di cui occorre incrementare il puntatore per eseguire `pqi++`.

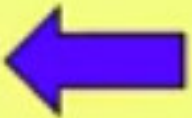
Metodi di un template di classe

In un template di classe è possibile definire metodi inline

```
template <class T>
class Queue {
    ...
public:
    Queue() : primo(0), ultimo(0) {}
    ...
};
```

La **definizione esterna** di un metodo in un template di classe richiede la seguente sintassi:

```
template <class T>
class Queue {
    ...
public:
    Queue() ;
    ...
};

// definizione esterna 
template <class T>
Queue<T>::Queue() : primo(0), ultimo(0) {}
```

Un metodo di un template di classe è un template di funzione. Esso non viene istanziato quando viene istanziata la classe **ma se e soltanto se** il programma usa effettivamente quel metodo.

Completiamo la definizione del template `Queue<T>`.

```
// file Queue.h
#ifndef QUEUE_H
#define QUEUE_H

template <class T>
class QueueItem {
public:
    // per gli scopi di Queue basta questo costruttore
    QueueItem(const T& val): info(val), next(0) {}
    T info;
    QueueItem* next;
};
```



```
// sempre nel file Queue.h
```



```
template <class T>
class Queue {
public:
    Queue() : primo(0), ultimo(0) {}
    bool empty() const;
    void add(const T&);
    T remove();
    /* Attenzione: distruttore, costruttore di copia e
       assegnazione profondi */
    ~Queue();
    Queue(const Queue&);
    Queue& operator=(const Queue&);
private:
    QueueItem<T>* primo;        // primo el. della coda
    QueueItem<T>* ultimo;      // ultimo el. della coda
};
```


// sempre nel file Queue.h



```
template <class T>
bool Queue<T>::empty() const {
    return (primo == 0);
}

template <class T>
void Queue<T>::add(const T& val) {
    if(empty())
        primo = ultimo = new QueueItem<T>(val);
    else { // aggiunge in coda
        ultimo->next = new QueueItem<T>(val);
        ultimo = ultimo->next ;
    }
}
```

```

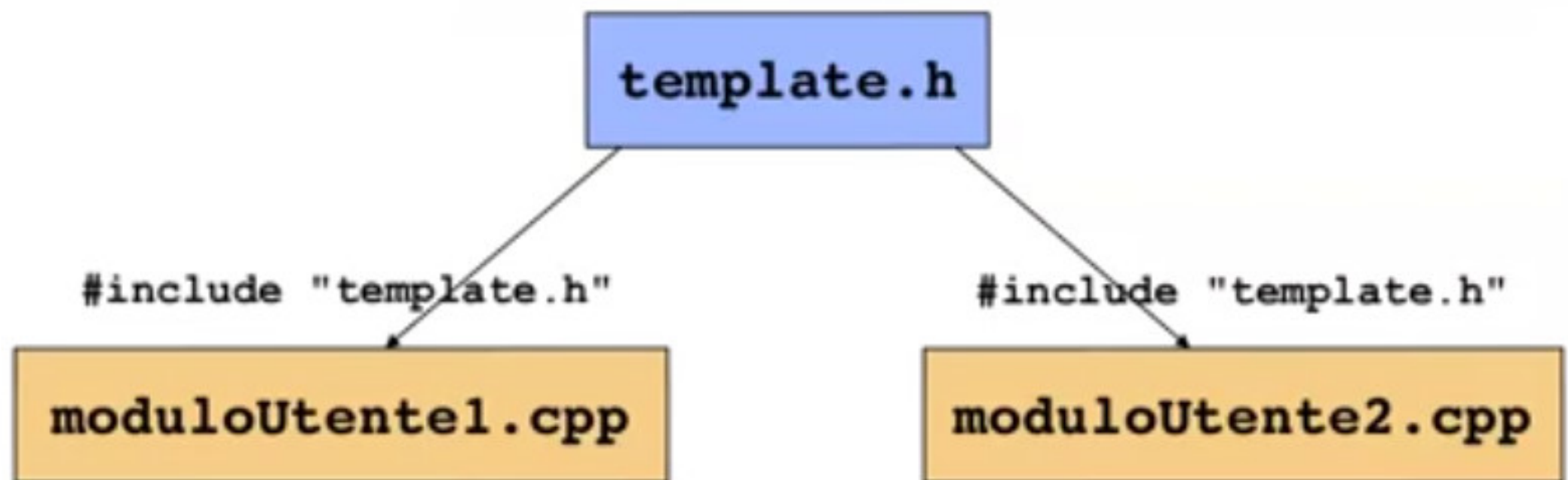
// nel file Queue.h
#include <iostream>

template <class T>
T Queue<T>::remove() {
    if (empty()) {
        std::cerr << "remove() su coda vuota" << std::endl;
        exit(1);    // BAD PRACTICE
    }
    QueueItem<T>* p = primo;
    primo = primo->next;
    T aux = p->info;
    delete p;
    return aux;
}

template <class T>
Queue<T>::~~Queue() { // distruzione profonda
    while (!empty()) remove();
}
#endif

```

Modello di compilazione per inclusione del template di classe



Vediamo mediante un esempio quando vengono create le istanze dei template di classe e dei template dei metodi.

```
#include<iostream>
using std::cout; using std::endl;
#include "Queue.h" // il file Queue.h contiene
                  // le definizioni dei template

int main() {
    Queue<int>* pi = new Queue<int>;
    // vengono istanziati la classe Queue<int> ed il suo
    // costruttore Queue<int>() perché new deve costruire un
    // oggetto della classe
    int i;
    for (i = 0; i < 10; i++) pi->add(i);
    // vengono istanziati i metodi add<int>
    // e empty<int>, la classe QueueItem<int> e il
    // suo costruttore QueueItem<int>()
    for (i = 0; i < 10; i++)
        cout << pi->remove() << endl;
    // viene istanziato il metodo remove<int> e
    // il distruttore standard ~QueueItem<int>
}
```

Amicizie in template di classe



Dichiarazione nel template di classe **C** di una classe o funzione **friend non template**

```
class A { ..... int fun(); ..... };

template<class T>
class C {
    friend int A::fun();
    friend class B;
    friend bool test();
};
```

La classe **B**, la funzione `test()` e il metodo `A::fun()` della classe **A** sono **friend** di tutte le istanze del template di classe **C**.

Dichiarazione nel template di classe **C** di un template di classe **A** o di funzione **fun** **friend associato**

```
template <class U1,...,class Uk> class A;  
template <class V1,...,class Vj> void fun(...);  
  
template <class T1,...,class Tn> class C {  
    friend class  A<...,Tj,...>;  
    friend void fun<...,Tj,...>(...);  
};
```



EXAMPLE

```
template<class T>
class C {
private:
    T t;
public:
    C(const T&);
    friend void f_friend<T>(const C<T>&);
    // amicizia associata
};

template<class T>
C<T>::C(const T& x) : t(x) {}

template<class T>
void f_friend(const C<T>& c) {
    cout << c.t << endl; // per amicizia
}

int main() {
    C<int> c1(1); C<double> c2(2.5);
    f_friend(c1); // stampa: 1
    f_friend(c2); // stampa: 2.5
}
```


Dichiarazione nel template di classe **C** di un template di classe o di funzione **friend non associato**

```
template<class T>
class C {

    template<class Tp>           // amicizia con template
    friend int A<Tp>::fun();    // di metodo

    template<class Tp>           // amicizia con template
    friend class B;             // di classe

    template<class Tp>           // amicizia con template
    friend bool test(C<Tp>);    // di funzione
};
```

Alcuni compilatori pre-standard non supportavano quest'ultima tipologia di dichiarazioni `friend`. Il compilatore GNU `g++` supporta i template di classe e di funzione `friend` non associati.

EXAMPLE

```
template <class T>
class C {
    template <class V>
    friend void fun(const C<V>&); // amicizia non associata
private:
    T x;
public:
    C(const T& y): x(y) {}
};

template <class T>
void fun(const C<T>& t) {
    cout << t.x << " "; // per amicizia (associata)
    C<double> c(3.1);
    cout << c.x << endl; // per amicizia NON ASSOCIATA
}

int main(){
    C<int> c(4);
    C<string> s("pippo");
    fun(c); // stampa: 4 3.1, istanz.implicita fun<int>
    fun(s); // stampa: pippo 3.1, istanz.implicita fun<string>
}
```


È naturale associare ad ogni istanza di `QueueItem` una sola istanza amica della classe `Queue`, ovvero quella associata.

```
template <class T>
class QueueItem {
friend class Queue<T>; ←
private:
    T info;
    QueueItem* next;
    QueueItem(const T& val) : info(val), next(0) {}
};
```

```
template <class T>
ostream& operator<<(ostream&, const Queue<T>&) ;
```

Di quali amicizie abbiamo bisogno?

```
template <class T>
ostream& operator<<(ostream& os, const Queue<T>& q) {
    os << "(" ;
    QueueItem<T>* p = q.primo; // amicizia con Queue
    for (; p != 0; p = p->next) // amicizia con QueueItem
        os << *p << " "; // operator<< per il tipo QueueItem
    os << ")" << endl;
    return os;
}
```

Dobbiamo quindi dichiarare `operator<<` come funzione amica associata sia della classe `Queue` che della classe `QueueItem`.



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

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