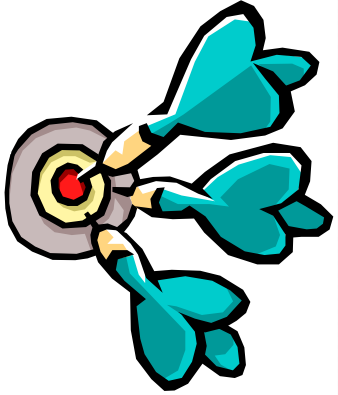
Programmazione concorrente in C++11 - Sincronizzazione

Programmazione di Sistema

A.A. 2017-18

Argomenti



- Operazioni atomiche
- Condition variable
- Esecuzione di task singoli

Operazioni atomiche

- Le normali operazioni di accesso in lettura e scrittura non offrono nessuna garanzia sulla visibilità delle operazioni che sono eseguite in parallelo da più thread
- I processori supportano alcune istruzioni specializzate per permettere l'accesso atomico ad un singolo valore

La classe std::atomic<T>

- Offre la possibilità di accedere in modo atomico al tipo T
  - Garantendo che gli accessi concorrenti alla variabile sono osservabili nell'ordine in cui avvengono
  - Questo garantisce il meccanismo minimo di sincronizzazione
- Le operazioni di lettura e scrittura di questi oggetti contengono al proprio interno istruzioni di memory fence
  - Che garantiscono che il sottosistema di memoria non mascheri il valore corrente della variabile

La classe std::atomic<T>

- Le operazioni non possono essere riordinate
  - Utili per segnalare condizioni di terminazione
  - Oppure per generare dipendenze di tipo "happens_before" tra attività differenti

La classe std::atomic<T>

```
std::atomic<boolean> done=false;

void task1() {
  //continua ad elaborare fino a che non viene detto di
smettere
  while (! done.load() ) {
    process();
  }
}

void task2() {
  wait_for_some_condition();
  //segnala che il task1 deve finire
  done.store(true);
  //...
}

void main() {
  auto f1=std::async(task1);
  auto f2=std::async(task2);
}
```

La classe std::atomic<T>

- Le operazioni di inizializzazione non sono atomiche
  - Quelle di accesso tramite load() e store(T t), sì
- Operazioni atomiche di tipo Read/Modify/Write
  - fetch_add(val) – aggiunge val al valore corrente (+=)
  - fetch_sub(val) – sottrae val dal valore corrente (-=)
  - operator++() – equivalente a fetch_add(1)

La classe std::atomic<T>

- Operazioni atomiche di tipo Read/Modify/Write
  - operator--() – equivalente a fetch_sub(1)
  - exchange(val) – assegna val e ritorna il valore precedente
- Il template offre alcune specializzazioni per i tipi int e boolean

Gestire il risveglio

- Spesso un thread deve aspettare uno o più risultati intermedi prodotti altri thread
  - Per motivi di efficienza, l'attesa non deve consumare risorse e deve terminare non appena un dato è disponibile
- La coppia di classi promise/future offrono una soluzione limitata del problema
  - Valida quando occorre notificare la disponibilità di un solo dato

Gestire il risveglio

- La presenza di dati condivisi richiede come minimo l'utilizzo di un mutex
  - Per garantire l'assenza di interferenze tra i due thread che devono fare accesso ai dati
- Il polling ha due limiti
  - Consuma capacità di calcolo e batteria in cicli inutili
  - Introduce una latenza tra il momento in cui il dato è disponibile e il momento in cui il secondo thread si sblocca

Gestire il risveglio

```
bool ready;
std::mutex readyFlagMutex;

// cicla fino a che ready vale true
{
  std::unique_lock<std::mutex>
    ul(readyFlagMutex);

  while (!ready) {
    //rilascio il lock per permettere all'altro thread di
prenderlo
    ul.unlock();

    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    ul.lock();
  }
  // uso la risorsa
} // rilascia il lock
```

std::condition_variable

- Modella una primitiva di sincronizzazione che permette l'attesa condizionata di uno o più thread
  - Fino a che non si verifica una notifica da parte di un altro thread, scade un timeout o si ferifica una notifica spuria
- Richiede l'uso di un std::unique_lock<Lockable>
  - Per garantire l’assenza di corse critiche nel momento del risveglio

std::condition_variable

- Offre il metodo wait(unique_lock) per bloccare l'esecuzione del thread
  - Fino a quando non giunge una notifica
  - Senza consumare cicli di CPU
- Un altro thread può informare uno o tutti i thread attualmente in attesa che la condizione si è verificata
  - Attraverso i metodi notify_one() e notify_all()

Implementazione

- Mantiene una lista di thread in attesa della condizione
  - Inizialmente la lista è vuota
  - Quando un thread esegue il metodo wait(...), viene sospeso e aggiunto alla lista
  - Quando sono eseguiti i metodi notify_one() o notify_all(), uno o tutti i thread presenti nella lista sono risvegliati
  - Si basa sul S.O. per sospendere/risvegliare i thread

La funzione di attesa

- È basata su un sistema a “doppia porta”
  - Attesa della segnalazione
  - Riacquisizione del Mutex
  - Solo quando entrambe sono state superate, il metodo ritorna

1. Aggiunge il thread corrente alla lista di quelli da risvegliare
2. Rilascia il lock
3. Sospende il thread
4. (attesa passiva)
5. Riacquisisce il lock

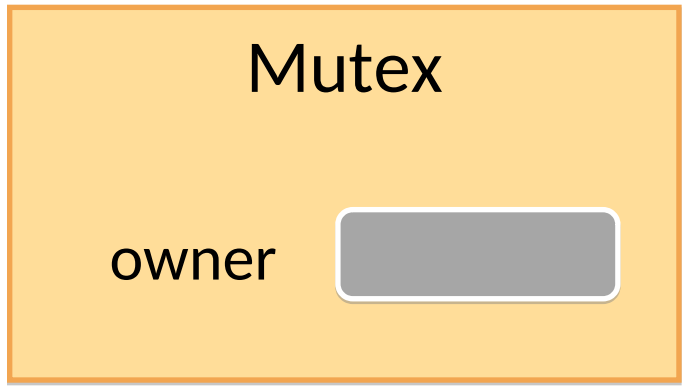
- Il metodo notify_one() sceglie un thread dalla lista di attesa e To risveglia
  - notify_all() li risveglia tutti

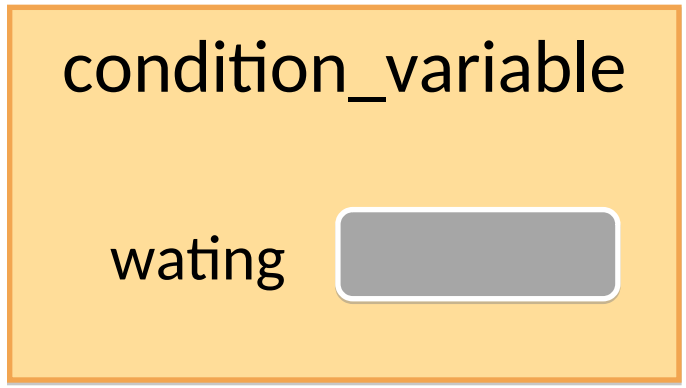
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato= ...;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```



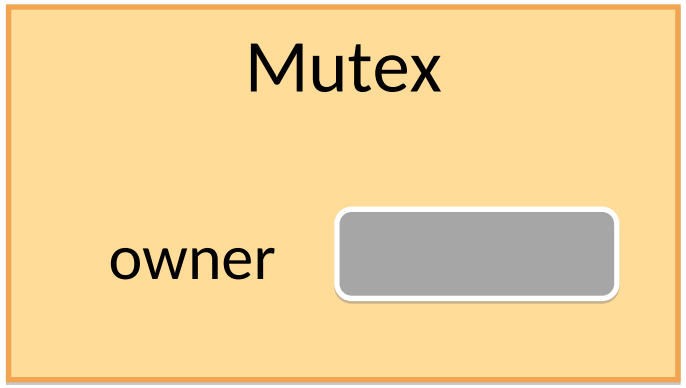


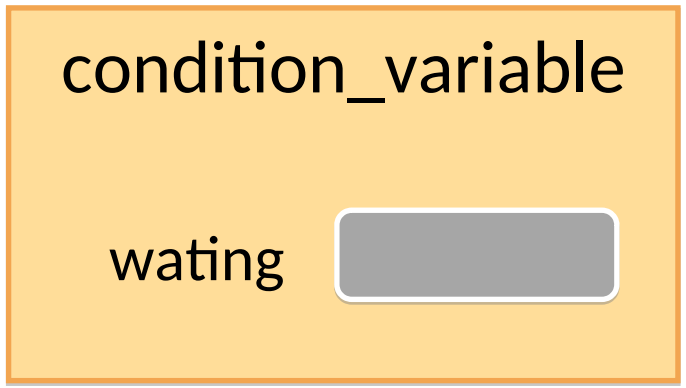
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
   ... //calcola un dato
   {
     lock_guard<mutex>
lg(m);
     dato= ...;
     cv.notify_one();
   }
   ...
}

void consume() {
   unique_lock<mutex>
ul(m);
   cv.wait(ul);
   //uso il dato
}
```



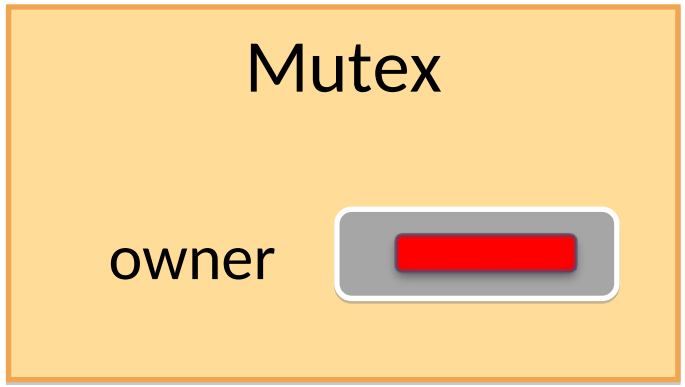


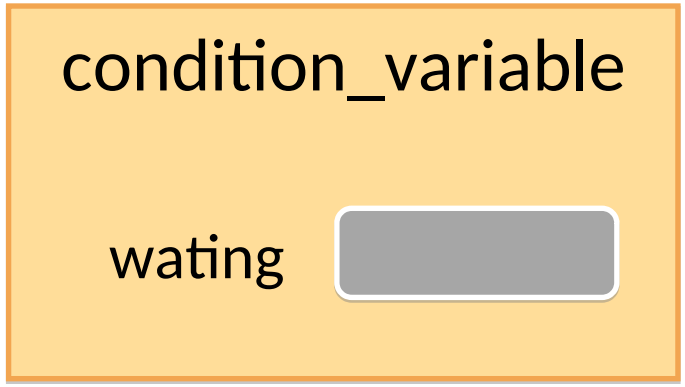
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato= ...;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```



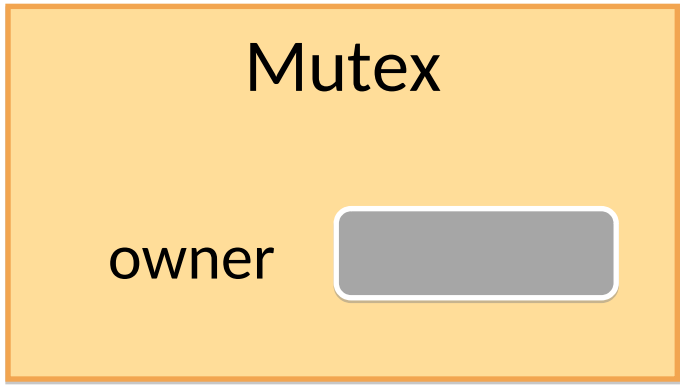


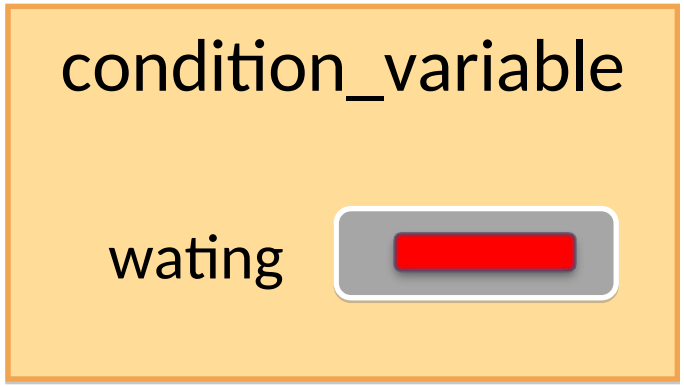
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato= ...;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```



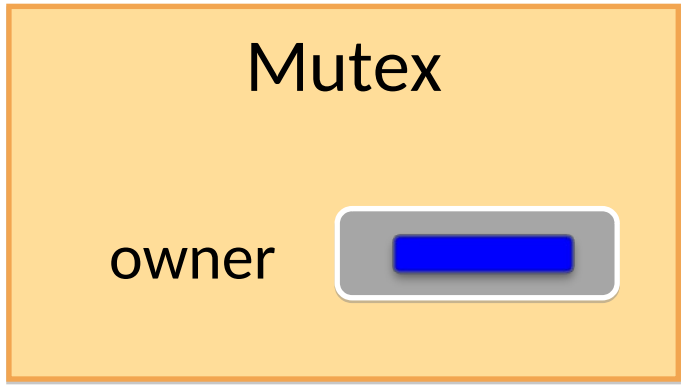


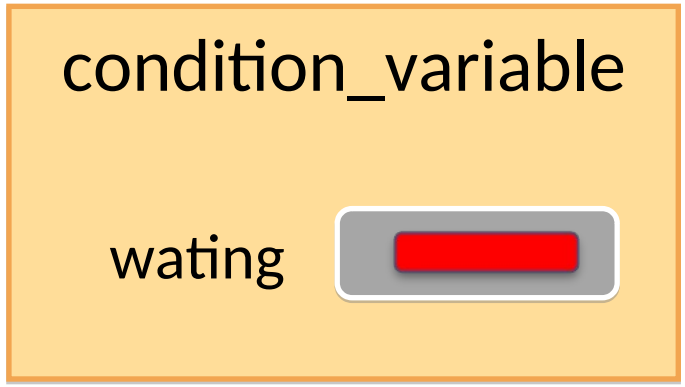
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato= ...;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```



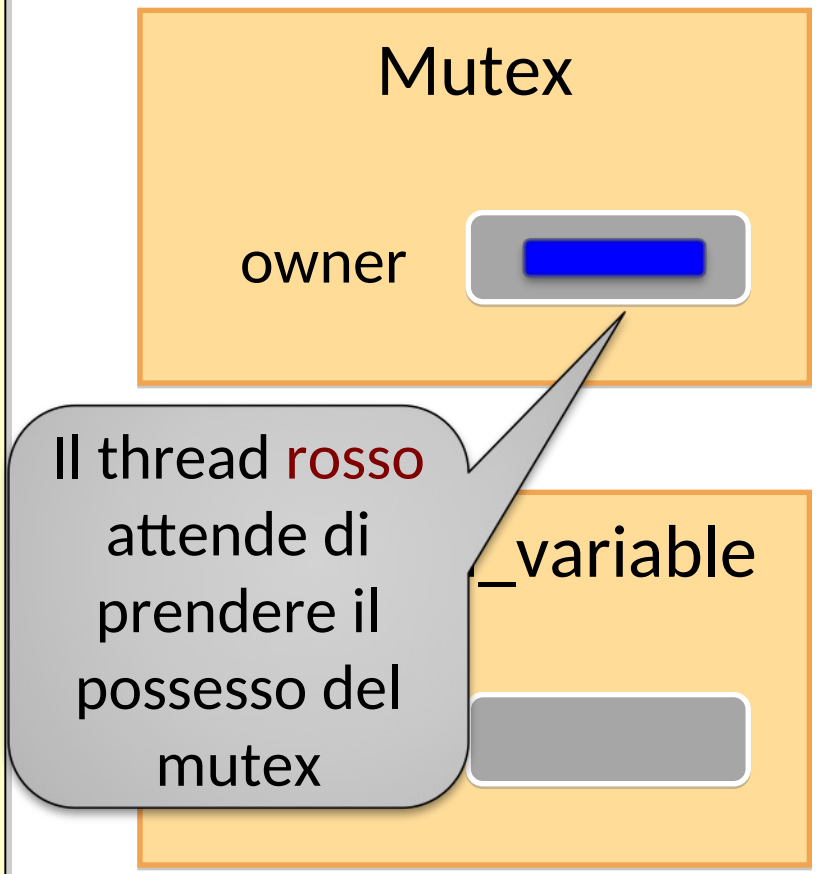


Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato= ...;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```

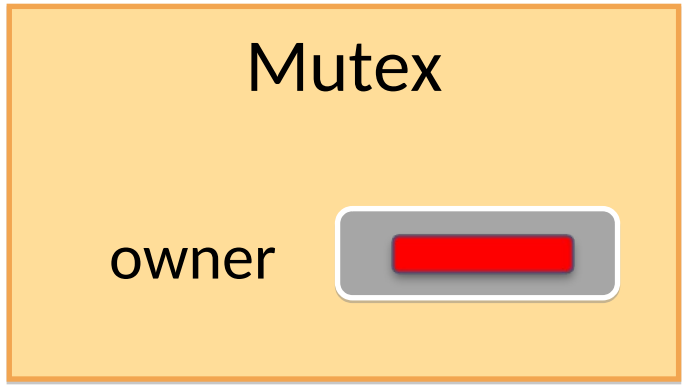


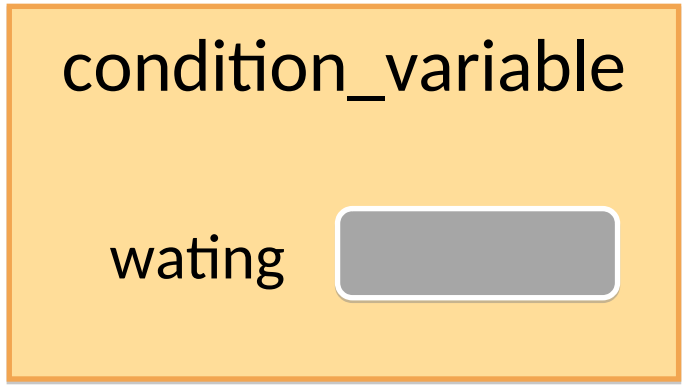
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato=    ;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```



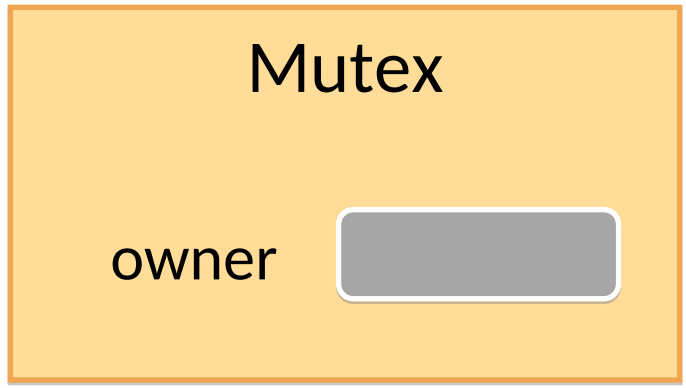


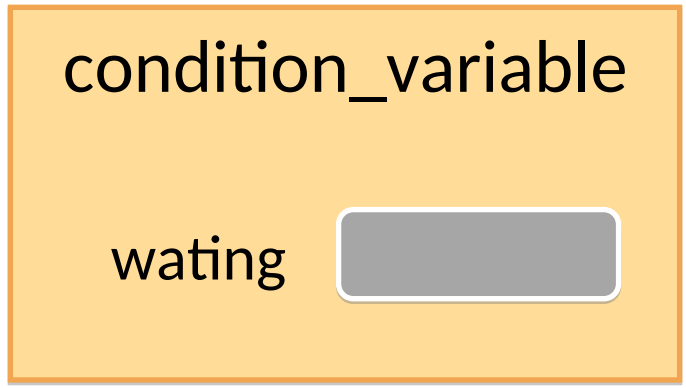
Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
  ... //calcola un dato
  {
    lock_guard<mutex>
lg(m);
    dato= ...;
    cv.notify_one();
  }
  ...
}

void consume() {
  unique_lock<mutex>
ul(m);
  cv.wait(ul);
  //uso il dato
}
```





std::condition_variable

- La presenza di un unico lock fa si che, se più thread ricevono la notifica, il risveglio sia progressivo
  - Non appena un thread rilascia il lock, un altro può acquisirlo e proseguire
- La relazione tra l'evento e la notifica è solo nella testa del programmatore
  - Per evitare notifiche spurie, si rende esplicito l'evento che si è verificato scrivendolo dentro una variabile condivisa (sotto il controllo del mutex)

std::condition_variable

- Permette, ad uno o più thread, di attendere, senza consumare risorse, la ricezione di una notifica
  - Proveniente da un altro thread
- È possibile che il thread sia risvegliato per altri motivi
  - Problema delle cosiddette **notifiche spurie**
  - Occorre, al ritorno dal metodo wait(), controllare se la condizione attesa è verificata

std::condition_variable

- Una versione overloaded del metodo wait, accetta come parametro un oggetto chiamabile
  - Alla ricezione di una notifica, il metodo wait invoca l'oggetto chiamabile
  - Ha il compito di valutare se l'evento è proprio quello atteso, restituendo true oppure false
  - Se il risultato è falso, si rimette in attesa; altrimenti ritorna al chiamante

Esempio

```
std::mutex mut;                     //protegge la coda
std::queue<data_chunk> data_queue; //dato condiviso
std::condition_variable data_cond; //indica che la coda non è
vuota

void data_preparation_thread() {
  while(more_data_to_prepare()) {
    data_chunk const data=prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk, [](){return !data_queue.empty();});
    data_chunk data=data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data)) break;
  }
}
```


wait_for(...) e wait_until(...)

- Limitano l'attesa nel tempo
  - Se chiamati senza indicare un oggetto chiamabile, restituiscono le costanti std::cv_status::timeout e std::cv_status::no_timeout per indicare l'esito dell'attesa
- Offrono l'opportunità al thread che esegue la notifica di completare la propria distruzione
  - Prima che i thread in attesa abbiano l'opportunità di osservare il dato condiviso

Lazy evaluation

- Ci sono varie situazioni in cui è utile rimandare la creazione ed inizializzazione di strutture complesse fino a quando non c'è la certezza del loro utilizzo
- In un programma sequenziale, ci si riferisce tipicamente alla struttura in questione attraverso un puntatore inizializzato a NULL

Lazy evaluation

- Quando occorre accedere alla struttura, si controlla se il puntatore abbia già un valore lecito
  - Nel caso in cui valga ancora NULL, si crea la struttura e se ne memorizza l'indirizzo all'interno del puntatore
- In un programma concorrente questa tecnica non può essere adottata direttamente
  - Il puntatore è una risorsa condivisa e il suo accesso deve essere protetto da un mutex

Lazy evaluation

- Per supportare questo tipo di comportamento, C++11 offre la classe
  - std::once_flag e la funzione std::call_once(...)
- Costituisce la struttura di appoggio per la funzione call_once
  - Registra, in modo thread safe, se è già avvenuta o sia in corso una chiamata a call_once

std::call_once(flag,f,...)

- Esegue la funzione f una sola volta
  - Se dal flag non risultano chiamate, inizia l'invocazione di f
  - Se un'altra chiamata è in corso, blocca l'esecuzione in attesa del suo risultato

std::call_once(flag,f,...)

```
#include <mutex>

class Singleton {
  static Singleton *instance;
  static std::once_flag inited;

  Singleton() {...} //privato

public:
  static Singleton *getInstance() {
    std::call_once( inited, []() {
        instance=new Singleton();
      });
    return instance;
  }

  //altri metodi...

};
```


Spunti di riflessione

- Si realizzi la classe generica Buffer che mantiene una coda di oggetti di tipo T
  - Il metodo push(T) inserisce un nuovo elemento
  - Il metodo pop() restituisce l'elemento più vecchio e rimane bloccato finché il buffer è vuoto