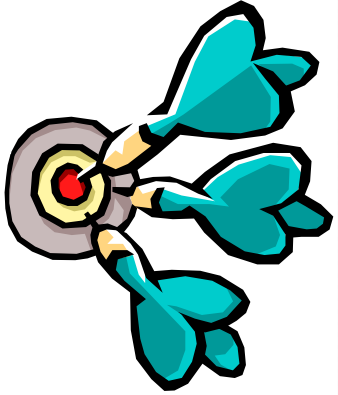
Programmazione concorrente in C++11 - Parte I

Programmazione di Sistema

A.A. 2017-18

Argomenti



- Esecuzione asincrona
- Attesa dei risultati
- Sincronizzazione

Programmazione concorrente in C++11

- Il concetto di thread è parte integrale dello standard del linguaggio
  - Facilita la scrittura di programmi portabili
- Introduce un livello di astrazione più elevato che facilita il programmatore
  - Permettendogli di concentrarsi sugli aspetti specifici del proprio algoritmo invece che sui dettagli del S.O.

Programmazione concorrente in C++11

- Sono disponibili due approcci
  - Uno di alto livello, basato su std::async e std::future
  - Uno di basso livello che richiede l'uso esplicito di thread e costrutti di sincronizzazione
- Entrambi richiedono un compilatore aggiornato
  - VisualStudio a partire dalla versione 2O12
  - GCC a partire dalla versione 4.7.x (con parte delle funzionalità già presenti in 4.6.y)

Esecuzione asincrona e risultati futuri

- Spesso, un compito complesso può essere decomposto in una serie di compiti più semplici
  - I cui risultati potranno essere poi combinati per fornire la funzionalità complessiva
- Due sotto-compiti sono tra loro indipendenti se
  - La computazione di uno non dipende da quella di un altro
  - Non fanno accesso, in scrittura, a dati condivisi
- Due sotto-compiti indipendenti possono essere eseguiti in parallelo
  - Combinando poi i risultati quando sono disponibili

Esecuzione asincrona e risultati futuri

- La funzione std::async e la classe std::future rendono molto semplice questo tipo di decomposizione
  - Entrambe sono definite nel file <future>

std::async()

- Prende come parametro un oggetto chiamabile
  - Puntatore a funzione o oggetto funzionale che ritornano un dato di tipo generico T ed eventuali parametri da passare
  - Restituisce un oggetto di tipo std::future<T>
- Quando questa funzione viene eseguita, se possibile, invoca in un thread separato l'oggetto chiamabile con i relativi parametri
  - In ogni caso, ritorna immediatamente, senza attenderne il completamento della funzione chiamata

Accedere al risultato

- Si accede al risultato dell'esecuzione invocando il metodo get() sull'oggetto future ritornato
  - Se l'esecuzione è andata a buon fine, restituisce il risultato
  - Se il thread secondario è terminato con un'eccezione, la rilancia nel thread corrente
  - Se l'esecuzione è ancora in corso, si blocca in attesa che finisca
  - Se l'esecuzione non è ancora iniziata, ne forza l'avvio nel thread corrente

Esempio

```
#include <future>
#include <string>
std::string f1(std:string p1, double p2) { … }
std::string f2(int p) { … }

int main() {
  // calcola f1("…", 3.14) + f2(18)
  std::future<std::string> future1 =
          std::async(f1, "…", 3.14);
  std::string res2= f2(18);
  std::string res1 = future1.get();
  std::string result = res1+res2;
}
```

Mentre f1() viene calcolato, esegue f2 nel thread principale

std::async

- La funzione accetta eventuali parametri da passare all'oggetto chiamabile
  - std::async(f1, "...", 3.14);
  - L'oggetto chiamabile può essere preceduto da una politica di lancio
- std::launch::async
  - Attiva un thread secondario
  - Lancia un'eccezione di tipo system_error se il multithreading non è supportato o non ci sono risorse per creare un nuovo thread
- std::launch::deferred
  - Valutazione pigra: l'oggetto chiamabile sarà valutato solo se e quando qualcuno chiamerà get() o wait() sul future relativo
  - Non verrà generato alcun thread aggiuntivo
- Se la politica viene omessa
  - Il sistema prova dapprima ad attivare un thread secondario
  - Se non può farlo, segna l'attività come deferred

Attenzione!

- La creazione di un thread secondario ha un costo significativo
  - Se le operazioni da eseguire sono poche (dell'ordine di un migliaio di cicli macchina) conviene eseguirle direttamente

Esempio

```
template <typename Iter>
int parallel_sum(Iter beg, Iter end)
{
    typename Iter::difference_type len = end-beg;
    if(len < 1000)     // pochi elementi,
                       //conviene valutazione sincrona
        return std::accumulate(beg, end, 0);

    Iter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<Iter>, mid,
end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(10000, 1);
    std::cout << "Somma:" << parallel_sum(v.begin(),
v.end()) << '\n';
}
```

std::future<T>

- Fornisce un meccanismo per accedere in modo sicuro e ordinato al risultato di un'operazione asincrona
  - Il tipo T rappresenta il tipo del risultato ritornato
- Mantiene internamente uno stato condiviso con il blocco di codice responsabile della produzione del risultato
- Quando si invoca il metodo get() lo stato condiviso viene rimosso
  - L'oggetto future entra in uno stato invalido
  - get() può essere chiamato UNA SOLA VOLTA

std::future<T>

- Per forzare l'avvio del task e attenderne la terminazione, senza prelevare il risultato, si utilizza il metodo wait()
  - Può essere chiamato più volte: se il task è già terminato, ritorna immediatamente
- È possibile attendere per un po' di tempo il completamento del compito, attraverso i metodi wait_for(...) e wait_until(...)

std::future<T>

- Wait_for e wait_until NON forzano l'avvio
  - Se il task è stato lanciato con la modalità deferred
- Richiedono un parametro, rispettivamente di tipo
  - std::chrono::duration
  - std::chrono::time_point
- Indicando una durata nulla, permettono di scoprire se il task sia o meno già terminato

std::future<T>

- Restituiscono
  - std::future_status::deferred se la funzione non è ancora partita
  - std:: future_status::ready se il risultato era già pronto o lo è diventato nel tempo di attesa
  - std:: future_status::timeout,
    se il tempo è scaduto, senza che il risultato sia diventato pronto

Attenzione!

- Quando un oggetto future viene distrutto, il distruttore ne attende la fine
  - Se la computazione è ancora attiva, questo può comportare attese significative
- Le attività lanciate in questo modo, non sono cancellabili, se non avendo cura di condividere, con la funzione chiamata, una variabile che possa essere usata come criterio di terminazione
  - Attenzione al riordinamento introdotto dal compilatore ed alle barriere di memoria

Esempio

```
int quickComputation(); //approssima il risultato con un'euristica
int accurateComputation(); // trova la soluzione esatta,
                           // richiede abbastanza tempo
std::future<int> f; // dichiarato all'esterno perché il suo ciclo di
vita
                    // potrebbe estendersi più a lungo della
funzione
int bestResultInTime()
{
  // definisce il tempo disponibile
  auto tp = std::chrono::system_clock::now() +
std::chrono::minutes(1);
  // inizia entrambi i procedimenti
  f = std::async (std::launch::async, accurateComputation);
  int guess = quickComputation();
  // trova il risultato accurato, se disponibile in tempo
  std::future_status s = f.wait_until(tp);
  // ritorna il miglior risultato disponibile
  if (s == std::future_status::ready) {
    return f.get();
  } else {
    return guess; // accurateComputation() continua...
  }
}
```

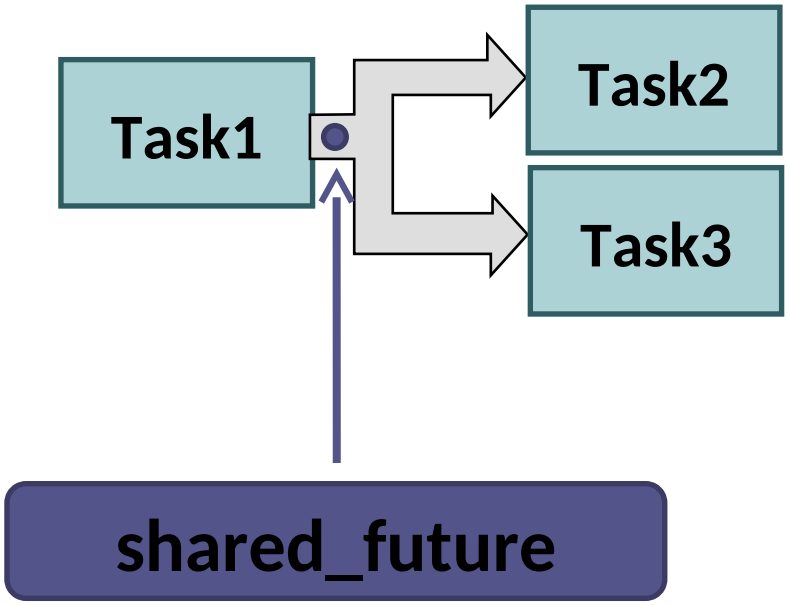
std::shared_future<T>

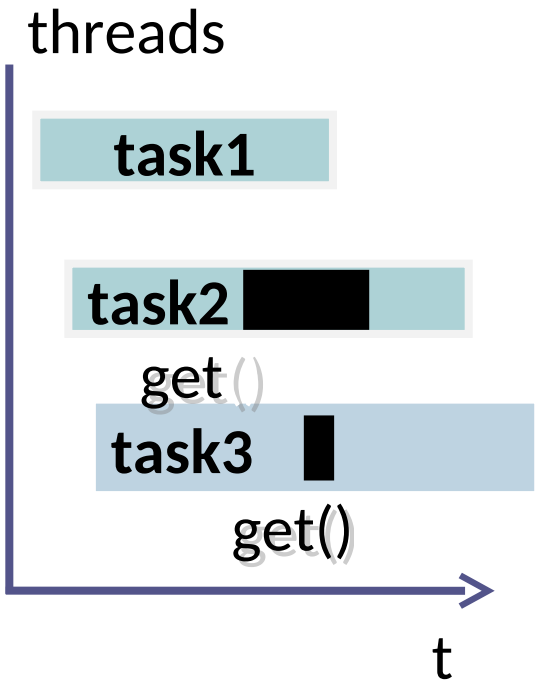
- La classe future<T> mette a disposizione il metodo share()
  - Utile se in più posti occorre poter valutare se un'operazione asincrona sia terminata e quale risultato abbia prodotto
- Restituisce un oggetto di tipo std::shared_future<T>
- Invalida lo stato dell'oggetto corrente
  - Che non può più essere usato

std::shared_future<T>

- std::shared_future<T> è copiabile oltre che movibile
  - A differenza di std::future<T> che può solo essere mosso
- A parte share(), offre gli stessi metodi di future<T>
  - Se get() viene chiamato più volte, produce sempre lo stesso risultato (a parte l'attesa)
- Per realizzare catene di elaborazione asincrone
  - In cui i risultati delle fasi iniziali sono messi a disposizione delle fasi successive
  - Senza ridurre, a priori, il grado di parallelismo

std::shared_future<T>





std::shared_future<T>

```
int task1(); // prima fase del calcolo

std::string task2(std::shared_future<int1>); // seconda fase
double task3(std::shared_future<int>); // terza fase

int main() {
  std::shared_future<int> f1 =
                  std::async(std::launch:async, task1).share();
  std::future<std::string> f2= std::async(task2, f1);
  std::future<double> f3 = std::async(task3, f1);

  try {
    std::string str = f2.get();
    double d= f3.get();
  } catch (std::exception& e) {
    //gestisci eventuali eccezioni
  }
}
```


Accedere a dati condvisi

- Capita frequentemente che due thread differenti debbano fare accesso ad una stessa informazione
  - In assenza di sincronizzazione, gli accessi potrebbero sovrapporsi e dare origine a interferenza
- Occorre proteggere gli accessi condivisi attraverso un opportuno meccanismo
  - La principale astrazione messa a disposizione dalla libreria C++11 è quella offerta dalla classe std::mutex

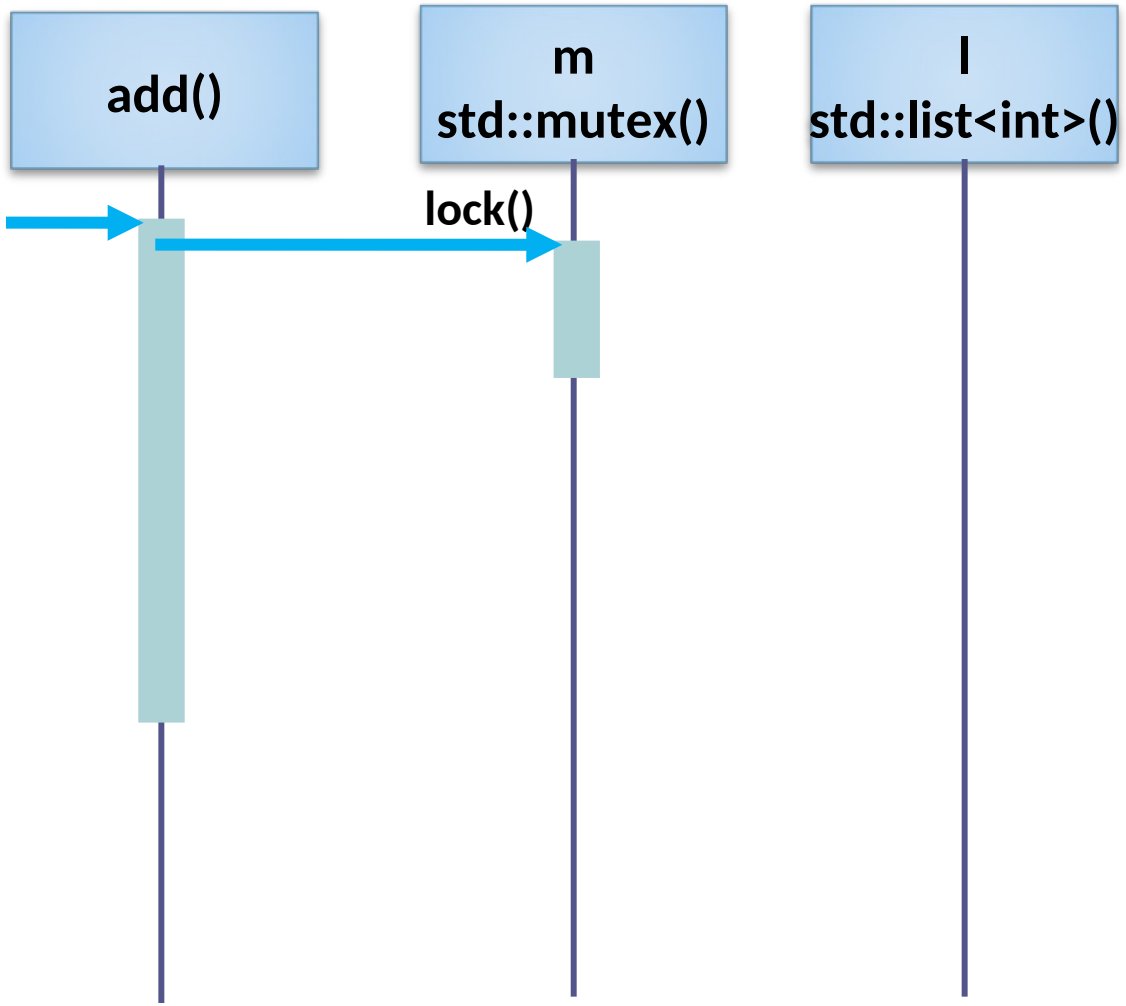
std::mutex

- Gli oggetti di questa classe permettono l'accesso controllato a porzioni di codice ad un solo thread alla volta
  - Mutex: contrazione di Mutual Exclusion
  - Definito nel file <mutex>
- Tutto il codice che fa accesso ad una data informazione condivisa deve fare riferimento ad uno stesso oggetto mutex
  - E racchiudere le operazioni tra le chiamate ai metodi lock() e unlock()

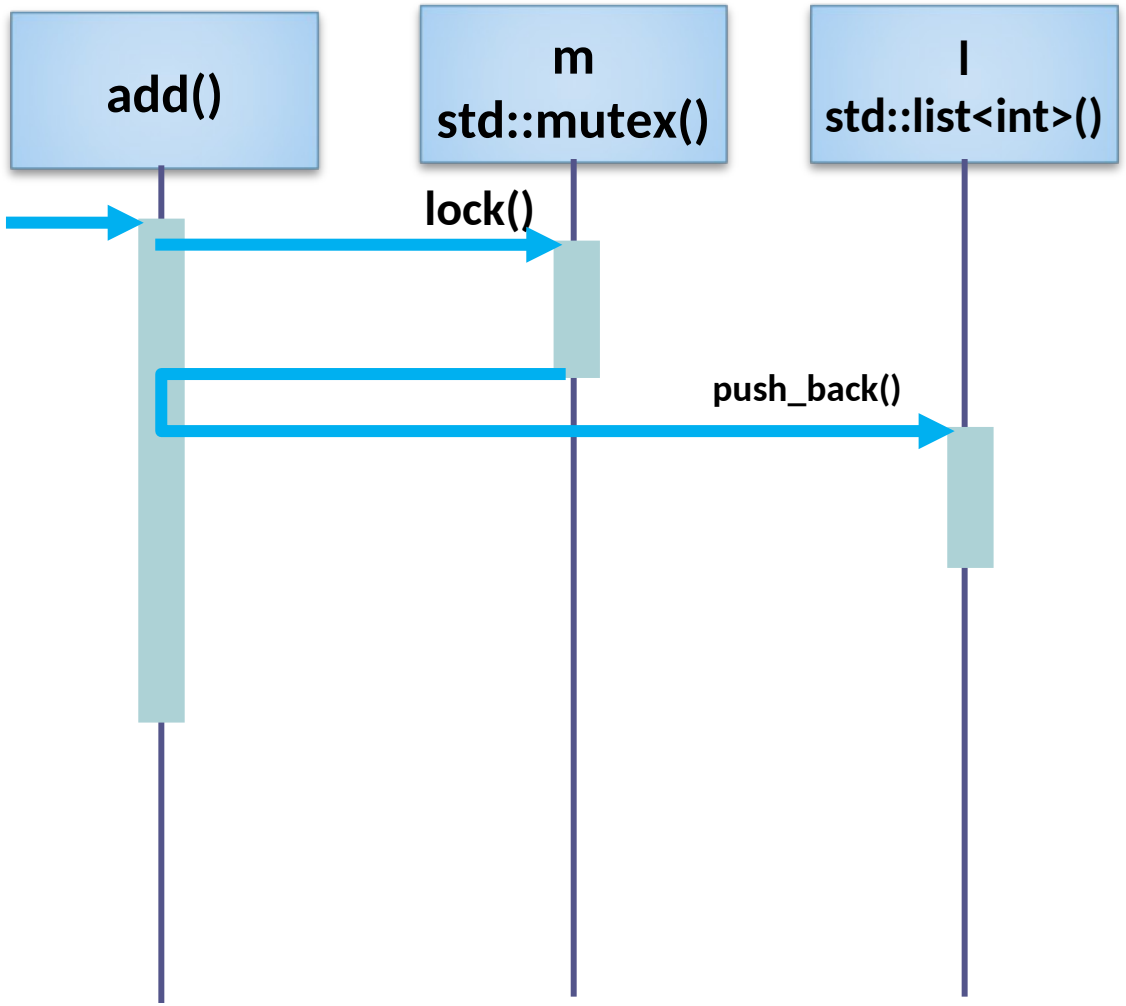
std::mutex

- lock() e unlock() entrambe includono una barriera di memoria
  - Garantisce la visibilità delle operazioni eseguite fino a quel punto
- Se un thread invoca il metodo lock() di un mutex mentre questo è posseduto da un altro thread, il primo thread si blocca in attesa che l'altro chiami unlock()
- Occorre fare in modo che, se si è preso possesso di un mutex tramite lock(), prima o poi lo si rilasci con unlock()

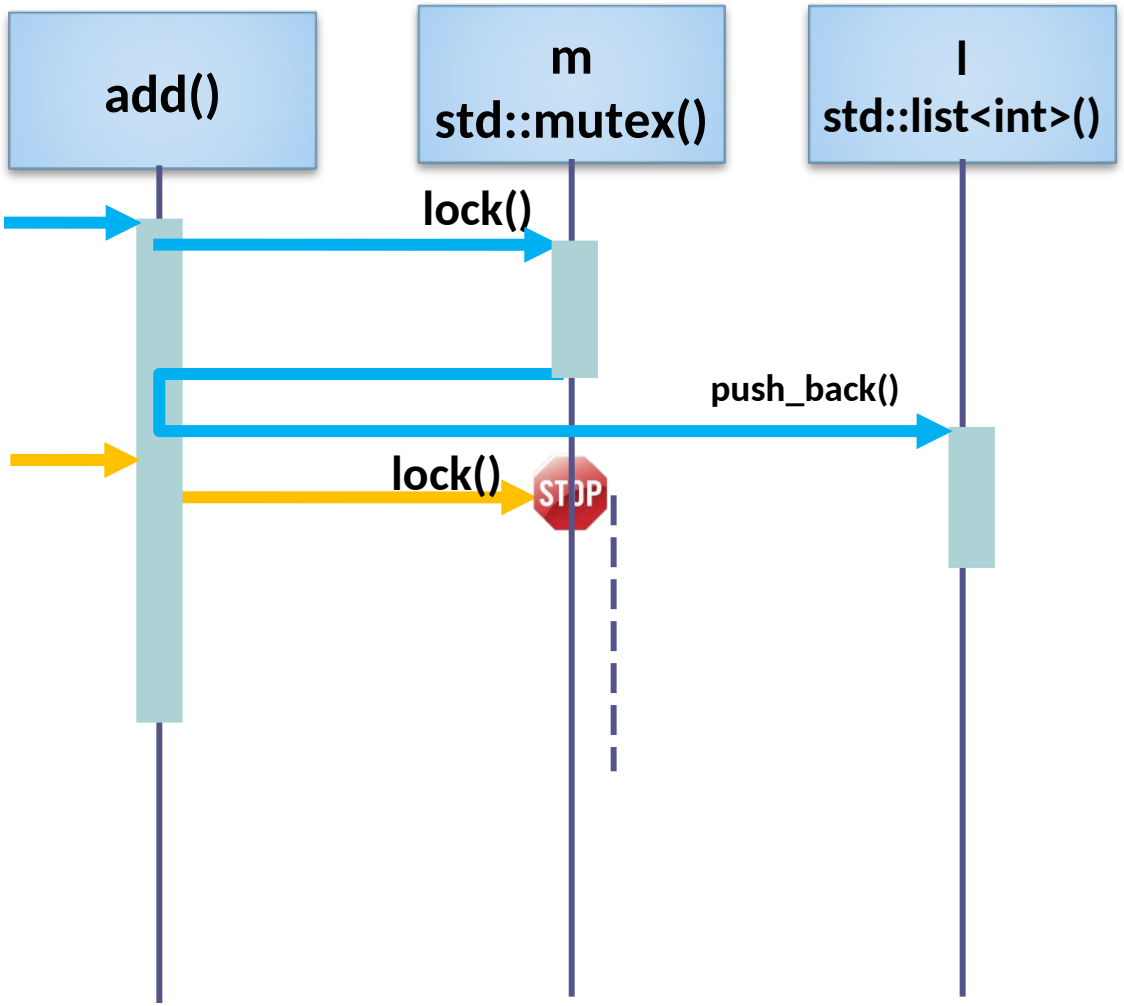
```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



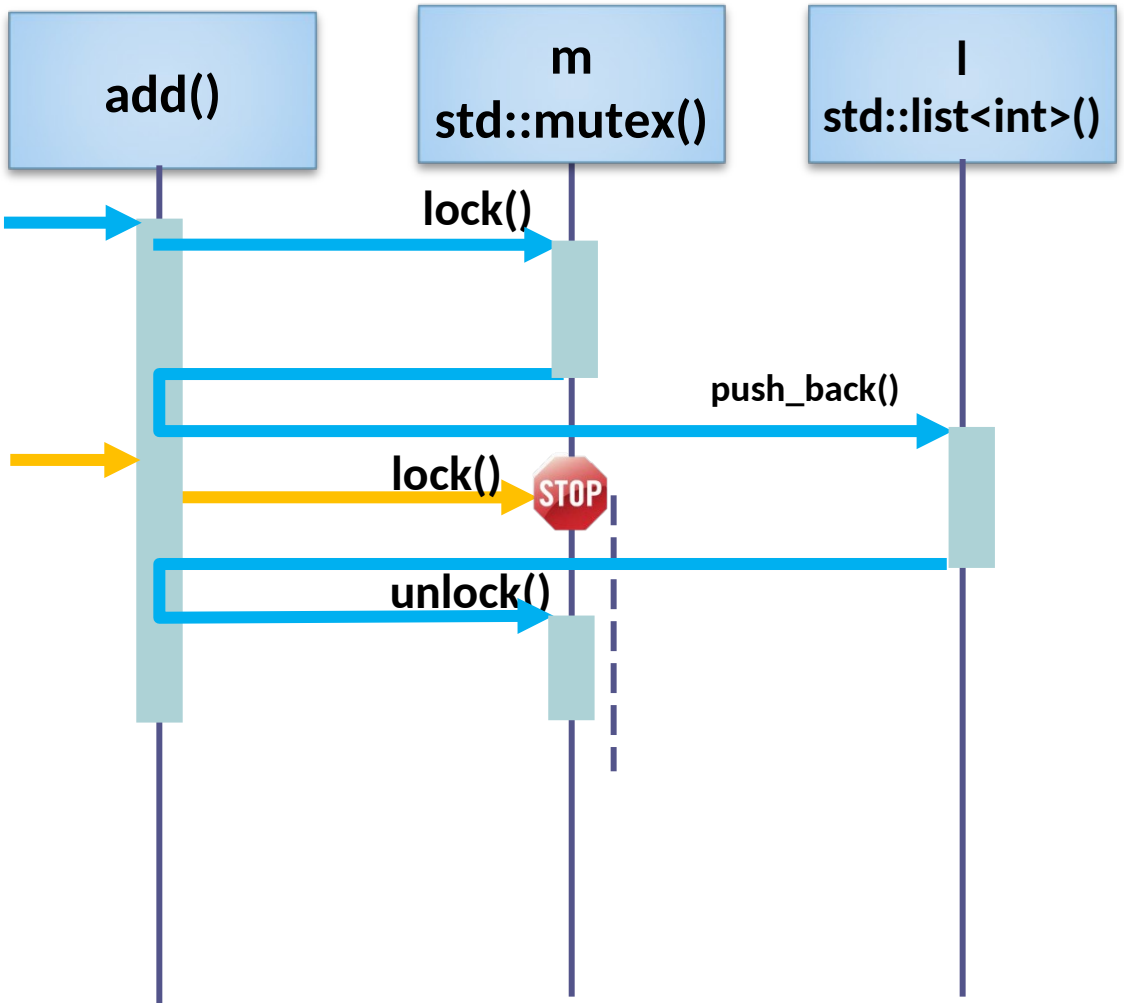
```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



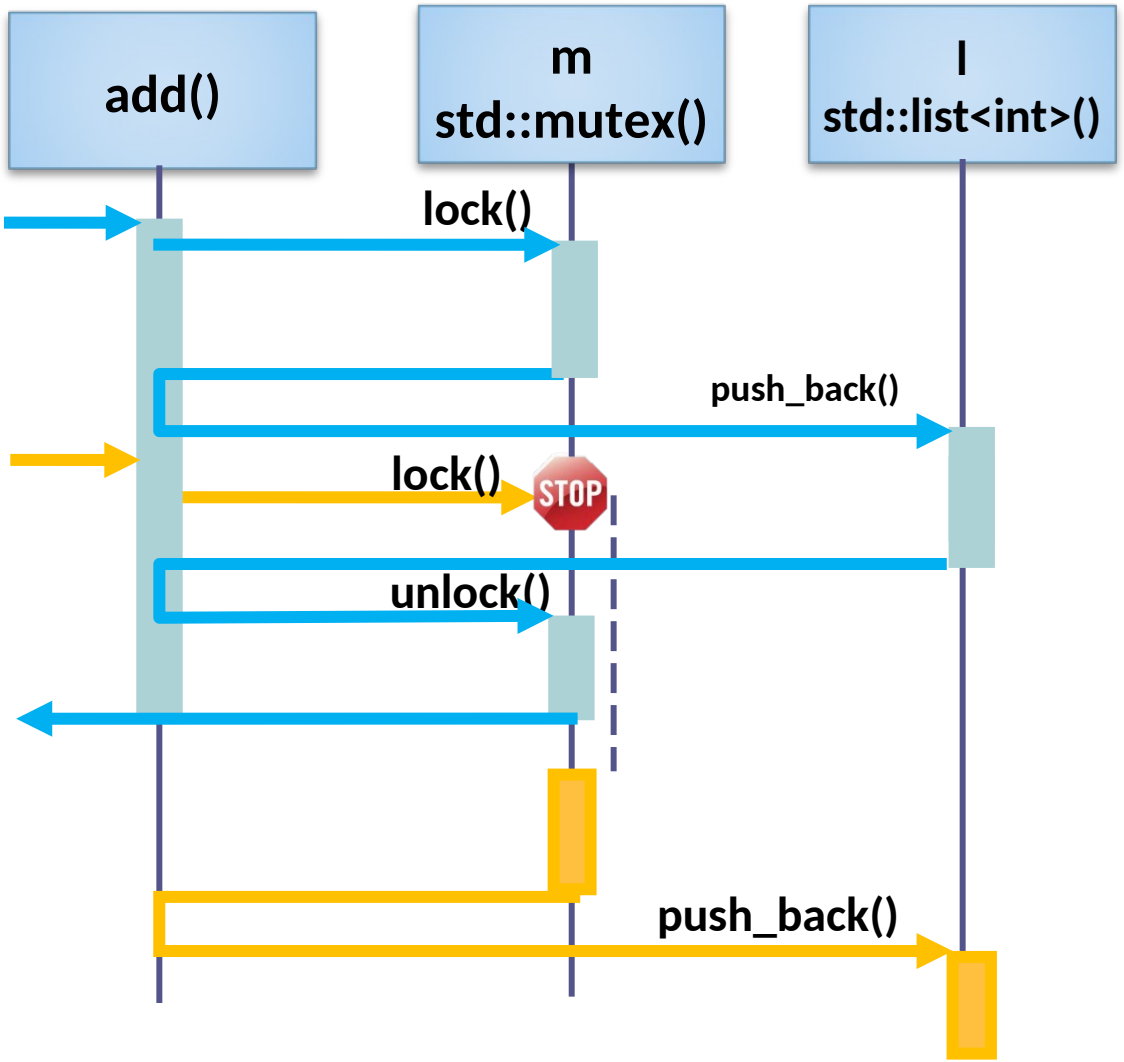
```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



```
std::list<int> l;
std::mutex m;

void add(int i){

  m.lock();

  l.push_back(i);

  m.unlock();

}
```



std::mutex

- Non c'è corrispondenza diretta tra l'oggetto mutex e la struttura dati che esso protegge
  - La relazione è nella testa del programmatore
- Tutti gli accessi alla struttura vanno protetti acquisendo dapprima il mutex
  - Anche le operazioni in sola lettura
- Un oggetto mutex può proteggere molte strutture diverse
  - Ma riduce il grado di parallelismo complessivo del programma
- Un mutex non è ricorsivo
  - Se un thread cerca di acquisirlo (tramite lock()) due volte, senza rilasciarlo, il thread si blocca per sempre

Regolare l'attesa

- std::recursive_mutex può essere acquisito più volte consecutivamente dallo stesso thread
  - Dovrà essere rilasciato altrettante volte, prima di consentire ad un altro thread di acquisirlo
- std::timed_mutex aggiunge i metodi try_lock_for() e try_lock_until()
  - Pone un limite al tempo di attesa massimo
  - Se il tempo scade senza che lock venga acquisito, restituiscono false

Regolare l'attesa

- std:recursive_timed_mutex unisce i due comportamenti
  - Il lock può essere acquisito più volte da parte dello stesso thread
  - È possibile limitare il tempo massimo di attesa

std::mutex

- Ogni volta che si acquisisce un mutex, occorre rilasciarlo
  - Anche se si verifica un'eccezione

std::lock_guard<Lockable>

- Per semplificare il codice e garantire che un mutex sia sempre rilasciato, viene messa a disposizione la classe generica std::lock_guard<Lockable>
  - Utilizza il paradigma RAII
- Il costruttore invoca il metodo lock() dell'oggetto passato come parametro
  - Il distruttore invoca unlock()
- Non offre nessun altro metodo

Esempio

```
template <class T>
class shared_list {
    std::list<T> list;
    std::mutex m;

    T& operator=(const shared_list<T>& that);
    shared_list(const shared_list<T>& that);

  public:

    int size() { std::lock_guard<std::mutex> l(m); return
list.size(); }

    T front() { std::lock_guard<std::mutex> l(m); return
list.front(); }

    void push_front(T t) {
      std::lock_guard<std::mutex> l(m);
      list.push_front(t);
    }

    //ecc.
};
```

Blocco condizionale

- In alcune situazioni, un programma non intende bloccarsi se non può acquisire un mutex
  - Ma fare altro nell'attesa che si liberi
- La classe mutex mette a disposizione il metodo try_lock()
  - Restituisce un valore booleano per indicare se è stato possibile acquisirlo o meno
- Se l'acquisizione ha avuto successo, il mutex può essere "adottato" da un oggetto lock_guard
  - Così da garantirne il rilascio al termine dell'utilizzo

Esempio

```
#include <mutex>

std::mutex m;

...
void someFunction() {
  while (m.try_lock() == false) {
    do_some_work();
  }

  std::lock_guard<std::mutex> l(m, std::adopt_lock);

  // l registra m al proprio interno, senza cercare di
acquisirlo

  ...

  //quando l viene distrutto, rilascia il possesso del mutex
}
```


std::unique_lock<Lockable>

- Estende il comportamento di lock_guard
  - È possibile rilasciare e riacquisire l'oggetto Lockable
  - Tramite i metodi lock() e unlock()
- I metodi lock() e unlock() lanciano un'eccezione di tipo std::system_error
  - se si cerca di rilasciare qualcosa che non si possiede o viceversa

std::unique_lock<Lockable>

- Il costruttore offre numerose politiche di gestione selezionate in base al secondo parametro
  - adopt_lock verifica che il thread possieda già il Locakble passato come parametro e lo adotta
  - defer_lock si limita a registrare il riferimento al Lockable, senza cercare di acquisirlo

Spunti di riflessione

- Si scriva un programma C++11 che utilizzi la funzione async per effettuare le statistiche sul numero di vocali presenti in due file di testo e stampi i risultati nel thread principale