GAMECENTRIC

Alberto Barbati
alberto@gamecentric.com

Coming Soon to a C++ Compiler Near You: Coroutines

#itcppcon18

Italian C++ Conference 2018

June 23, Milan

Alberto Barbati

- Programmatore C++ entusiasta dal 1990
- Nella game industry dal 2000
- Specializzato in videogame e tool per la produzione video-ludica
- Segue con attenzione i lavori della C++ Committee dal 2008

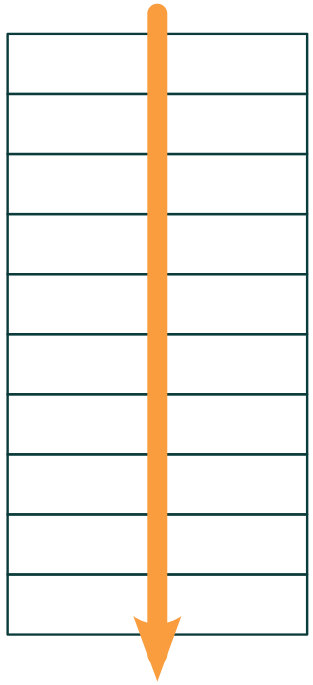
Argomenti di oggi

- Breve introduzione alle Coroutine
- Coroutine e C++: passato, presente e futuro
- Il Coroutines TS nel dettaglio

Introduzione alle coroutine

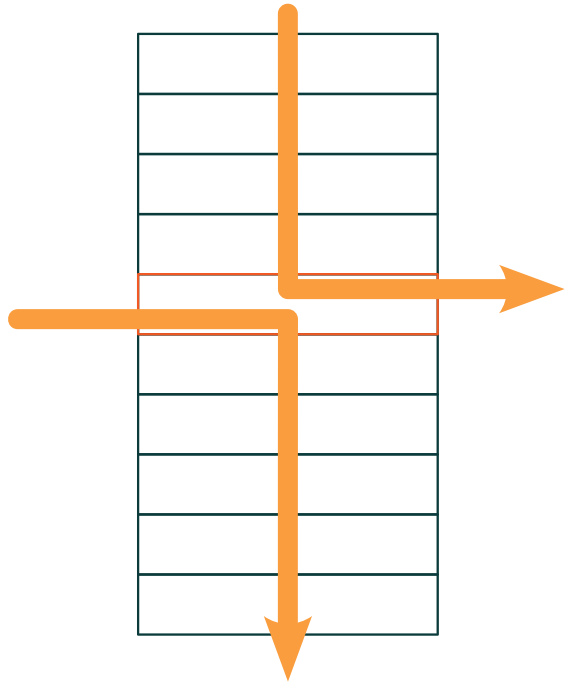
Routine

- Una funzione o routine è un sequenza di istruzioni che normalmente viene eseguita dall'inizio alla fine



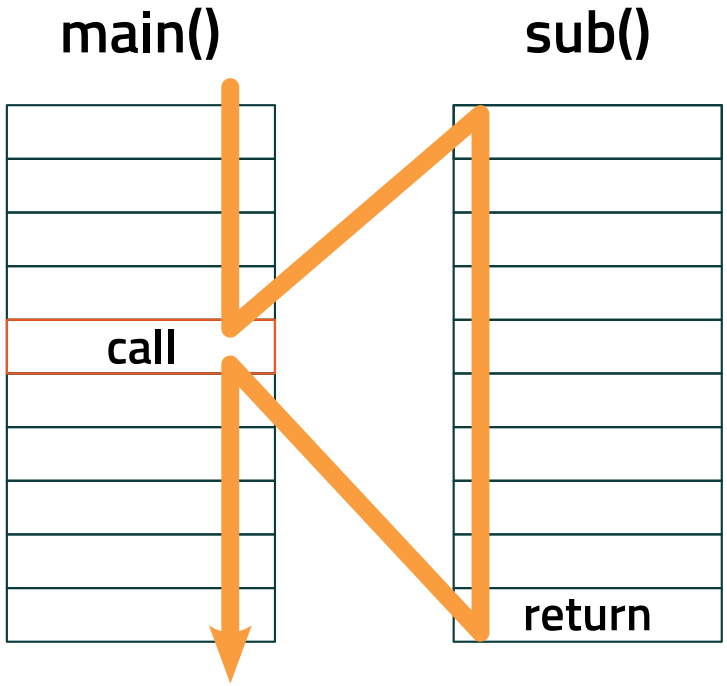
Punto di sospensione e riavvio

- Un punto di sospensione e riavvio è un punto della sequenza in cui l'esecuzione può essere sospesa (*suspend*) per eseguire codice esterno alla routine
- In un secondo tempo, l'esecuzione potrà essere riavviata (*resume*) e proseguire da dove si era interrotta



Chiamata a subroutine

- Una chiamata a subroutine è un punto di sospensione e riavvio in cui l'esecuzione di una routine principale viene sospesa per eseguire un'altra routine subordinata
- Raggiunto il termine della subroutine, l'esecuzione della routine principale viene riavviata

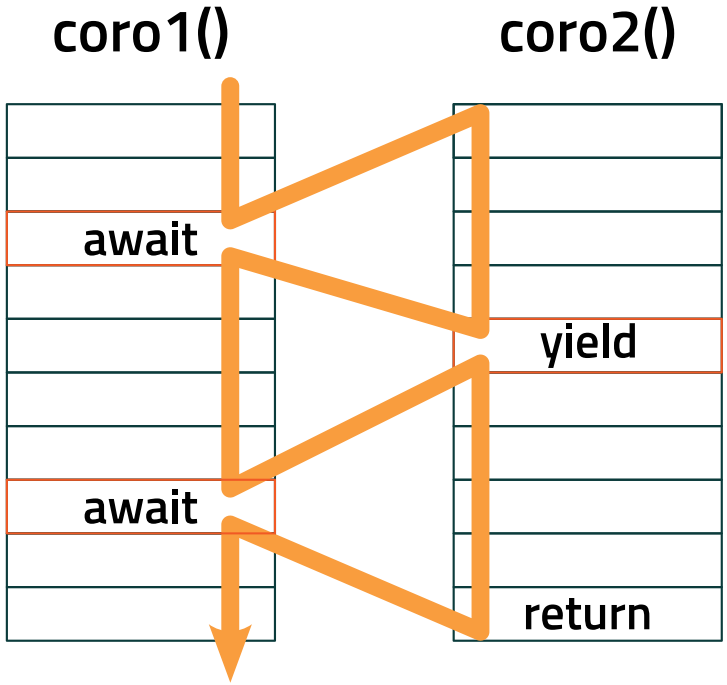


Coroutine

- Il concetto di coroutine è una generalizzazione che introduce nuovi tipi di punti di sospensione e riavvio senza restrizioni
- Questo permette al flusso di esecuzione di essere trasferito liberamente da una coroutine a qualsiasi altra, senza rispettare il vincolo di relazione tra routine principale e subroutine
- Questa generalità apre ad un ventaglio enorme di scenari applicativi

Scenario #1: generatori

- Una coroutine può sospendersi per riavviare il suo stesso chiamante
- Ogni punto di riavvio può essere una occasione di scambio dati tra chiamante e chiamato
- Consente l'implementazione di generatori lazy



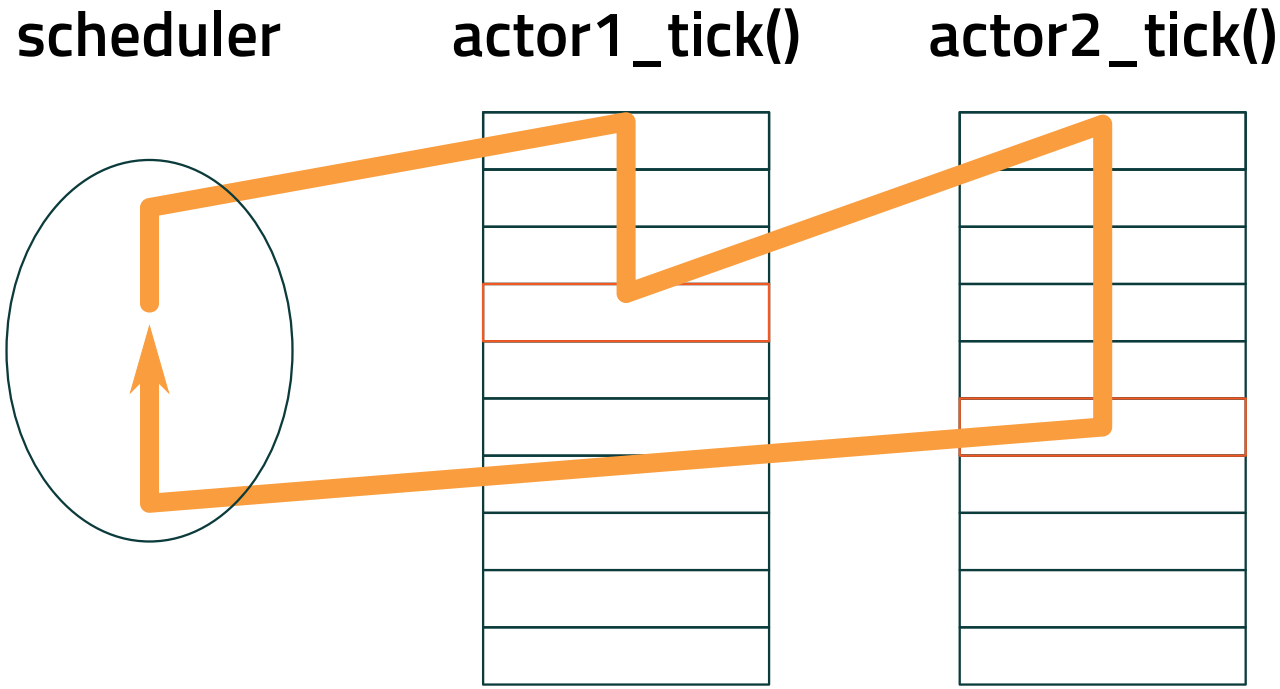
Scenario #2: multi-tasking cooperativo

- Nei videogiochi, è normale dover aggiornare continuamente un numero elevato di attori (personaggi, oggetti di gioco, VFX, suoni, ecc.)
- Per motivi di scalabilità e di performance, di solito si preferisce evitare le soluzioni multi-thread e effettuare tutti gli aggiornamenti in un unico thread
- Tradizionalmente, ad ogni tick di simulazione, su ciascun attore viene chiamata una funzione (ad es. Tick()) che effettua tutte le operazioni di aggiornamento necessarie per il tick corrente

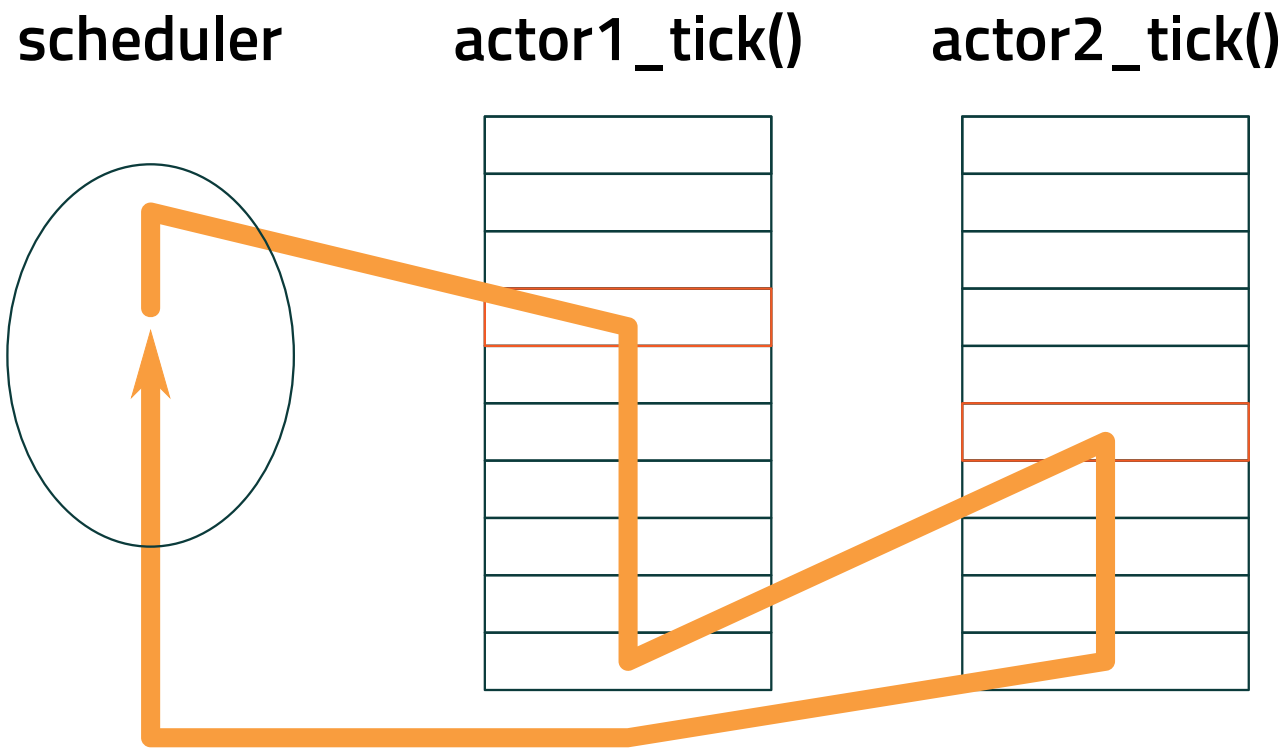
Scenario #2: multi-tasking cooperativo

- Il fatto che Tick() sia una funzione è però una limitazione, in quanto obbliga ciascun attore a completare l'intera operazione in un unico tick
- Le coroutine possono essere utilizzate per implementare operazioni di aggiornamento che richiedono più tick per essere completate, oppure che richiedono collaborazione asincrona tra attori differenti
- Ogni punto di sospensione può corrispondere all'attesa di un evento, come un timeout o il tick successivo, o al passaggio di informazione tra attori

Attori: primo tick



Attori: secondo tick

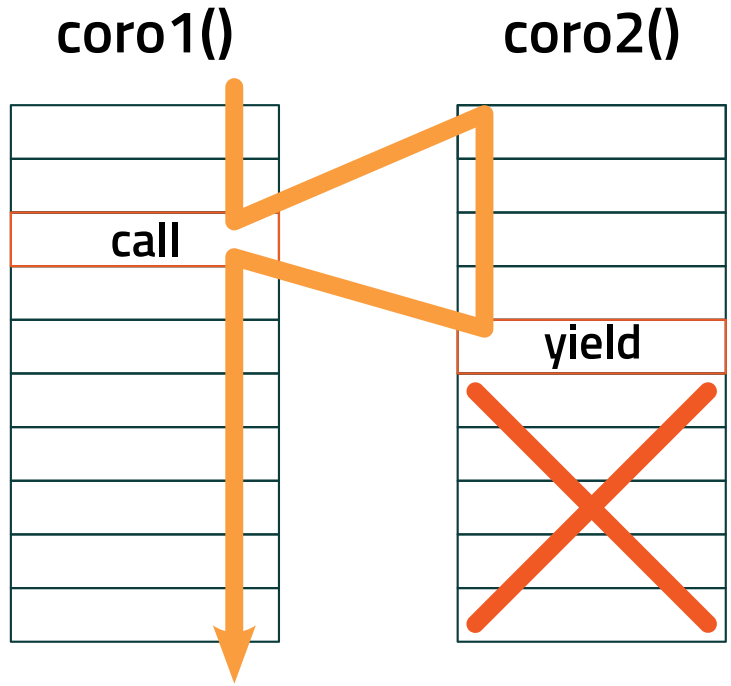


Scenario #3: chiamate asincrone

- Scrivere del codice che sfrutta chiamate asincrone (tipicamente per operazioni di I/O) è spesso laborioso
- Ci sono diverse tecniche: callback, future, continuazioni, ecc.
- L'utilizzo delle coroutine rende più semplice scrivere il codice in queste situazioni e a volte consente di evitare primitive di sincronizzazione, migliorando in alcuni casi le performance di codice critico

Scenario #4: abbandonare l'esecuzione

- Una coroutine sospesa ad un punto di esecuzione può essere abbandonata
- Questa tecnica può essere utilizzata per annullare operazioni asincrone o semplificare il codice di alcune pattern di programmazione funzionale



Contesto di esecuzione

- Al fine di poter riavviare una routine, dobbiamo essere in grado di salvare il contesto di esecuzione, ovvero lo stato della CPU e delle variabili locali, al momento della sospensione
- Nel caso di una chiamata a subroutine, tale "salvataggio" avviene in maniera naturale grazie all'utilizzo dello stack
- Possiamo usare lo stack perché è garantito che al momento del riavvio la subroutine è completata e quindi il suo stato non è più necessario

Stackful o stackless?

- Per le coroutine non è così semplice, perché esse possono essere sospese e riavviate in ordine arbitrario e spesso prima che abbiano terminato il loro compito
- Ci sono vari approcci implementativi al problema, che si distinguono in due grosse categorie: *stackful* e *stackless* a seconda che il contesto sia salvato sullo stack o da qualche altra parte
- Ciascuno dei due approcci ha pregi e difetti che sarebbe troppo lungo affrontare in questa sede

Coroutine e C++:
passato, presente e futuro

Coroutine e C++: a love story

- La prima proposta formale di introdurre le coroutine stackless in C++ viene presentata nel gennaio 2012, si parla ancora di “resumable functions”
- Nel febbraio del 2013 viene pubblicata la libreria Boost.Coroutine, una implementazione library-only di coroutine stackful e il mese successivo la libreria viene proposta formalmente per l’inserimento nella libreria standard
- Successivamente, vengono presentate molte altre proposte, a dimostrazione della complessità del problema e della varietà di approcci ad esso

Enter Gor

- Nell'ottobre 2014, Gor Nishanov (Microsoft) presenta una revisione delle "resumable functions", approfondendo dettagli che erano solo abbozzati
- La proposta di Gor Nishanov inizia ad attirare consensi, anche grazie alla realizzazione di una implementazione utilizzabile e funzionante in Visual Studio 2015
- Come è prassi dal C++14 in poi, la commissione decide quindi di procedere con la stesura di un TS (Technical Specification) dedicato alle coroutine

I am disappointed that stackless coroutines are being put into a TS rather than directly into the standard itself. I think they are ready and important for a few critical use cases (pipelines and generators).

– Bjarne Stroustrup, 2016

Coroutines TS

- Il Coroutines TS viene redatto e pubblicato nel luglio 2017, quindi troppo tardi per l'inserimento in C++17
- Nel frattempo Gor Nishanov lavora anche alla implementazione del TS in Clang 5, dimostrando che la specifica è implementabile anche su ABI differenti da quelle supportate da Visual Studio
- A questo punto, sembra che la strada del TS sia tutta in discesa e che quindi l'integrazione in C++20 sia scontata…

Jacksonville e Rapperswil

- Gli ultimi due incontri della C++ Committee si sono tenuti a Jacksonville (Marzo) e Rapperswil (Giugno)
- Per due volte la mozione di integrare il TS nella bozza di C++20 viene messa ai voti, ma, nonostante un largo supporto, non ci sono sufficienti consensi
- Tra i motivi, vi è la forte obiezione da parte di un team di Google che ha presentato inizialmente delle semplici critiche e successivamente una nuova controproposta completamente differente (Core Coroutines)

std::future<Coroutine>

- Il prossimo incontro della C++ Committee sarà a San Diego, nel novembre di quest'anno, dopodiché la bozza di C++20 entrerà in "feature freeze", quindi c'è ancora una sola, ultima, chance di avere le coroutine in C++20
- Si sta lavorando per apportare modifiche minori al TS, al fine di risolvere alcuni punti critici che sono stati evidenziati
- Nel caso peggiore, lo scenario probabile è la pubblicazione di un nuovo Coroutines TS v2 e lo slittamento a C++23

Coroutines TS

Un caso di successo: ciclo for su range

- In C++11 è stata introdotta la possibilità di scrivere un ciclo for così:

```
for (tipo valore : range)
{
    /* corpo */
}
```

- La specifica di questa istruzione è fornita presentando un frammento di pseudo-codice e dei punti di estensione

Pseudo-codice di un ciclo for

```
for (range-declaration : range-init)
    corpo
```

```
auto&& __range = range-init;
auto __begin = begin(__range);
auto __end = end(__range);
for (; __begin != __end; ++__begin )
{
    range-declaration = *__begin;
    corpo
}
```

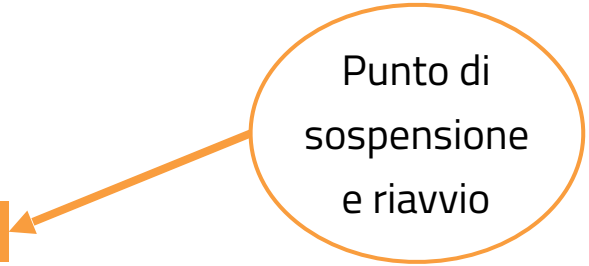
Punti di estensione

Punti di estensione del Coroutines TS

- Al fine di ottenere la massima flessibilità e apertura verso possibili estensioni da parte dell'utente, le specifiche del TS sono espresse con tecnica descrittiva analoga a quella del ciclo for
- Il numero di punti di estensione del TS è considerevole, siamo di fronte alla bellezza di 15 punti, che potrebbero diventare 18 se verrà dato seguito ad alcune critiche
- Questa scelta ardita concentra tutta la complessità sul lato della libreria, lasciando il codice utente relativamente semplice

Esempio #1: generatore

```
std::generator<int> gen_fibonacci()
{
    int i0 = 1;
    int i1 = 1;
    for (;;)
    {
        co_yield i0;
        int i2 = i0 + i1;
        i0 = i1;
        i1 = i2;
    }
}
```



Esempio #2: lettura asincrona da rete

```
task<vector<byte>> read_async(Tcp::addr_t address)
{
    byte buf[4096];
    vector<byte> result;
    auto connection = co_await Tcp::Connect(address);
    for (;;)
    {
        auto bytesRead = co_await connection.read(buf, sizeof(buf));
        if (bytesRead == 0)
            break;
        result.insert(result.end(), buf, buf + bytesRead);
    }
    co_return result;
}
```

Definizione di C++ coroutine

- Nell'attuale formulazione del TS, una coroutine è una funzione nel cui corpo viene utilizzata una delle seguenti keyword:
  - `co_await`
  - `co_yield`
  - `co_return`
- Dal punto di vista del chiamante, che potrebbe vedere solo la dichiarazione, una coroutine è una funzione come tutte le altre!

Promessa

- Il punto di partenza da cui si dipana tutto il meccanismo del TS è il cosiddetto oggetto "promessa" (promise)
- Il nome è stato scelto perché la sua funzione ricorda molto il tipo `std::promise` della libreria standard <future>
- Prima di tutto il compilatore deve dedurre il tipo della promessa e per fare ciò prende in esame la firma della coroutine

Dalla firma alla promessa

- Ad esempio, se la dichiarazione è:

```
my_coroutine coro(my_data data);
```

- Il tipo della promessa è

```
std::coroutine_traits<my_coroutine, my_data>::promise_type
```

std::coroutine_traits

- Se R, P1, ..., Pn sono, rispettivamente i tipi del valore di ritorno e di tutti i parametri della coroutine, allora il tipo della promessa è:

```
std::coroutine_traits<R, P1, ..., Pn>::promise_type
```

- Il primo punto di estensione consiste quindi nello specializzare questo template
- Nella maggioranza dei casi, il tipo della promessa è determinato unicamente dal tipo del valore di ritorno della coroutine

Importanza della promessa

- Il tipo della promessa fornisce l'implementazione per molti altri punti di estensione, in particolare fornisce:
  - La modalità di allocazione del contesto di esecuzione
  - La presenza di punti di sospensione aggiuntivi prima e dopo il corpo della funzione
  - La creazione dell'oggetto risultante dall'invocazione della coroutine
  - Le modalità per ritornare valori al chiamante
  - La gestione delle eccezioni non gestite dal corpo della coroutine

Allocazione del contesto

- Poiché il Coroutines TS è una implementazione stackless, al fine di consentire il caso generale, il contesto non viene memorizzato sullo stack
- Per default, viene usato l'operatore `new` per allocare memoria sullo heap, però la promessa ha l'opportunità di fornire una funzione di allocazione alternativa
- Questo automatismo è oggetto di critica, perché in alcuni casi (ad es. i generatori) l'allocazione sullo stack sarebbe possibile e preferibile
- Alcuni casi sono coperti dalla Heap Allocation eLision Optimization (HALO), ma, come tutte le ottimizzazioni, il meccanismo non è sotto il controllo dell'utente

Invocare una coroutine

- In base alla scelta, effettuata a compile-time, del tipo della promessa, all'invocazione di una coroutine accade questo:
  - Viene allocata memoria per il contesto di esecuzione, all'interno della quale viene costruita un'istanza della promessa
  - La promessa produce il valore di ritorno della coroutine, che verrà fornito al chiamante
  - Viene avviata la coroutine fino al primo punto di sospensione
- Tramite l'oggetto ritornato, il chiamante potrà controllare l'esecuzione della coroutine stessa e/o ottenere valori da questa

Punti di sospensione e riavvio

```
co_await expr
co_yield expr
```

- I punti di sospensione e riavvio sono inseriti in corrispondenza della valutazione di una *await-expression* o una *yield-expression*
- La seconda è definita in termini della prima, quindi ci limitiamo qui a vedere brevemente come è implementata una await-expression

`co_await expr`

- L'operazione di valutazione di una await-expression avviene in due tempi
- La prima cosa da fare è ottenere un oggetto *awaitable*
  - Viene valutata l'espressione `expr`
  - Viene data alla promessa la possibilità di trasformare il valore in un tipo differente, chiamando la funzione `p.await_transform`, se dichiarata
  - Viene dato al valore la possibilità di cambiare tipo, chiamando un overload di `operator co_await`, se disponibile

Oggetti awaitable

- Gli oggetti awaitable sono anch'essi fonte importante di punti di estensione, in quando sono loro che implementano la politica di sospensione e riavvio
- Un oggetto awaitable `e` è tenuto a fornire le seguenti operazioni:
  - `e.await_ready()`
  - `e.await_suspend(h)`
  - `e.await_resume()`

`co_await expr`: sospensione e riavvio

```
if (e.await_ready() == false)
{
    // Sospendi questa coroutine e selezionane
    // un'altra coroutine da riavviare
    coroutine_handle<> hNew = e.await_suspend(hCurrent);
    hNew.resume();
  ------------------------------------------------------
    // Quando questa coroutine sarà riavviata,
    // l'esecuzione riprenderà da qui
}
result = e.await_resume(); // valore della await-expression
```

std::coroutine_handle<>

- `std::coroutine_handle<>` è un template di libreria per il controllo delle coroutine a basso livello
- Tramite la tecnica del type-erasing, consente di trattare indifferentemente qualsiasi coroutine
- Consente di riavviare (resume) o abbandonare (destroy) la coroutine controllata

Un esempio della versatilità del TS

```
using std::optional;
using std::string;

optional<size_t> length(const optional<string>& opt_str)
{
    const string& str = co_await opt_str;
    co_return str.length();
}
```

Riassumendo

- Il Coroutines TS è complicato a causa dell'elevato numero di punti di estensione, resi necessari per far fronte alla naturale complessità el problema e dei vincoli di design
- La complessità è però concentrata sul lato della libreria, è possibile utilizzare le coroutine senza entrare nei dettagli del macchinario sottostante
- Il TS è solido e versatile e ha potenziale per l'utilizzo in molte applicazioni
- Il TS non copre tutti i casi d'uso, ma in sei anni non si è vista una proposta riesca ad accontentare tutti, forse non è possibile e forse è meglio così

Riassumendo

- Ci sono due implementazioni conformi al TS utilizzabili oggi stesso
- Ci sono ancora barlumi di speranza di avere il TS integrato in C++20, ma non è affatto scontato, dovremo attendere novembre per la decisione finale
- Se le avremo in C++20, la formulazione finale potrebbe cambiare, ma sarà sostanzialmente quella del TS
- Se invece sarà tutto rimandato a C++23, ci sarà probabilmente un nuovo TS e tutto può succedere

Grazie dell’attenzione

Thanks to the sponsors!

Italian C++ Conference 2018 – June 23, Milan #itcppcon18