C++14

Novità del nuovo standard

Nicola Gigante

C++ User Group Udine

19 dicembre 2014

Di cosa parleremo

- Novità introdotte dal nuovo standard
  - nel linguaggio
  - nella libreria standard
  - nel processo di evoluzione dello standard stesso
- `std::future`
  - Oltre il C++14
  - Library Foundamentals Technical Specification v1
  - Stato dei lavori per il C++17

Cos'è il C++14

International Standard ISO/IEC 14882:2014(E)
Programming Language C++

Approvato il 18 Agosto, è una minor revision del C++11.

- Consolidamento
- Correzioni
- Alcune nuove funzionalità importanti che non erano entrate a far parte del C++11 per motivi "burocratici"

Supporto dai compilatori

La novità più importante di questo standard è il tasso di conformità dei principali compilatori:

- C++14 è la prima versione dello standard ad aver avuto un compilatore feature-complete *dal giorno zero* (`clang`).
- Ora supportato totalmente anche da GCC 5 (parzialmente anche dal 4.9)
- Molte feature disponibili anche in Visual C++ 2015

Risultato:

- Se state già scrivendo codice C++11, approfittare delle novità della nuova versione è molto probabilmente già una buona idea.

Novità divise per area

Programmazione generica e metaprogramming:

- Alias type traits
- `std::integer_sequence` e correlati
- Lookup *per tipo* in `std::tuple`
- Variable templates

Programmazione funzionale:

- Lambda functions polimorfe
- Automatic function return type deduction

Altro:

- `std::make_unique`
- Relaxed `constexpr` functions
- `std::shared_timed_mutex`
- User-defined literals, binary literals, digit separators...

Di cosa parleremo oggi

Le mie feature preferite:

- `std::make_unique`
- Generalized `constexpr` functions

Smart pointers

C++11 ha introdotto due smart pointer:

- `std::unique_ptr`, per ownership esclusiva
- `std::shared_ptr`, per ownership condivisa

std::make_shared

`std::make_shared` crea uno shared pointer senza esplicitare la chiamata all'operatore new:

```
std::shared_ptr<Type> ptr { new Type{arg1, arg2, arg3} };
```

```
auto ptr = std::make_shared<Type>(arg1, arg2, arg3);
```

Vantaggi:

- Meno ripetizioni
- Performance

std::make_unique

In C++11 non hanno ritenuto necessario includere anche `std::make_unique`, l'equivalente per `std::unique_ptr`:

- Nessun vantaggio in termini di performance
- Banale da implementare

```
template<typename T, typename ...Args>
std::unique_ptr<T> make_unique(Args&& ...args) {
  return std::unique_ptr<T>{new T{std::forward<T>(args)...}};
}
```

Quindi perchè introdurlo nel C++14?

Exception safety

Questo codice è 100% corretto?

```
void func(std::unique_ptr<T>, std::unique_ptr<T>);

func(std::unique_ptr<T>(new T), std::unique_ptr<T>(new T));
```

Exception safety

Questo codice è 100% corretto? No.

```
void func(std::unique_ptr<T>, std::unique_ptr<T>);

func(std::unique_ptr<T>(new T), std::unique_ptr<T>(new T));
```

Se il costruttore di T lancia un'eccezione, si verifica un memory leak.

Soluzione

Soluzione corretta al 100%

```
void func(std::unique_ptr<T>, std::unique_ptr<T>);

func(std::make_unique<T>(), std::make_unique<T>());
```

Tutto qui?

I dare you say new/delete again, I double dare you

L'importanza dell'aggiunta di `std::make_unique` ha a che fare con uno dei principi più importanti del Modern C++:

> *Never use new and delete.*

Ovvero:

- Scegliere la politica di ownership degli oggetti come importante punto di design dell'intero programma.
- Usare gli smart pointer per forzare la politica di ownership scelta.
- Usare `std::make_shared` e `std::make_unique` per allocare e costruire l'oggetto e contemporaneamente incapsularlo nell'handler.

Generalized constant expressions

C++11 ha introdotto il concetto di funzioni e oggetti *constexpr*.

- Un oggetto `constexpr` è garantito essere inizializzato in modo statico.
- Una funzione `constexpr` può essere valutata *dal compilatore* quando gli argomenti sono *constant expressions*.

```
constexpr int func(int x) { ... }

constexpr int value = func(42);
```

Restrizioni

Dovendo essere valutate dal compilatore, le funzioni `constexpr` devono sottostare a parecchie restrizioni:

- Una singola istruzione `return`
- Nessun side-effect esterno: consentite solo chiamate ad altre funzioni `constexpr`
- Nella pratica: deve essere codice funzionale puro.

Sono restrizioni molto forti:

- Implementando questo sistema, ci si è accorti che queste restrizioni sono inutilmente forti.
- Danno vita a codice complicato, che risulta inefficiente quando eseguito a runtime.

C++14 relaxed constexpr

Il C++14 rilassa molte delle restrizioni su cosa può fare una funzione `constexpr` al suo interno.
Non ci devono essere side-effects esterni, però vengono consentite:

- La dichiarazione di variabili e la mutazione di oggetti la cui vita termina con la funzione stessa (quindi variabili locali).
- Control flow arbitrario: `if`, `switch`, `for`, `while`, ecc...

Sono ancora vietate le eccezioni e l'allocazione dinamica di memoria, e tutto ciò che richieda di "ispezionare" la rappresentazione interna dei valori (es. cast di un puntatore ad un intero).

Esempio

Verso il C++17

Una delle novità più interessanti del nuovo standard è stata *il processo con il quale è stato sviluppato*

- Rapporto più trasparente tra il comitato e la community
- Ciclo di feedback più stretto tra il comitato e gli implementatori
- Modularità delle proposte:
  istituzione delle Technical Specifications

Verso il C++17

Il C++17 sarà la prossima major release del linguaggio:

- Concepts: generic programming for the rest of us
- Riscrittura della STL sul concetto di *Range*
- Più librerie standard di supporto
  networking, filesystem, grafica 2D, ecc...
- Migliore supporto per concorrenza e parallelismo
  resumable functions, parallel algorithms, ecc...

Perchè parlarne ora?

Per evitare i ritardi del C++11, il comitato ha *modularizzato* il proprio lavoro.

- Attività divise tra *Working Groups*, il cui lavoro viene pubblicato separatamente a cadenza regolare in documenti chiamati *Technical Specifications*
- Le prime versioni delle TS pubblicate dai vari WG sono già state pubblicate. Alcune sono già state implementate!

Library Foundamentals TS

- Contiene aggiunte e modifiche di varia natura alla libreria standard
- Alcune di queste erano previste per l'inclusione nel C++14, ma sono slittate per vari motivi.
- Alcune di queste sono molto importanti:
  - `std::optional<T>`
  - `std::string_view`
  - `std::array_view`

Optional values

`std::optional<T>` rappresenta un oggetto di tipo `T` che *potrebbe non esistere*.

- Simile a `Maybe` di Haskell
- Risolve il problema del *missing return value*

Missing return value

Cosa possono restituire delle funzioni non totali ma suriettive?

```
int div(int x, int y); // What if y == 0?

int parseInt(std::string s); // What if s is not a number?

std::string
getline(std::istream&); // What if there's no data?
```

Il problema si pone specialmente quando *non c'è* un valore di output non valido da restituire in caso di errore (es `-1` o `NULL`).

Soluzioni (parziali)

- Flag esterne:
  Problemi: non componibile, non thread-safe, può essere ignorato, va documentato
- Eccezioni:
  Problemi: non componibile, poco manutenibile, può essere ignorato, va documentato, poco performante
- `std::pair<T, bool>`
  Problema: che succede se T è costoso da costruire e/o copiare, o non possiede un default constructor?

Soluzione

```
std::optional<int> parseInt(std::string);
```

- Se la stringa non contiene un numero, il risultato è `std::nullopt`.
- Il chiamante deve esplicitamente estrarre il valore, ed esplicitamente *scegliere* di ignorare la possibilità di un errore.

Soluzione

```
std::string input = "banana?";
auto result = parseInt(input);

if(result) {
  std::cout << *result << "\n";
} else {
  std::cout << input << "␣is␣not␣a␣number\n";
}
```

Soluzione

```
std::string input = "42";
auto result = parseInt(input);

// Throw an exception if something goes wrong
std::cout << "I'm sure the string parsed as: " << *result;
```

Perchè parlarne ora?

Incluso nella Library Foundamentals TS v1, ma *già implementato*:

- LLVM `libc++`, quindi Mac OS X e iOS, FreeBSD e Linux
- GCC `libstdc++`, quindi GNU/Linux e Unix di ogni tipo
- Boost.Optional, quindi *everywhere*
- Implementazione stand-alone di riferimento su `https://github.com/akrzemi1/Optional`
  Singolo header da includere direttamente nel proprio codice

Conclusioni

Nonostante sia una minor release, il C++14 mette della carne sul fuoco.

- Ancora più facile seguire le best-practice del Modern C++
- Disponibile da subito, già implementato dai compilatori principali
- Primo risultato della riorganizzazione del Committee

Domande?