Accesso ai dati in C++ moderno

Correttezza, sicurezza e altre cose da nulla

Nicola Gigante

C++ User Group Udine

Italian C++ Meetup - Pordenone - 7 febbraio 2015

Chi siamo

C++ User Group Udine

- Nato alla fine dell'anno scorso
- Scopo: fornire un punto di ritrovo per chi usa/studia/si interessa di C++ nel territorio di Udine e dintorni
- Costola di AsCI - Associazione Cultura Informatica
- Sito web e mailing list:
  `http://cpp.ud.it`
- Incontri (quasi) periodici

Di cosa parleremo

Interfacciarsi ai database SQL in modo *moderno*:

- Maggiore produttività
- Maggiore sicurezza

Come?

- Type-safety
- Lightweight abstractions
- EDSLs

Di cosa parleremo

Due case studies:

- SQL++11: Layer di interfaccia diretta con gli RDBMS
  - Attuale stato dell'arte
  - Interrogazioni typesafe e sicure
  - Sfrutta fino all'osso le nuove caratteristiche del C++11
  - In attivo sviluppo
- ODB: Object-Relational Mapper moderno e typesafe
  - Mapping tra classi C++ e relazioni SQL
  - Maturo, robusto e con molto supporto
  - Feature-packed

Impedance mismatch

- Il C++ è un linguaggio fortemente tipato, con un type system potente e flessibile.
- L'SQL è un linguaggio fortemente tipato, molto espressivo nel descrivere l'interazione tra componenti diverse dei dati.
- Perchè l'interazione tra i due linguaggi avviene attraverso delle *stringhe*?

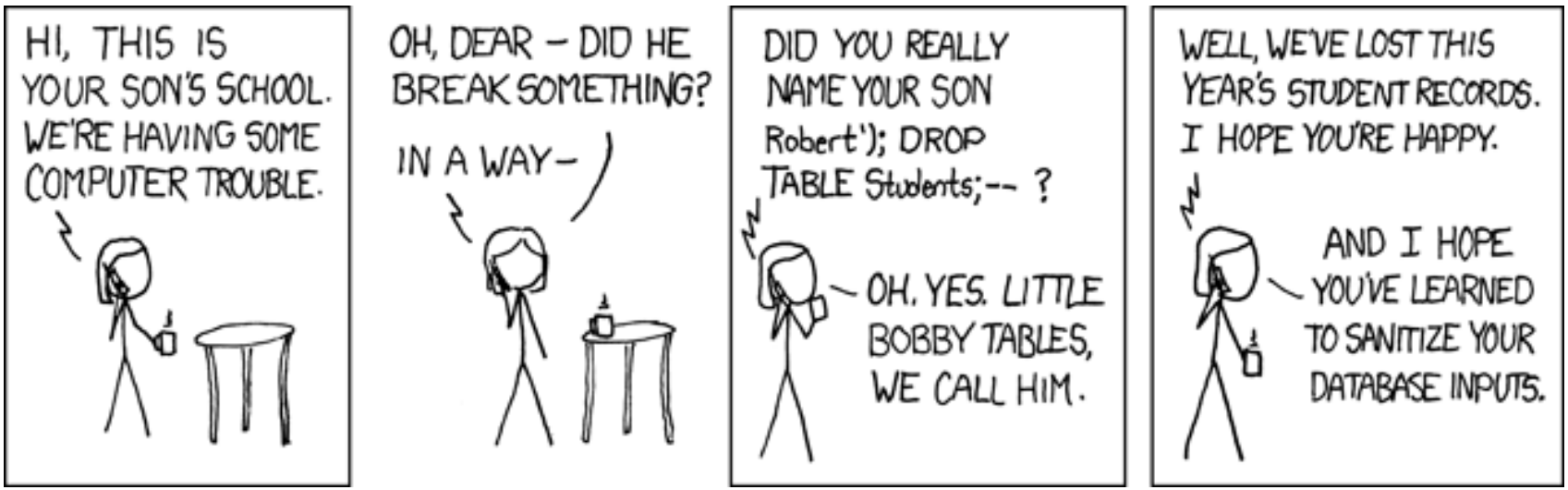
Impedance mismatch

Tipica API per l'uso di database SQL:

```
QSqlQuery query;
query.prepare("SELECT name, surname FROM person"
              "WHERE name = ? AND age < ?");
query.bindValue(0, 42);
query.bindValue(1, "Bartosz");
query.exec();

while (query.next()) {
  QString name = query.value(0).toString();
  QString surname = query.value(1).toString();
  std::cout << name << "\n" << surname << "\n";
}
```

Cosa potrebbe mai andare storto?



Cosa potrebbe mai andare storto?

- La query viene analizzata dal server, nel momento in cui viene eseguita (o preparata)
- Il compilatore non sa nulla del server, della struttura delle relazioni SQL, e nemmeno della *sintassi* della query SQL
- L'RDBMS non sa nulla dei tipi di dato usati nel codice C++
- Qualsiasi errore viene riportato durante il debug...
- ... o peggio, in produzione

Cosa potrebbe mai andare storto?

Ciò causa problemi di:

- Sicurezza: SQL injection, e più in generale cross site scripting, privilege escalations (qualcuno ricorda shellshock?)
- Produttività: un errore su un sistema in produzione costa enormemente di più di uno localizzato durante lo sviluppo, sia in termini di tempo che di denaro.

Soluzione: farsi aiutare dal typesystem

- Il type system del C++ è molto flessibile e potente
- Linea guida: convertire errori nella *logica* del codice, in errori di *sintassi*
- Fare in modo che codice non corretto non possa neanche essere compilato
- Vantaggi: correttezza, sicurezza, facilità di refactoring, ecc...

Embedded Domain Specific Languages

- Il C++ offre a chi scrive librerie la possibilità di integrarsi molto a fondo nel linguaggio.
- Tanto che alcune librerie sembrano *estenderlo*.
- Un DSL è un linguaggio specifico per il dominio applicativo di ciò che si sta sviluppando (es. SQL, DDL, XML, ecc...)
- Un Embedded DSL non è un linguaggio separato, ma è realizzato "estendendo" il linguaggio host (es. il C++).
- La flessibilità nel realizzare EDSL è uno dei fattori del successo dei linguaggi funzionali moderni (es. Haskell).

SQL++11

Un EDSL per l'accesso a database SQL:

- Sintassi simile a SQL ma integrata nel C++
- La libreria deve *conoscere* la struttura delle relazioni SQL
- Mapping automatico tra i tipi C++ e i tipi di SQL
- Gli errori nelle query si traducono in *errori di compilazione*
- Sfrutta fino all'osso il C++11: variadic templates, constexpr, decltype, ecc... Clang e GCC no problem

Un primo esempio

```
// Dichiaro le tabelle a cui accedero'
Person person;

// Costruisco la query
auto query =
  select(person.name, person.surname)
  .from(person)
  .where(person.age < 42 and person.name == "Bartosz");

// Connessione al database
mysql::database db{/* setup connessione */};

// Eseguo e itero i risultati
for(const auto&row : db(query)) {
  std::string name    = row.name;
  std::string surname = row.surname;
  std::cout << name << "\n" << surname << "\n";
}
```

Un primo esempio

```
auto query =
  select(person.name, person.surname)
  .from(person)
  .where(person.age < 42 and person.name == "Bartosz");
```

- Tabelle e campi sono riferiti *per nome*, non tramite una stringa
- Il passaggio di parametri alla query è implicito

Un primo esempio

```
for(const auto&row : db(query)) {
  std::string name    = row.name;
  std::string surname = row.surname;
  std::cout << name << "\n" << surname << "\n";
}
```

- Il risultato della query è un range iterabile
- Il nome dei campi del risultato è riferito per nome, non per posizione
- Il tipo dei campi è noto in fase di compilazione

Un esempio più complesso

```
auto query =
  select(player.name.as(name),
         player.coins.as(coins),
         kart.kart_model.as(k))
  .from(player).join(kart).on(person.kart == k)
  .group_by(coins);
```

- Join arbitrari
- Espressioni arbitrarie nelle clausole `where`.
- Praticamente tutta la sintassi dell'SQL è disponibile (subquery, funzioni aggregate, ecc...)

Dynamic queries

- Cosa faccio se devo scegliere, ad esempio, il nome di un campo a runtime?
  - `dynamic_select`, `dynamic_where`, ecc...
- Si rinuncia al controllo a compile-time solo della singola parte della query che va scelta a runtime.

Com'è possibile?

Expression Templates:

- Ogni funzione e operatore non esegue la propria operazione, ma restituisce un oggetto che la rappresenta
- A tutti gli effetti si costruisce a *compile-time* un albero sintattico dell'espressione
- L'espressione viene valutata dal backend dello specifico motore (MySQL, PostgreSQL, ecc...) per creare realmente la query.
- L'albero sintattico è riflesso sia nel valore che nel *tipo* degli oggetti.

Com'è possibile?

Ovviamente dobbiamo avere in scope i nomi di tabelle e colonne, con rispettivi tipi

- La libreria ha bisogno di sapere la struttura delle relazioni del database
- Si dichiara un *trait* per ogni tabella.
- Viene fornito un tool per generare i tipi a partire dagli statement SQL

Backends

Per ogni RDBMS specifico si usa un backend differente

- Il backend ha il compito di generare la query secondo la sintassi e le peculiarità del sistema specifico.
- Definisce il mapping dei tipi SQL con i tipi di basi del C++
- Può *ottimizzare* la query a piacimento
- Backend per i più popolari motori open source: MySQL, PostgreSQL, SQLite
- Non solo SQL: backend sperimentale per container STL (Linq anyone?)

Backends

Ogni backend definisce cosa è valido e cosa no:

- Tipi
- Sintassi, restrizioni, ecc...
- Costrutti aggiuntivi non standard

Risultato:

- Refactoring: Incompatibilità causate dal cambio del backend risultano in errori di *compilazione*
- Flessibilità: Non si è limitati al minimo comune denominatore

Dove si sta andando

SQL++11 è ancora in pieno sviluppo

- Semplificare il mapping tra tabelle SQL e tipi
- Aggiungere backends
- Aggiungere features mancanti

Riferimenti

- http://github.com/rbock/sqlpp11
- Talk @ Meeting C++ 2014:
  http://meetingcpp.com/index.php/mcpp2014.html

ODB

Framework di Object-Relational Mapping per C++

- Mapping tra classi C++ e SQL
- Concetto di classe *persistente*
- Pragma per annotare la dichiarazione della classe
- Fase di preprocessing per generare il codice di supporto

Differenze con SQL++11

- È un ORM, non solo un layer di accesso a SQL
- Framework maturo, robusto, commercialmente supportato
- Utilizzabile in modalità C++98 e C++11
- Stessi vantaggi: correttezza, sicurezza, manutenibilità
- Feature più avanzate:
  - Transazioni
  - Migrazione di schema

Esempio

```
struct person {
  // ...
  std::string email;

  std::string name;
  int age;
};
```

Esempio

```
#pragma db object
struct person {
  // ...
#pragma db id
  std::string email;

  std::string name;
  int age;
};
```

Esempio

```
using query_t = odb::query<person>;
mysql::connection db{/* setup connessione */};

for(person&p : db.query<person>(query_t::age < 30))
{
  std::cout << p.name;
}
```

- La sintassi dell'EDSL è un po' meno immediata (per via della compatibilità con il C++98)
- Corrispondenza con SQL comunque molto alta
- Come sempre, type safety garantita

Update

```
person john {"john@barmes.org", "John", 42};
john.age++;

db.update(john);
```

Mapping dei tipi

- Il mapping dei tipi SQL con i tipi C++ è molto flessibile in ODB
- Tipi di base, array, blobs, ecc..
- Relazioni tra tabelle mappate in `shared_ptr` ad altri oggetti.
- Mapping con i container STL
- Bridge per Qt e Boost
  Es. È possibile mappare una colonna di tipo `ARRAY` in un `QVector` invece di un `std::vector`
- Valore `NULL` mappato in `odb::nullable` o `boost::optional`

Riferimenti

http://codesynthesis.com

Per finire

Il C++ moderno può fornire dei vantaggi significativi:

- Facilità d'uso
- Type safety
- Type safety
- Ah... type safety
- Facciamoci aiutare dal type system nel nostro lavoro di manutenzione, refactoring, debugging, ecc...

Domande?