Template Metaprogramming: sfatiamo i miti

Nicola Gigante

C++ User Group Udine

27 maggio 2015

Di cosa parleremo

1. Introduzione
   - Programmazione generica
   - Template Metaprogramming
2. Templates
   - Funzioni e classi template
   - Istanziazione
   - Specializzazioni
3. Template Metaprogramming
   - Hello world
   - Type traits
   - Case study: Boost.Units

Templates

I template sono il meccanismo con cui il C++ implementa il paradigma della *programmazione generica*.

- Scrivere codice indipendente dal tipo dei dati su cui opera
- Una delle funzionalità più importanti e distintive del C++

Programmazione generica

La programmazione generica, implementata tramite i template, è la caratteristica che rende il C++ un linguaggio che vale la pena considerare.
Permette di ottenere codice:

- Astratto, di alto livello,
- Type-safe (correttezza controllata tramite il type system),
- Veloce (zero runtime overhead) quanto codice C di basso livello (anzi, a volte di più)

Type-driven development

Il typesystem del C++ è particolarmente espressivo. Sfruttando il typesystem ci si può far aiutare dal compilatore ad eliminare svariati tipi di bug.

- Forzare il rispetto di invarianti
- Impedire la rappresentazione di stati non validi
- Impedire operazioni non valide

La Template Metaprogramming è un insieme di tecniche che permette di ottenere queste funzionalità.

Template Metaprogramming

Template metaprogramming significa, in poche parole:

- Sfruttare il meccanismo dei template per manipolare *i tipi*.
  - Arricchire i tipi di informazioni utili
  - Esaminare i tipi
  - Trasformare i tipi
- Generare il codice appropriato (di solito corretto per costruzione) a partire dai tipi.

Template Metaprogramming

Al termine Template Metaprogramming spesso si associano alcuni pensieri:

- È mostruosamente complicato fare queste cose
- “A me non servono cose così elaborate”

Template Metaprogramming

Al termine Template Metaprogramming spesso si associano alcuni pensieri:

- È mostruosamente complicato fare queste cose
  Capendo pochi concetti di base si possono fare molte cose
- "A me non servono cose così elaborate"
  I vantaggi sono potenzialmente utili a chiunque

Spiegando il funzionamento di queste tecniche, cercheremo di toccare questi punti.

Funzioni template

```
template<typename T>
T max(T v1, T v2) {
  return v1 > v2 ? v1 : v2;
}

// ...
int    x = max(42, 33);     // 42
double y = max(3.14, 2.71); // 3.14
```

Classi template

```
template<typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
};

// ...
pair<int, double> p{42, 3.14};

std::cout << p.first << ' ' << p.second << "\n";
```

Istanziazione

L'uso di un template implica la sua *istanziazione*.

```
template<typename T>
class vector { /* ... */ }; // Dichiarazione

vector<int> v; // Istanziazione
```

- Il compilatore *sostituisce* gli argomenti ai parametri in tutto il corpo della classe/funzione template.
- Una nuova classe/funzione viene così generata e viene compilata normalmente.

Istanziazione

Il parsing di un template avviene in due fasi:

- La prima fase avviene quando il compilatore incontra inizialmente la dichiarazione del template.
  - Avviene il parsing di tutti gli elementi sintattici e il typechecking di tutto ciò che non dipende dai parametri.
- La seconda fase avviene quando il template viene *istanziato*, una volta per ogni combinazione di parametri.
  - Si termina il typechecking di tutto il corpo.
  - Una parte di codice che dipende dal parametro di un template si dice “type-dependent”.

Specializzazioni

Una singolo tipo ottenuto istanziando il template si chiama *specializzazione* del template.
È possibile fornire esplicitamente l'implementazione di una specializzazione particolare:

```
template<typename T>
class vector {
  // generic implementation
};

template<>
class vector<bool> {
  // compact implementation
};
```

Non-type parameters

I template possono accettare anche parametri *non-tipo*.

```
template<typename T, int N>
class array {
  T data[N];
  // ...
};

array<int, 4> a = {1, 2, 3, 4};
```

Tipi annidati

In C++ un tipo si può dichiarare all'interno di qualsiasi scope, in particolare anche all'interno di una classe. È comune utilizzare un nested type per fornire un tipo correlato alla propria classe in modo generico.

```
template<typename T>
class vector {
public:
  using iterator = T *;
  // ...
};

vector<int>::iterator it; // tipo 'int*'
```

Tipi annidati

Un fastidio "grammaticale" va tenuto presente quando si utilizzano tipi annidati dentro ad un tipo *type-dependent*.

```
template<typename T>
void func()
{
  std::vector<int>::iterator it1; // ok

  std::vector<T>::iterator it2; // not valid
}
```

Tipi annidati

Un fastidio "grammaticale" va tenuto presente quando si utilizzano tipi annidati dentro ad un tipo *type-dependent*.

```
template<typename T>
void func()
{
  std::vector<int>::iterator it1; // ok

  typename std::vector<T>::iterator it2; // ok
}
```

Template Metaprogramming - Hello world

```
template<int N>
struct fact {
  static const int value = N * fact<N - 1>::value;
};

template<>
struct fact<0> {
  static const int value = 1;
};

// ...
std::cout << fact<4>::value; // stampa 24
```

Hello world

Un template può istanziare se stesso con parametri diversi (non c'è motivo per cui non lo possa fare). Ricorsione libera.

```
template<int N>
struct fact {
  static const int value = N * fact<N - 1>::value;
};
```

Hello world

Il meccanismo delle specializzazioni esplicite fornisce un'"istruzione di selezione".

```
template<>
struct fact<0> {
  static const int value = 1;
};
```

Type traits

Un type trait è una “funzione” a type-level.

- Uno o più parametri template (tipi o valori)
- Risultato calcolato a compile-time (un altro tipo o un altro valore)

Ottenere informazioni sui tipi

```
template<typename T1, typename T2>
struct is_same {
  static const bool value = false;
};

template<typename T>
struct is_same<T, T> {
  static const bool value = true;
};

std::cout << is_same<int,int>::value; // stampa true
std::cout << is_same<char,int>::value; // stampa false
```

Trasformare i tipi

```
template<typename T>
struct remove_pointer {
  using type = T;
};

template<typename T>
struct remove_pointer<T *> {
  using type = T;
}
```

Type traits library

La libreria standard fornisce una grande varietà di type traits di base.

- Classificazione dei tipi:
  `is_void`, `is_integral`, `is_floating_point`, `is_class`, `is_pointer`, `is_reference`, ecc...
- Informazioni sui tipi:
  `is_const`, `is_empty`, `is_abstract`, ecc...
- Informazioni sulle operazioni supportate:
  `is_default_constructible`, `is_copy_constructible`, `is_assignable`, ecc...
- Trasformazione dei tipi: `add_const`, `remove_const`, `make_unsigned`

Case study: Boost.Units

Nel 1999 la NASA ha perso alla deriva il Mars Climate Orbiter, una sonda da 643 milioni di dollari. Il motivo?

> *The primary cause of this discrepancy was that one piece of ground software supplied by Lockheed Martin produced results in a United States customary unit ("American") [...], while a second system, supplied by NASA, that used those results expected them to be in metric units [...].*

`http://en.wikipedia.org/wiki/Mars_Climate_Orbiter`

Boost.Units

Boost.Units è una libreria che permette di associare un'unità di misura a valori numerici.

- Impedire tramite il type-system la compilazione di operazioni non valide
- Forzare l'esecuzione delle giuste conversioni
- Tutti i controlli vengono fatti dal compilatore. Zero runtime overhead.

Esempio

```
#include <boost/units/......>

using namespace boost::units;
using namespace boost::units::si;

quantity<mass> m  = 1 * kilogram;
quantity<force> f = 1 * newton;

quantity<acceleration> a = f / m;

quantity<acceleration> a = f * m; // Compilation error
```

Boost.Units

Esempio perfetto di Type-Driven Development implementato con Template Metaprogramming.

- Si arricchiscono i tipi con informazioni usate solo a compile-time.
  Es. `quantity<mass>` invece di `double`.
- Le operazioni non valide sono intercettate dal compilatore.
  Es. `quantity<mass>()+quantity<time>()` non compila
- Si calcolano automaticamente nuovi tipi a partire dai tipi forniti.
  Es. Il tipo di `quantity<mass>()/quantity<time>()` è `quantity<velocity>`

Come funziona tutto ciò?

Scriveremo una versione molto semplificata dello stesso meccanismo:

- Dichiariamo alcuni tipi per rappresentare le varie unità di misura.
- Dichiariamo il tipo `quantity`, che dovrà essere un semplice wrapper intorno ad un `double`, ma il cui tipo tiene conto dell'unità di misura scelta.
- Dichiariamo dei type trait per calcolare l'unità di misura del risultato delle operazioni aritmetiche.
  (es. spazio/tempo $\rightarrow$ velocità)
- Dichiariamo le operazioni aritmetiche solo tra i tipi giusti per evitare operazioni non valide.

Demo

Code at
`http://tinyurl.com/nod2clv`