I template del C++

Enea Zaffanella

Metodologie di programmazione
Laurea triennale in Informatica
Università di Parma
enea.zaffanella@unipr.it

Template di funzione

- un template di funzione è un costrutto del linguaggio C++ che consente di scrivere uno **schema parametrico** per una funzione
- esempio:

```
// dichiarazione pura di un template di funzione
template <typename T>
T max(T a, T b);

// definizione di un template di funzione
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

Elementi di un template

- **nome del template**: nell'esempio precedente abbiamo prima dichiarato e poi definito un template di funzione di nome `max` (nota: al solito, la definizione è anche una dichiarazione)
- **parametri del template**: nell'esempio, `T` è un parametro di template
  - il parametro viene dichiarato essere un `typename` (cioè, il nome di un tipo) nella lista dei parametri del template
  - la parola chiave `typename` può essere sostituita da `class`, ma in ogni caso indica un qualunque tipo di dato, anche built-in (quindi meglio scrivere `typename`)
  - la lista dei parametri può contenerne più di uno, separati da virgole
  - oltre ai parametri `typename`, ne esistono di altre tipologie (valori, template)

Nomi dei parametri di template

- per convenzione (non è una regola del linguaggio), spesso si usano nomi maiuscoli per i parametri di tipo
- il nome `T` è comunque arbitrario (e può essere cambiato a piacere): è stato scelto, probabilmente, per indicare che si intende un tipo qualunque
- esempio:

```
void foo(int a);
void foo(int b); // (ri-) dichiara la stessa funzione foo

template <typename T>
T max(T a, T b);
template <typename U>
U max(U x, U y); // (ri-) dichiara lo stesso template max
```

- come nel caso dei normali parametri delle funzioni, anche per i parametri dei template è opportuno scegliere **nomi significativi**

Istanziazione di template

- dato un template di funzione, è possibile "generare" da esso una o più funzioni mediante il meccanismo di **istanziazione**
- l'istanziazione fornisce un **argomento** per ognuno dei **parametri** del template
- l'istanziazione avviene **a tempo di compilazione**
- l'istanziazione avviene spesso in maniera **implicita**, quando si fa riferimento al nome del template allo scopo di usarne una particolare istanza
- durante la compilazione (di una unità di traduzione), se si forniscono più volte gli **stessi argomenti** ad un template, l'istanziazione avverrà **solo la prima volta**
- se si forniscono argomenti distinti allo stesso template, si otterranno istanziazioni distinte

Esempi di istanziazione implicita

- nell'esempio seguente, il template `max` viene istanziato (implicitamente) due volte, usando le parentesi angolate per fornire (esplicitamente) l'argomento per il parametro del template

```
void foo(int i1, int i2, double d1, double d2) {
  // istanziazione della funzione
  //   int max<int>(int, int);
  int m1 = max<int>(i1, i2);

  // istanziazione della funzione
  //   double max<double>(double, double)
  double m2 = max<double>(d1, d2);
  // non c'è istanziazione (usa funzione già istanziata)
  double m3 = max<double>(d1+d2, d1-d2);
}
```

Deduzione degli argomenti per l'istanziazione

- quando si istanzia un **template di funzione**, solitamente si evita la sintassi esplicita per gli argomenti del template, lasciando al compilatore il compito di **dedurre** tali argomenti a partire dagli argomenti passati alla chiamata di funzione; esempio:

```
void foo(char c1, char c2) {
  // istanziazione della funzione
  //   char max<char>(char, char);
  int m = max(c1, c2);
}
```

- il legame `T = char` viene dedotto dal tipo degli argomenti (`c1` e `c2`) usati nella chiamata
- nota bene: il tipo di `m` (cioè `int`) **non** influisce sul processo di deduzione per `T`

Fallimento della deduzione

- il processo di deduzione potrebbe fallire a causa di ambiguità:

```
void foo(double d, int i) {
  int m1 = max(d, i); // errore di compilazione
  int m2 = max<int>(d, i); // ok: evito la deduzione
}
```

- nel primo tentativo di istanziazione il compilatore non riesce a dedurre un unico tipo `T` che sia coerente con `d` (`double`) e con `i` (`int`)
- *repetita juvant*: il tipo di `m1` (cioè `int`) **non** influisce sul processo di deduzione e quindi non può essere usato per risolvere l'ambiguità

Osservazione importante

- esiste una differenza sostanziale tra un template di funzione e le sue possibili istanziazioni
  - un template di funzione **NON** è una funzione (è un "generatore" di funzioni)
  - una istanza di un template di funzione è una funzione
- alcune conseguenze:
  - non posso prendere l'indirizzo di un template di funzione
    (prendo l'indirizzo di una sua istanza specifica)
  - non posso effettuare una chiamata di un template di funzione
    (chiamo una sua istanza specifica)
  - non posso passare un template di funzione come argomento ad un'altra funzione
    (passo una istanza specifica, cioè passo il suo indirizzo sfruttando il type decay)
  - se compilo una unità di traduzione ottenendo un object file e ne osservo il contenuto con il comando `nm`, vedrò solo le **istanze** dei template di funzione
    (non vedrò i template di funzione)
- nel parlare comune, però, spesso si dice "chiamata di un template di funzione" (o "chiamata di funzione templatica") intendendo la chiamata di una specifica istanza

Una analogia

- consideriamo il caso di uno studente che intenda richiedere alla Segreteria Studenti la modifica del proprio piano degli studi
- sui siti web dei corsi di laurea è disponibile un modulo per compilare la *"domanda di modifica del piano degli studi"*
- i moduli sono schemi di domanda, parametrici, e corrispondono al concetto di template
- in essi sono lasciati degli spazi (parametri) che devono essere compilati con i dati (argomenti) del corso di studi e dello studente
- il processo di istanziazione corrisponde alla compilazione del modulo: il modulo compilato corrisponde quindi all'istanza del template
- alla Segreteria Studenti occorre fare avere l'istanza (il modulo compilato), in quanto del solo template (il modulo in bianco) non saprebbero che farsene

Specializzazione (esplicita) di template di funzione

- a volte la definizione di un template di funzione è adeguata **per molti non per tutti** i casi di interesse
  - il codice generico potrebbe fornire una implementazione poco efficiente quando ai parametri del template sono associati argomenti particolari
  - oppure potrebbe fornire un risultato ritenuto errato
- ad esempio, il template `max` può essere istanziato anche con il tipo `T = const char*`
  - si ottiene una funzione che restituisce il massimo dei due puntatori
  - ma l'utente intendeva fare il confronto lessicografico tra due stringhe stile C
- per risolvere questi casi, è possibile fornire una definizione alternativa della funzione templatica, **specializzata** per gli argomenti in questione

Esempio di specializzazione

```
// definizione del template primario
template <typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}

// specializzazione del template per T = const char*
template <>
const char* max<const char*>(const char* a, const char* b) {
  return strcmp(a, b) > 0;
}
```

- a livello sintattico, si noti la lista vuota dei parametri <> ad indicare che si tratta di una **specializzazione totale** (non sono ammesse specializzazioni parziali per i template di funzione)

Esempio di specializzazione (ii)

- anche in questo caso è possibile omettere la lista degli argomenti (`<const char*>`), lasciando che venga dedotta dal compilatore

```
template <>
const char* max(const char* a, const char* b) { ... }
```

- si noti che sarebbe comunque possibile evitare la specializzazione del template e fornire invece l'implementazione specifica per il tipo `const char*` come **funzione "normale"**, sfruttando l'overloading di funzioni:

```
const char* max(const char* a, const char* b) { ... }
```

- diventa quindi importante capire come si comporta il meccanismo di risoluzione dell'overloading in questi casi (approfondimento effettuato successivamente)

Istanziazioni esplicite di template

- abbiamo visto che per un template è possibile fornire **istanziazioni implicite** (date dall'uso del template) e **specializzazioni esplicite**
- esiste anche la possibilità di richiedere **esplicitamente** al compilatore l'**istanziazione** di un template, indipendentemente dal fatto che questo venga utilizzato o meno
- sono previste due sintassi, corrispondenti a due casi di uso distinti (che tipicamente occorrono in unità di traduzione **diverse** facenti parte della stessa applicazione)
  - dichiarazione di istanziazione esplicita
  - definizione di istanziazione esplicita

Esempi di istanziazione esplicita

- **dichiarazione** di istanziazione (nell'unità di traduzione A):

```
extern template float max(float a, float b);
```

- **definizione** di istanziazione (nell'unità di traduzione B):

```
template float max(float a, float b);
```

- a livello sintattico, si notino:
  - l'assenza della lista dei parametri del template (la parola chiave `template` NON è seguita dalle parentesi angolate), che differenzia le istanziazioni esplicite dalle **specializzazioni** esplicite
  - la presenza o meno della parola chiave `extern`, che differenzia le **dichiarazioni** dalle **definizioni** di istanziazione

Istanziazioni esplicite: perché?

- un uso accorto delle istanziazioni esplicite consente di
  - diminuire i tempi di compilazione
  - generare object file di dimensioni minori
- la **dichiarazione di istanziazione** informa il compilatore che, quando verrà usata quella istanza del template, **NON** deve essere prodotta la corrispondente definizione dell'istanza (evitando quindi la generazione del codice)
- intuitivamente, la parola chiave `extern` indica che il codice dovrà essere trovato dal linker "esternamente" a questa unità di traduzione, cioè in un object file generato dalla compilazione di un'altra unità di traduzione
- la **definizione** di istanziazione, invece, informa il compilatore che quella particolare istanza del template deve essere generata, a prescindere dal fatto che nella unità corrente venga o meno usata; in questo modo, le altre unità potranno essere collegate con successo

Template di classe

- un template di classe è un costrutto del linguaggio C++ che consente di scrivere uno **schema parametrico** per una classe/struct.
- quasi tutti i concetti esposti per il caso dei template di funzione possono essere applicati ai template di classe: nel seguito si sottolineano le differenze (poche ma importanti)

```
// dichiarazione pura di un template di classe
template <typename T> class Stack;

// definizione di un template di classe
template <typename T>
class Stack {
public:
  /* ... */
  void push(const T& t);
  void pop();
  /* ... */
};
```

Nome del template vs nome dell'istanza

- nel caso dei template di classe, è importante distinguere tra il nome del template (`Stack`) e il nome di una sua specifica istanza (`Stack<int>`)
- infatti, fino al c++14, per i template di classe **NON si applica la deduzione dei parametri del template**: la lista degli argomenti va indicata obbligatoriamente

```
Stack<int> s1; // istanziazione implicita del tipo Stack<int>
Stack s2 = s1; // errore: non viene dedotto il tipo T = int
auto s2 = s1;  // ok: dal c++11 abbiamo la deduzione di tipo
               // dall'inizializzatore, usando auto
```

- nota: lo standard c++17, introducendo la CTAD, ha rimosso (in alcuni casi) questa limitazione; non tratteremo questa estensione

Nome del template vs nome dell'istanza (ii)

- quando siamo **all'interno dello scope del template di classe** è lecito usare il nome del template per indicare, in realtà, il nome della classe

```
template <typename T>
class Stack {
  // qui Stack è una abbreviazione (lecita) di Stack<T>
  Stack& operator=(const Stack&);
  /* ... */
}; // usciamo dallo scope di classe

// definizione (out-of-line)
template <typename T>
Stack<T>& // il tipo di ritorno è fuori scope
Stack<T>::operator=(const Stack& y) { // parametro nello scope
  Stack tmp = y; // nello scope di classe, è sufficiente Stack
  // ...
}
```

Istanziazione on demand

- quando si istanzia implicitamente una classe templatica, vengono generate solo le funzionalità necessarie per il funzionamento del codice che causa l'istanziazione
- esempio: se istanzio `Stack<int>` ma non uso (direttamente o indirettamente) il metodo

  ```
  void Stack<int>::push(const int&);
  ```

  tale metodo NON verrà istanziato
- questa scelta del linguaggio ha pro e contro:
  - **contro**: quando scrivo i test per la classe templatica devo prestare attenzione a fornire un insieme di test che copra tutte le funzionalità di interesse; le funzionalità NON testate (e quindi non istanziate) potrebbero addirittura generare errori di compilazione al momento dell'istanziazione da parte dell'utente
  - **pro**: per lo stesso motivo, posso usare un sottoinsieme delle funzionalità della classe istanziandola con argomenti che soddisfano solo i requisiti di quelle funzionalità; il fatto che quegli argomenti siano "scorretti" per le altre funzionalità (che non uso) non mi impedisce l'utilizzo dell'interfaccia "ristretta"

Esempio

- supponiamo che il template di classe `Stack<T>` fornisca un metodo `print()`, implementato invocando il corrispondente metodo `print()` del parametro `T` su ognuno degli oggetti contenuti nello stack
- questo significa che, per usare il metodo `Stack<T>::print()`, il tipo `T` **deve** fornire a sua volta il metodo `T::print()`
- si noti, per esempio, che il tipo `int` non è una classe e quindi un tentativo di istanziare `Stack<int>::print()` genera un errore di compilazione
- l'errore, però, lo si ottiene **solo** se effettivamente si prova a istanziare `Stack<int>::print()`; se ci si limita ad istanziare gli altri metodi, come `Stack<int>::push()`, l'istanza è generata correttamente ed è utilizzabile

Specializzazioni di template di classe

- come nel caso dei template di funzione, anche i template di classe possono essere istanziati (implicitamente o esplicitamente) e specializzati
- un esempio di **specializzazione totale**: l'header file standard `<limits>` fornisce il template di classe `std::numeric_limits`, usando il quale si possono ottenere informazioni sui tipi numerici built-in:

```
#include <limits>

int foo() {
  long minimo = std::numeric_limits<long>::min();
  long massimo = std::numeric_limits<long>::max();
  bool char_con_segno = std::numeric_limits<char>::is_signed;
}
```

Specializzazioni di template di classe (ii)

- nell'header file <limits> troviamo, tra le altre cose, le specializzazioni totali che consentono di rispondere alle interrogazioni dell'utente:

```
/* ... */
// numeric_limits<char> specialization.
template<>
  struct numeric_limits<char>
/* ... */
// numeric_limits<long> specialization.
template<>
  struct numeric_limits<long>
/* ... */
```

Un'altra specializzazione: std::vector⟨bool⟩

- un altro esempio è dato dalla specializzazione `std::vector<bool>` del template `std::vector` (definita nell'header file `vector`)
- la specializzazione è definita allo scopo di fornire una versione del contenitore ottimizzata per **risparmiare spazio di memoria** (codificando ogni valore booleano con un singolo bit)

Specializzazioni parziali di template di classe

- a differenza dei template di funzione, i template di classe supportano anche le **specializzazioni parziali**
- sono specializzazioni di template che sono applicabili non per una scelta specifica degli argomenti (come nel caso delle specializzazioni totali), ma per sottoinsiemi di tipi
- una specializzazione parziale di un template (di classe), quindi, è **ancora un template di classe**, ma di applicabilità meno generale

Tornando all'analogia precedente . . .

- riprendiamo l'analogia dei moduli per la domanda di modifica di piano di studi:
  - il modulo da compilare è il template primario (per uno studente qualsiasi)
  - il modulo compilato in ogni sua parte è l'istanza (di uno specifico studente)
- una **specializzazione (esplicita) totale** corrisponde ad una domanda di modifica di piano degli studi "fuori standard", fatta (ad personam) da uno specifico studente e che non segue necessariamente lo schema del modello generale
- una **specializzazione (esplicita) parziale**, invece, corrisponde ad un *modulo diverso* (quindi è ancora un template), che però viene utilizzato solo da uno specifico sottoinsieme degli studenti (per esempio, gli studenti iscritti alle lauree magistrali a ciclo unico)
- quando uno studente di Medicina e Chirurgia chiede il modulo da compilare, gli si fornisce il modulo specializzato (parzialmente): per ottenere la domanda vera e propria dovrà compilare (istanziare) il modulo specializzato

Esempio di specializzazione parziale

- le specializzazione parziali di template di classe sono poco frequenti (anche nella libreria standard)
- tra di esse tratteremo (quando affronteremo gli iteratori) il caso della specializzazione parziale per i puntatori degli `std::iterator_traits`
- nell'header file `<iterator>` troviamo:

```
// Partial specialization for pointer types.
template<typename _Tp>
  struct iterator_traits<_Tp*>
  /* ... */
```

- a livello di sintassi, il fatto che si tratti di una specializzazione **parziale** si deduce dalla contemporanea presenza:
  - della lista, non vuota, dei parametri del template (`<typename _Tp>`), e
  - della lista, non vuota, degli argomenti del template (`<typename _Tp*>`) nella quale si nomina ancora il parametro del template

Altre tipologie di template

- gli standard più recenti hanno introdotto nuove forme di template, sui quali non faremo approfondimenti per motivi di tempo
- **template di alias**

```
template <typename T>
using Vec<T> = std::vector<T, std::allocator<T>>;
```

  istanziando i quali si ottengono alias di tipi di dato

- **template di variabile**

```
template <typename T>
const T pi = T(3.14159265358979323851);
```

  istanziando i quali si ottengono variabili globali (namespace scope) o dati membro statici (class scope)