Template in C++

Template di funzione

Un template di funzione è un costrutto del linguggio C++ che consente di scrivere un *modello* (schema) parametrico per una funzione.

Esempio:

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

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

Nell'esempio precedente abbiamo prima dichiarato e poi definito un template di funzione di nome `max`.

> Come al solito, la definizione è anche una dichiarazione.

Nell'esempio, `T` è un parametro di template: il parametro viene dichiarato essere un `typename`(nome di 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 per coerenza si dovrebbe preferire l'uso di "typename"). La lista dei parametri può contenerne un numero arbitrario, separati da virgole; oltre ai parametri che sono nomi di tipo, vedremo che esistono anche altre tipologie (valori, template).

> Si noti che per convenzione (non è una regola del linguaggio) 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); // dichiara la stessa funzione foo

template
T max(T a, T b);
template
U max(U x, U y); // dichiara lo stesso template max
```

Torna all'indice

Istanziazione di un template di funzione

Dato un template di funzione, è possibile "generare" da esso una o più funzioni mediante il meccanismo di istanziazione (del template): l'istanziazione fornisce un *argomento* (della tipologia corretta) ad ognuno dei parametri del template.

L'istanziazione avviene spesso in maniera *implicita*, quando si fa riferimento al nome del template allo scopo di "usarne" una particolare istanza. 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.

Esempio:

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

  // istanziazione della funzione
```

```
  //   double max(double, double)
  double d = max(d1, d2);
}
```

Quando si istanzia un template di funzione, è possibile evitare la sintassi esplicita per gli argomenti del template, lasciando al compilatore il compito di *dedurre* tali argomenti a partire dal tipo degli argomenti passati alla chiamata di funzione.

Esempio:

```
void foo(char c1, char c2) {
  // istanziazione della funzione
  //   char max(char, char);
  int m = max(c1, c2);
  // il legame T = char viene dedotto dal tipo degli argomenti c1 e c2;
  // si noti che il tipo di m (int) non influisce sul processo di deduzione
}
```

Il processo di deduzione potrebbe fallire a causa di ambiguità:

```
void foo(double d, int i) {
  int m = max(d, i); // errore
  // il compilatore non può dedurre un unico tipo T coerente con d e i
  int m = max(d, i); // ok: evito la deduzione
}
```

Torna all'indice

Nota bene

È opportuno sottolineare la differenza sostanziale tra un template di funzione e le sue possibili istanziazioni. In particolare: un template di funzione NON è una funzione (è un "generatore" di funzioni); una istanza di un template di funzione è una funzione.

Per esempio, NON posso prendere l'indirizzo di un template di funzione (posso prendere l'indirizzo di una istanza specifica); non posso effettuare una chiamata di un template (chiamo una istanza specifica); non posso passare un template come argomento ad una funzione (passo una istanza specifica, che corrisponde a passarne l'indirizzo usando il type decay).

Se compilo una unità di traduzione ottenendo un object file e osservo il contenuto dell'object file con il comando `nm`, vedrò solo le *istanze* dei template di funzione (non vedrò i template di funzione).

Fatta questa doverosa sottolineatura, va comunque detto che nel linguaggio comune spesso si parla di "chiamata di un template di funzione" per indicare la chiamata di una sua specifica istanza.

Torna all'indice

Una utile analogia

A livello intuitivo può essere utile fare la seguente analogia. Sui siti web dei corsi di laurea sono spesso resi disponibili dei moduli per compilare una domanda di modifica di piano di studio.

Questi moduli sono dei "modelli" di domanda, parametrici, e corrispondono al concetto di template: in essi sono lasciati degli spazi (i parametri) che devono essere compilati con i dati dello studente per produrre una (vera e propria) domanda di modifica di piano di studio.

Il processo di istanziazione corrisponde alla compilazione del modulo: ad ogni parametro si associa il corrispondente argomento. Il modulo compilato corrisponde quindi all'istanza del template. Alla segreteria studenti occorre fare avere l'istanza (il modulo compilato), in quanto del template (il modulo in bianco) non saprebbero che farsene.

Torna all'indice

Specializzazione esplicita di un template di funzione

Capita a volte che la definizione di un template di funzione sia adeguata per molti, ma non per tutti i casi di interesse; ad esempio, il codice scritto potrebbe fornire un risultato ritenuto sbagliato quando ai parametri del template sono associati argomenti particolari.

Ad esempio, il template `max` può essere istanziato anche con il tipo `const char*`, ottenendo una funzione che restituisce il massimo dei due puntatori passati, quando invece, molto probabilmente, le intenzioni dell'utente era di fare un confronto lessicografico tra due stringhe stile C.

Per ovviare, è possibile fornire una definizione alternativa della funzione templatica, "specializzata" per gli argomenti problematici, nel modo seguente:

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

// specializzazione esplicita (per T = const char*) del template
template <>
const char* max(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).

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 stato possibile evitare la specializzazione del template e fornire la versione specifica per i `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 (l'approfondimento verrà effettuato successivamente).

Torna all'indice

Instanziazioni 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 effettivamente utilizzato.

Son previste due sintassi, corrispondenti a due casi di uso distinti (che tipicamente occorrono in unità di traduzione diverse facenti parte della stessa applicazione).

1. Dichiarazione di istanziazione esplicita

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

2. Definizione di istanziazione esplicita

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

A livello sintattico, si noti l'assenza della lista dei parametri (la parola chiave `template` NON è seguita da parentesi angolate): questo differenzia le istanziazioni esplicite dalle specializzazioni esplicite.

Il caso 1 (dichiarazione) 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. In pratica, la *dichiarazione* di istanziazione esplicita impedisce che vengano effettuate le istanziazioni implicite (per diminuire i tempi di compilazione e/o generare object file più piccoli).

Il caso 2 (definizione) è complementare al caso 1: si informa il compilatore che quella particolare istanza del template va generata, a prescindere dal fatto che in questa unità di traduzione ne venga o meno effettuato l'utilizzo. Serve ad assicurarsi che le altre unità di traduzione (che hanno usato il caso 1) possano essere collegate con successo.

Torna all'indice

Template di classe

Un template di classe è un costrutto del linguggio che consente di scrivere un *modello* parametrico per una classe.

Quasi tutti i concetti esposti per il caso dei template di funzione possono essere applicati alle classi: nel seguito si sottolineano le differenze (poche ma importanti).

Esempio:

```
// dichiarazione pura di un template di classe
// (nota: il nome T del parametro potrebbe essere omesso)
template
class Stack;

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

Nel caso dei template di classe, è ancora più importante distinguere tra il nome del template (`Stack`) e il nome di una specifica istanza (per esempio, `Stack<std::string>`). Infatti, per i template di classe NON si applica la deduzione dei parametri del template: la lista degli argomenti va indicata obbligatoriamente.

```
Stack s1; // istanziazione implicita del tipo Stack
            // (in particolare, del costruttore di default)

Stack s2 = s1; // errore: non viene dedotto il tipo T = int

auto s2 = s1;  // ok: il C++11 ha introdotto la deduzione di tipo
               // dall'inizializzatore, usando `auto`; viene anche
               // istanziato implicitamente il costruttore di copia
```

L'unico caso in cui è lecito usare il nome del template di classe per indicare (in realtà) il nome della classe ottenuta mediante istanziazione è all'interno dello scope del template di classe stesso.

Per esempio:

```
template
class Stack {
 /* ... */
 /* qui gli usi di Stack sono abbreviazioni (lecite) di Stack */
 Stack& operator=(const Stack&);
 /* ... */
}; // usciamo dallo scope di classe
```

```
// definizione (out-of-line)
template
Stack& // il tipo di ritorno è fuori scope di classe, devo scrivere
Stack::operator=(const Stack& y) { // parametro in scope di classe
 Stack tmp = y; // in scope di classe, è sufficiente Stack
 swap(tmp);
 return *this;
}
```

Torna all'indice

Istanziazione on demand

E' importante sottolineare che, quando si istanzia implicitamente una classe templatica, vengono generate solo le funzionalità necessarie per il funzionamento del codice che causa l'istanziazione. Quindi, nell'esempio precedente, per la classe `Stack<int>` NON vengono istanziati i metodi `Stack<int>::push(const int&)` e `Stack<int>::pop()`. Verranno istanziati se e quando utilizzati.

Questo scelta del linguaggio ha conseguenze positive e negative:

- In negativo: 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 instanziate) potrebbero addirittura generare errori di compilazione al momento dell'instanziazione da parte dell'utente.
- In positivo: per lo stesso motivo, posso usare un sotto-insieme 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 (non usate) non mi impedisce l'utilizzo dell'interfaccia "ristretta".

Esempio: Supponiamo che la 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 "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();` l'istanziazione dei metodi `Stack<int>::push()` e `Stack<int>::pop()` continua ad essere corretta e utilizzabile.

Torna all'indice

Istanziazioni e 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* di template di classe è fornito all'interno dell'header file standard `<limits>`, che fornisce il template di classe `std::numeric_limits`, attraverso il quale si possono per esempio ottenere informazioni sui tipi built-in.

Esempio di uso:

```
#include

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

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

```
// [...]
// numeric_limits specialization.
template<>
```

```
struct numeric_limits
// [...]
// numeric_limits specialization.
template<>
struct numeric_limits
// [...]
```

Un altro esempio è dato dalla specializzazione `std::vector<bool>` del template `std::vector`, creata allo scopo di fornire una versione del contenitore ottimizzata per risparmiare memoria (codificando ogni valore booleano con un singolo bit).

Torna all'indice

Specializzazioni parziali

A differenza dei template di funzione, i template di classe supportano anche le *specializzazioni parziali*. Si tratta di 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.

Torna all'indice

Un'altra analogia

Riprendiamo l'analogia dei moduli per la domanda di modifica di piano di studi: si era detto che il modulo in bianco è il template (per uno studente qualsiasi) e 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 *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.

Gli esempi di specializzazione parziale di template di classe sono meno frequenti (anche nella libreria standard). Tra di essi tratteremo (quando affronteremo gli iteratori) il caso della specializzazione parziale per i puntatori degli `std::iterator_traits`. Nell'header file `<iterator>` (in realtà, in un header file interno che dipende dalla specifica implementazione) troviamo:

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

Il fatto che si tratti di una specializzazione parziale si deduce dalla contemporanea presenza della lista (non vuota) dei parametri del template e della lista (non vuota) degli argomenti del template, nella quale si nomina ancora il parametro del template.

Torna all'indice

Altri template

Gli standard più recenti hanno introdotto nuove forme di template, sui quali non faremo approfondimenti.

Template di alias:

```
template
using Vec = std::vector>;
```

Template di variabile:

```
template
const T pi = T(3.14159265358979323851L);
```

Torna all'indice

Compilazione dei template

Il processo di compilazione dei template richiede che lo stesso codice sia analizzato dal compilatore in (almeno) due contesti distinti:

1. al momento della definizione del template, e
2. al momento della instanziazione del template.

Nella prima fase (definizione del template) il compilatore si trova ad operare con informazione incompleta. Si consideri il seguente esempio:

```
template
void incr(int& i, T& t) {
 ++i; // espressione indipendente dai parametri del template
 ++t; // espressione dipendente dai parametri del template
}
```

Sulla prima espressione il compilatore può effettuare tutti i controlli di correttezza previsti (sintattici e di semantica statica) e nulla vieterebbe di generare una porzione del codice oggetto.

Sulla seconda espressione, invece, gli unici controlli che possono essere effettuati sono dei banali controlli sintattici: non c'è modo per il compilatore di sapere se il tipo T fornisce effettivamente un operatore di preincremento. Questo controllo (e la segnalazione di eventuali errori) viene quindi "rimandato" alla successiva fase di instanziazione del template.

Torna all'indice

Conseguenza 1

Una prima conseguenza, di cui tenere conto quando si scrivono i programmi, è che la definizione di un template deve essere disponibile i tutti i punti del programma nei quali se ne richiede l'istanziazione.

In pratica, esistono tre modi per organizzare il codice sorgente quando si scrivono funzioni o classi templatiche:

1. Includere le definizioni dei template (comprese la definizioni di eventuali funzioni membro dei template di classe) prima di ogni loro uso nella unità di traduzione.

2. Includere le dichiarazioni del template (comprese le dichiarazioni delle eventuali funzioni membro dei template di classe) prima di farne uso e successivamente (prima o dopo gli usi) includere le definizioni del template nella unità di traduzione.

3. Sfruttando il meccanismo delle istanziazioni esplicite, includere solo le dichiarazioni dei template e le *dichiarazioni* di istanziazione esplicita prima di ogni loro uso nell'unità di traduzione, assicurandosi che le definizioni dei template e le *definizioni* di istanziazione esplicita siano fornite in un'altra unità di traduzione.

L'approccio più comune, perché più semplice, è il primo. Il secondo approccio si usa solo quando necessario (per esempio, nel caso di funzioni templatiche che si invocano l'un l'altra ricorsivamente). Il terzo approccio è usato raramente, tipicamente al solo scopo di diminuire i tempi di compilazione.

> Si noti che, nell'esercitazione sulla templatizzazione della classe Stack, abbiamo seguito il primo approccio, spostando tutte le definizioni dei metodi della classe all'interno dell'header file Stack.hh.

Torna all'indice

Conseguenza 2

Una seconda conseguenza del meccanismo di compilazione in due fasi dei template è che, in alcuni casi, occorre modificare il codice di implementazione dei template di funzioni o classe per fornire al compilatore qualche informazione utile ad evitare errori di compilazione.

Per esempio, consideriamo il seguente codice (non templatico):

```
struct S {
 using value_type = /* ... dettaglio implementativo ... */;
 /* ... */
};

void foo(const S& s) {
 S::value_type* ptr;
 /* ... */
}
```

Supponiamo ora di voler templatizzare la classe `S`, rendendola parametrica rispetto ad un qualche tipo usato al suo interno. Intuitivamente, il processo di "lifting" porterebbe ad adattare il codice in questo modo:

```
template
struct S {
 using value_type = /* ... dettaglio implementativo ... */;
 /* ... */
};

template
void foo(const S& s) {
 S::value_type* ptr;   // errore: ptr non dichiarato
 /* ... */
}
```

Il compilatore segnala un errore quando esamina la definizione del template di funzione foo (nella prima fase della compilazione dei template). In effetti, il compilatore si trova di fronte a codice del tipo

```
nome1 * nome2
```

e non sa nulla di `nome1` e `nome2`: siccome `nome1` è dipendente dal parametro template `T`, il compilatore assume che sia il nome di un "valore" (non di un "tipo"), interpretando l'istruzione come applicazione dell'operatore `*` binario; viene quindi segnalato un errore perché `nome2` (che non dipende dal parametro `T`) non è stato dichiarato.

Per risolvere il problema e comunicare correttamente le nostre intenzioni al compilatore, occorre informarlo che `S<T>::value_type` indica il nome di un tipo, aggiungendo la parola chiave `typename`:

```
template
void foo(const S& s) {
 typename S::value_type* ptr;   // ok, dichiaro un puntatore
 /* ... */
}
```

> NOTA BENE: occorre rendersi conto che la problematica suddetta si potrebbe presentare in maniera subdola. Si consideri per esempio questa variante:

```
int p = 10;

template
void foo(const S& s) {
 S::value_type* p;   // compila senza errori (operator* binario)
 /* ... */
}
```

> Per pura sfortuna, esiste una dichiarazione di un intero `p` visibile quando il compilatore valuta l'istruzione, per cui il compilatore non rileva l'errore (assumendo che il programmatore intenda fare una sorta di moltiplicazione del valore di `S<T>::value_type` con il valore 10 memorizzato nella variabile `p` dello scope globale).

Torna all'indice