Deduzione automatica dei tipi di dato

Enea Zaffanella

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

Template type deduction

- la template type deduction (deduzione dei tipi per i parametri template) è il processo messo in atto dal compilatore per semplificare l'istanziazione (implicita o esplicita) e la specializzazione (esplicita) dei template di funzione
- questa forma di deduzione è utile per il programmatore in quanto consente di evitare la scrittura (noiosa, ripetitiva e soggetta a sviste) della lista degli argomenti da associare ai parametri del template di funzione
- il processo è intuitivo e comodo da usare, ma può riservare sorprese
- in questa nota, prenderemo spunto da alcuni esempi contenuti nel testo
  Effective Modern C++ di Scott Meyers (O'Reilly)

Le regole per la deduzione (i)

- per semplificare al massimo la discussione, supponiamo di avere la seguente dichiarazione di funzione templatica:

  ```
  template <typename TT> void f(PT param);
  ```

  dove
  - `TT` è il **nome** del parametro del template di funzione
  - **PT** è una **espressione sintattica** (anche complicata) che denota il tipo del parametro `param` della funzione
- nota: chiaramente, affinché vi possa essere una deduzione, occorre che il nome `TT` occorra all'interno dell'espressione sintattica **PT**
- negli esempi seguenti si assume che `i` e `ci` siano state definite in questo modo:

  ```
  int i = 0;
  const int ci = 1;
  ```

Le regole per la deduzione (ii)

- a fronte della chiamata di funzione `f(`**arg**`)`, il compilatore usa il tipo **targ** di **arg** per dedurre:
  - un tipo specifico **tt** per `TT`
  - un tipo specifico **pt** per **PT**

  causando l'istanziazione del template di funzione e ottenendo la funzione

  `void f<`**tt**`>(`**pt**` param);`
- nota: i tipi dedotti **tt** e **pt** sono correlati, ma spesso non sono identici
- il processo di deduzione distingue tre casi:
  1. **PT** è sintatticamente uguale a `TT&&` (cioè, **PT** è una **universal reference**)
  2. **PT** è un tipo puntatore o riferimento (ma non una universal reference)
  3. **PT** non è né un puntatore né un riferimento

Universal reference

- si ha un riferimento "universale" quando abbiamo solo l'applicazione di `&&` al nome di un parametro typename, senza nessun altro modificatore
- quindi, se `TT` è il parametro typename:

`TT&&`	universal reference
`const TT&&`	rvalue reference (**non** universal reference)
`std::vector<TT>&&`	rvalue reference (**non** è universal reference)

- l'aggettivo "universal" indica che, sebbene venga usata la sintassi per i riferimenti a rvalue, può essere dedotto per **PT** un riferimento a rvalue **oppure** a lvalue, a seconda del tipo **targ** di **arg**
- nota: la terminologia "universal reference" non fa parte dello standard; è stata introdotta da Meyers per spiegare più semplicemente una parte delle regole di deduzione

Caso 1: **PT** universal reference

- `template <typename TT> void f(TT&& param);`
- esempio 1.1 (deduzione di rvalue):

```
f(5);  // targ = int
       // deduco pt = int&&, tt = int
f(std::move(i)); // targ = int&&
                 // deduco pt = int&&, tt = int
```

- esempio 1.2 (deduzione di lvalue):

```
f(i);  // targ = int&
       // deduco pt = int&,  tt = int&
f(ci); // targ = const int&
       // deduco pt = const int&, tt = const int&
```

Caso 2: **PT** puntatore

- si effettua un *pattern matching* tra il tipo **targ** e **PT**, ottenendo i tipi **tt** e **pt** di conseguenza
- esempio 2.1: `template <typename TT> void f(TT* param);`

```
f(&i);  // targ = int*
        // deduco pt = int*, tt = int
f(&ci); // targ = const int*
        // deduco pt = const int*, tt = const int
```

- esempio 2.2: `template <typename TT> void f(const TT* param);`

```
f(&i);  // targ = int*
        // deduco pt = const int*, tt = int
f(&ci); // targ = const int*
        // deduco pt = const int*, tt = int
```

Caso 2: **PT** riferimento non universale

- il caso dei riferimenti è analogo
- esempio 2.3: `template <typename TT> void f(TT& param);`

```
f(i);  // targ = int&
       // deduco pt = int&, tt = int
f(ci); // targ = const int&
       // deduco pt = const int&, tt = const int
```

- esempio 2.4: `template <typename TT> void f(const TT& param);`

```
f(i);  // targ = int&
       // deduco pt = const int&, tt = int
f(ci); // targ = const int&
       // deduco pt = const int&, tt = int
```

Caso 3: **PT** né puntatore, né riferimento

- `template <typename TT> void f(TT param);`
  abbiamo un passaggio **per valore**, quindi argomento e parametro sono due oggetti distinti: eventuali riferimenti e qualificazioni const (a livello esterno) dell'argomento **non si propagano al parametro**
- esempio 3.1:

```
f(i);  // targ = int&
       // deduco pt = int, tt = int
f(ci); // targ = const int&
       // deduco pt = int, tt = int
```

- esempio 3.2: le qualificazioni const **a livello interno** si propagano al parametro

```
const char* const p = "Hello";
f(p);  // targ = const char* const&
       // deduco pt = const char*, tt = const char*
```

Deduzione tipi per auto

- a partire dal c++11, nel linguaggio è stata introdotta la possibilità di definire le variabili usando la parola chiave `auto` (senza specificarne esplicitamente il tipo)
- si lascia al compilatore il compito di dedurre il tipo a partire dall'espressione usata per inizializzare la variabile (che **deve** avere un inizializzatore)
- esempi:

```
auto i = 5;                 // i ha tipo int
const auto d = 5.3;         // d ha tipo const double
auto ii = i * 2.0;          // ii ha tipo double
const auto p = "Hello";     // p ha tipo const char* const
```

- la **auto type deduction** segue in larga parte le stesse regole della template type deduction

Esempio di auto type deduction

- quando si osserva una definizione di variabile come

```
auto& ri = ci;
```

  ci si riconduce al caso precedente della chiamata `f(`**arg**`)` per il template di funzione

```
template <typename TT> void f(PT param);
```

  nel quale
  - la parola chiave `auto` svolge il ruolo del parametro template `TT`
  - la sintassi `auto&` corrisponde a **PT**
  - l'inizializzatore `ci` (di tipo `const int&`) corrisponde all'espressione **arg**
- questo esempio corrisponde al **caso 2** della deduzione di parametri template
  - per `auto` si deduce il tipo `const int`
  - per `ri` si deduce il tipo `const int&`

Attenzione: un caso particolare

- la auto template deduction **differisce** dalla template type deduction quando l'inizializzatore è indicato mediante la sintassi che prevede le **parentesi graffe** (*sintassi uniforme di inizializzazione*)
- esempio:

```
auto i = { 1 };
```

- in questo caso speciale, che non approfondiremo, l'argomento si considera di tipo `std::initializer_list<T>`
- alcune **linee guida di programmazione** suggeriscono di usare `auto` **quasi sempre** per inizializzare le variabili; nell'acronimo AAA (Almost Always Auto), la prima A indica appunto la presenza di eccezioni alla linea guida, che sono proprio quelle dette sopra per gli inizializzatori con parentesi graffe