Overloading

Definizione e meccanismo di risoluzione

Il C++ supporta il concetto di overloading (sovraccaricamento) per il nome delle funzioni: è possibile definire più funzioni che condividono lo stesso nome e si differenziano solo per il numero e/o il tipo degli argomenti.

L'overloading è utile, in generale, per evitare di dovere fornire un nome distinto a funzioni che, in buona sostanza, fanno operazioni del tutto analoghe, solo su argomenti di tipo diverso.

Per esempio, nella libreria matematica del C (linguaggio di programmazione che *NON* supporta il concetto di overloading), abbiamo le tre funzioni

```
float sqrtf(float arg);
double sqrt(double arg);
long double sqrtl(long double arg);
```

che calcolano la radice quadrata di un float, un double e un long double. La differenziazione nel nome risulta alquanto artificiosa. In contrasto, nella libreria standard del C++ (header file `cmath`) si trovano le tre funzioni:

```
float sqrt(float arg);
double sqrt(double arg);
long double sqrt(long double arg);
```

Per il programmatore è più semplice ricordare un solo nome. La differenza tra i due approcci può diventare ancora più evidente se si pensa ad una funzione di stampa (per esempio `print`) che debba essere applicata a dozzine o centinaia di tipi di dato diversi.

> Va comunque notato che anche il C supporta una forma ristretta di overloading, che però è confinata al caso degli operatori definiti sui tipi built-in. Per esempio, nell'espressione `v + 1` l'operatore `+` corrisponde alla somma di interi se `v` è di tipo intero, alla somma di double se `v` è di tipo double, ecc...

Oltre ai suddetti motivi di comodità, vedremo come l'overloading di funzione risulterà essenziale quando dovremo scrivere codice generico (utilizzando template di classe e template di funzione).

Quando però si passa dal ruolo di utente di una interfaccia software al ruolo di sviluppatore dell'interfaccia stessa, occorre prestare molta attenzione a non creare un insieme di funzioni in overloading che sia "fuorviante" per l'utente dell'interfaccia.

È quindi necessario capire a fondo i meccanismi della cosiddetta "risoluzione dell'overloading", che è il processo seguito dal compilatore all'atto di esaminare ogni singola chiamata di funzione allo scopo di stabilire quale delle funzioni dichiarate debba essere invocata per quella chiamata.

Torna all'indice

Le tre fasi della risoluzione dell'overloading di funzione

Come detto, la risoluzione dell'overloading è un processo svolto staticamente dal compilatore, quando trasforma in assembler il codice contenuto nell'unità di traduzione (prodotta dal preprocessore). In particolare, non può essere influenzata dalle informazioni presenti in altre unità di traduzione (compilazione separata) e neanche dalle informazioni disponibili a tempo di esecuzione.

Daremo una descrizione abbastanza dettagliata (ma comunque incompleta) del processo, che si suddivide in tre fasi. Queste fasi vengono ripetute per ogni singola chiamata di funzione presente nel codice sorgente:

1. individuazione delle funzioni candidate;
2. selezione delle funzioni utilizzabili;
3. scelta della migliore funzione utilizzabile (se esiste).

Torna all'indice

Fase 1: le funzioni candidate

L'insieme delle funzioni candidate per una specifica chiamata di funzione è un sottoinsieme delle funzioni che sono state dichiarate all'interno dell'unità di traduzione. In particolare, le funzioni candidate:

1. hanno lo stesso nome della funzione chiamata (non importa numero e tipo degli argomenti)
2. sono visibili nel punto della chiamata

Nella valutazione del primo punto (*nome*) occorre comunque tenere presente che, per gli operatori, la sintassi della chiamata di funzione può variare nella forma:

- operatore prefisso: `++a`
- operatore postfisso: `ai[5]`
- operatore infisso: `a - b`
- sintassi funzionale: `operator*(a, b)`

mentre la corrispondente dichiarazione utilizzerà sempre la forma del nome esteso (rispettivamente, `operator++`, `operator[]`, `operator-` e `operator*`).

Più interessante (e complicato) è il secondo punto, riguardante la visibilità delle dichiarazioni nel punto di chiamata. Occorre infatti prestare attenzione alle seguenti casistiche:

1. Nel caso di invocazione di un metodo di una classe mediante la sintassi obj.metodo (oppure `ptr->metodo`), se l'oggetto `obj` (oppure il puntatore `ptr`) hanno come tipo statico `S`, allora la ricerca inizierà nello scope di classe `S`.

   > Si noti che viene considerato il tipo *statico* (cioè il tipo noto a tempo di compilazione), mentre viene ignorato il tipo *dinamico* (che è noto solo a tempo di esecuzione).

   Esempio:

```
struct S { /* ... */ };
struct T : public S { /* ... */ };
S* ptr = new T;
ptr->foo(); // chiamata
// ptr ha tipo statico S* e tipo dinamico T* (esempio Java Quadr./Rett.)
// quindi la ricerca di foo avviene a partire dallo scope di S
// (eventuali funzioni T::foo non sono visibili nel punto di chiamata)
```

2. Nel caso di qualificazione della funzione chiamata, la ricerca delle funzioni candidate inizierà nel corrispondente scope. Esempio:

```
namespace N {
  void foo(int);
}
void foo(char); // dichiarazione NON visibile per la chiamata
int main() {
    N::foo('c'); // la ricerca inizia nello scope di namespace N
}
```

3. Attenzione a non confondere l'overloading con l'hiding Esempio:

```
struct S { void foo(int); };
struct T : public S { void foo(char); };
T t; // tipo statico e dinamico coincidono (T)
t.foo(5);
// la ricerca inizia nello scope di T; la funzione S::foo
// non va in overloading, perché viene nascosta (hiding);
```

4. Attenzione ad eventuali dichiarazioni e direttive di using, che modificano la visibilità delle dichiarazioni: nell'esempio precedente, se nella classe T fosse aggiunto `using S::foo;` allora `S::foo` e `T::foo` sarebbero entrambe visibili e andrebbero in overloading.

5. Attenzione all'ADL (Argument Dependent Lookup)

Torna all'indice

ADL - Argument Dependent Lookup

La regola ADL (Argument Dependent Lookup), detta anche *Koenig's lookup*, stabilisce che

1. nel caso di una chiamata di funzione NON qualificata,
2. se vi sono (uno o più) argomenti "`arg`" aventi un tipo definito dall'utente (cioè hanno tipo `struct/class/enum S`, o riferimento a `S` o puntatore a `S`, possibilmente qualificati) e
3. il tipo suddetto è definito nel `namespace N`

allora la ricerca delle funzioni candidate viene effettuta anche all'interno del namespace N.

Esempio:

```
namespace N {
 struct S { };
 void foo(S s);
 void bar(int n);
} // namespace N

int main() {
 N::S s;
 foo(s);   // chiamata 1
 int i = 0;
 bar(i);   // chiamata 2
}
```

1. Per la chiamata 1, si applica la regola ADL, perché il nome `foo` non è qualificato e l'argomento `s` ha tipo `struct S` definito dall'utente all'interno del `namespace N`. Quindi il `namespace N` viene "aperto" rendendo visibile la dichiarazione di `N::foo(S)` nel punto della chiamata (e quindi rendendola candidata).

2. In contrasto, per la chiamata 2, NON si applica la regola ADL, perché l'argomento ha tipo `int` (che non è definito dall'utente) e quindi non viene aperto nessun namespace.

   NOTA BENE: la regola ADL è quella che consente al programma `"Hello, world!"` di funzionare come ci si aspetta. La chiamata `std::cout << "Hello, world!"` corrisponde alla invocazione di funzione `operator<<(std::cout, "Hello, world!")` La chiamata non è qualificata e il primo argomento ha tipo `std::ostream`, che è un tipo definito dall'utente all'interno del `namespace std`. Quindi, il `namespace std` viene "aperto" e tutte le funzioni di nome `operator<<` dichiarate al suo interno diventano visibili (e quindi candidate).

Torna all'indice

Fase 2: selezione delle funzioni utilizzabili

Effettuata la scelta delle funzioni candidate, occorre verificare quali di queste funzioni potrebbero essere effettivamente utilizzabili (*viable*) per risolvere la specifica chiamata considerata.

Per decidere se una funzione candidata è utilizzabile, è necessario verificare che:

1. il numero degli argomenti (nella chiamata di funzione) sia compatibile con il numero parametri (nella dichiarazione di funzione);
2. ogni argomento (nella chiamata di funzione) abbia un tipo compatibile con il corrispondente parametro (nella dichiarazione di funzione).

Con riferimento alla compatibilità del numero di argomenti:

- attenzione ad eventuali valori di default per i parametri;

- attenzione all'argomento implicito (`this`) nelle chiamate di metodi non statici.

Con riferimento alla compatibilità dei tipi argomento/parametro, occorre tenere conto che, oltre al caso della corrispondenza perfetta tra i due tipi (chiamata conversione identità o match perfetto, anche se tecnicamente in questo caso non si effettua nessuna conversione), è applicabile tutta una serie di conversioni implicite che potrebbero consentire di convertire il tipo dell'argomento nella chiamata nel tipo del parametro nella dichiarazione di funzione.

Si coglie l'occasione per ricordare la classificazione delle conversioni implicite in:

1. corrispondenze esatte (identità, trasformazioni di lvalue, qualificazioni);
2. promozioni;
3. conversioni standard;
4. conversioni definite dall'utente.

Torna all'indice

Sequenza di conversioni

Quando si verifica se una funzione è utilizzabile, per ogni argomento della chiamata è possibile effettuare una sequenza di conversioni (i.e., non ci si limita ad usare una sola conversione implicita).

Una sequenza di conversione "standard" è composta da: $$ \text{trasf. di lvalue + (promozione o conv. standard) + qualificazione} $$ Le varie componenti sono opzionali, ma se presenti devono essere nell'ordine specificato.

Esempio: supponendo che la classe D è derivata dalla classe base B,

```
double d = 3.1415; // un lvalue di tipo double
void foo(int);     // funzione che accetta un intero per valore

foo(d); // chiamata
```

la funzione `void foo(int)`, se candidata, è anche utilizzabile.

Si applica prima una trasformazione di lvalue (nello specifico, una trasformazione da lvalue a rvalue, che corrisponde alla lettura del valore contenuto nella locazione d) e quindi una conversione standard (da double a int).

Una sequenza di conversione "utente" è composta da: $$ \text{seq. conv. standard + conv. utente + seq. conv. standard} $$

Torna all'indice

Fase 3: selezione della migliore utilizzabile

Sia N il numero di funzioni utilizzabili:

- se N = 0, allora ho un errore di compilazione;
- se N = 1, allora l'unica utilizzabile è la migliore;
- se N > 1, allora occorre classificare le funzioni utilizzabili in base alla "qualità" delle conversioni richieste; se la classificazione (spiegata sotto) determina un'unica vincitrice, quella funzione è la migliore utilizzabile; altrimenti si ottiene un errore di compilazione (*chiamata ambigua*).

Per ognuno degli M argomenti presenti nella chiamata, si crea una classifica delle funzioni utilizzabili. La funzione migliore (se esiste) è quella che è preferibile rispetto a tutte le altre. Per decidere se `X` è preferibile rispetto ad `Y`, si confrontano `X` e `Y` su tutte le M classifiche corrispondenti agli M argimenti. `X` è preferibile a `Y` se:

- non perde in nessuno degli M confronti (i.e., vince o pareggia);
- vince almeno uno degli M confronti.

Fissato un argomento `A_i`, la funzione X vince rispetto alla funzione Y se la sequenza di conversioni `X_i` usata da X per `A_i` vince rispetto alla sequenza `Y_i` usata da Y.

Le regole del linguaggio per stabilire se una sequenza `X_i` vince sulla sequenza `Y_i` sono abbastanza intricate e non uniformi (sono elencate un certo numero di eccezioni). Nello standard C++14 la loro spegazione occupa almeno 3 pagine dense. A scopo didattico, usiamo quindi una versione semplificata.

Una sequenza `X_i` vince sulla sequenza `Y_i` se la peggiore conversione `X_worst` usata in `X_i` vince sulla peggiore conversione `Y_worst` usata in `Y_i`. Una conversione `X_worst` vince sulla conversione `Y_worst` se ha un "rank" migliore (corrispondenze esatte vincono sulle promozioni, che vincono sulle conversioni standard, che vincono sulle conversioni utente).

Torna all'indice

Esempio

!\[\[overloading_esempio.jpg\]\]

Torna all'indice

Ulteriori osservazioni

1. Le regole fin qui esposte NON prendono in considerazione alcuni casi speciali che si applicano in presenza di funzioni candidate ottenute istanziando (o specializzando) template di funzione. Questi casi verranno introdotti al momento opportuno.

2. Vale la pena sottolineare che, quando si scelgono le funzioni candidate per una chiamata, il numero e il tipo dei parametri della funzione NON sono considerati in alcun modo (entrano in gioco solo nella seconda fase, quando si restringe l'insieme delle candidate al sottoinsieme delle funzioni utilizzabili).

3. Analogamente, è opportuno sottolineare che, nel caso di invocazione di un metodo di una classe, il fatto che tale metodo sia dichiarato con accesso `public`, `private` o `protected` NON ha nessun impatto sul processo di risoluzione dell'overloading. Il processo determina la migliore funzione utilizzabile (se esiste); in seguito, se la funzione scelta non è accessibile per il chiamante, verrà segnalato un errore di compilazione, che però NON ha nulla a che fare con la risoluzione dell'overloading (in particolare, la migliore funzione utilizzabile ma non accessibile NON viene mai sostituita da un'altra funzione utilizzabile e accessibile).

Torna all'indice