

2. Overloading di Funzione

L'overloading é fondamentale in C++ in quanto rende il linguaggio piú comodo e flessibile, inoltre fornisce il supporto principale per il polimorfismo.

In questo capitolo si vedranno i concetti base dell'overloading di funzione e i metodi per la sua risoluzione.

Prerequisiti

Conversioni Implicite

Il compilatore effettua una conversione implicita quando il tipo utilizzato di un dato non corrisponde a quello atteso. Viene dunque definita una sequenza di operazioni che consente di convertire da un tipo all'altro.

Si parla di **promozione** o di **conversione** a seconda dell'operazione che si deve effettuare ad esempio:

```
long int a = 123; // Promozione
int b = 3.4 // Conversione
```

Sintenticamente la gerarchia é `double > float > long > int > char > bool`. La conversione potrebbe presentare delle perdite di informazione.

Overloading

In C++ viene data la possibilità di definire piú funzioni che condividono lo stesso nome ma si differenziano per tipo e per numero e tipo di argomenti. Risulta utile per evitare di dare nomi diversi a funzioni che fanno sostanzialmente la stessa operazione ma su tipi diversi.

Ad esempio:

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

Dal punto di vista dello sviluppatore dell'interfaccia, però, occorre prestare attenzione al fine di evitare un insieme di funzioni in overloading che possa risultare fuorviante all'utente.

Serve quindi conoscere a fondo i meccanismi per la cosiddetta **risoluzione** dell'overloading.

Risoluzione dell'Overloading

Questo é il processo seguito dal compilatore (senso stretto) atto a esaminare ogni chiamata di funzione e quindi di stabilire quale effettivamente debba essere invocata per quella chiamata.

Segue tre passaggi questo processo:

1. Individua i candidati

Una funzione viene detta **candidata** quando ha lo stesso nome della funzione che é stata chiamata ed é visibile nel punto in cui é stata chiamata.

Il secondo punto é il piú delicato, entra anche in gioco la regola **ADL** (Argument Dependent Lookup).

Essa stabilisce che se la chiamata NON é qualificata e vi sono argomenti aventi un tipo definito dall'utente che appartiene al namespace N, allora la ricerca delle funzioni candidate avverrá anche all'interno di N.

Esempio:

```
namespace N {  
  
    struct S {};  
    void foo(S s);  
    void bar(int n);  
}  
  
int main() {  
  
    N::S s;  
    foo(s); // Si applica ADL perché foo non é stata qualificata  
           // e s ha tipo struct S e rende la dichiarazione visibile  
           // N::foo(s).  
  
    int i=0;  
    bar(i) // NON si applica ADL poiché l'argomento é di tipo  
           // int non definito dall'utente. Di conseguenza non si apre  
           // il namespace.  
}
```

2. Seleziona le funzioni utilizzabili

Tra le candidate quelle **utilizzabili** sono quelle che hanno il numero di argomenti (nella chiamata) uguale a quello dei parametri (nella dichiarazione).

In piú serve che gli argomenti abbiano il tipo uguale al parametro corrispondente.

Nel caso della compatibilit  argomento-parametro bisogna tenere conto che non si parla solo di corrispondenze esatte, ma sono consentite anche le varie conversioni implicite.

3. Seleziona la migliore

Se N   il numero di funzioni utilizzabili quando $N=0$ si ha un errore di compilazione.

Con $N=1$ ho gi  trovato la migliore.

Mentre se $N>1$, occorre classificare le funzioni utilizzabili in base alla qualit  delle conversioni richieste.

Se la classificazione determina un'unica vincitrice allora quella   la migliore, altrimenti si ha un errore di compilazione (**chiamata ambigua**).

In generale una funzione vince su un'altra quando la sua conversione peggiore vince sulla conversione peggiore dell'altra.

Classifica Conversioni

Le conversioni in C++ si possono suddividere in quattro categorie, le vediamo partendo dalla pi  alta alla pi  bassa in termini di rank, che ci servir  appunto nella terza fase della risoluzione dell'overloading.

1. Corrispondenze esatte

Corrispondono a quelle conversioni implicite che preservano il valore dell'argomento. Si dividono in:

- *Identit *

Dette anche match perfetti.

Effettivamente non   una conversione, si verifica quando tipo di partenza e tipo di destinazione coincidono.

- *L-Value*

Sono le conversioni che si verificano quando si passa da locazione (l-value) a valore (r-value) come succede, ad esempio, nel decadimento array-puntatore.

- *Qualificazioni*
Aggiungono il qualificatore **const**.

```
const int* &i
```

2. Promozioni

Corrispondono alle conversioni implicite che preservano il valore dell'argomento e servono quando si vuole operare, ad esempio, tra un tipo **int** e un tipo piú piccolo.

Quest'ultimo dev'essere **promosso** a **int** (o **unsigned int**) per poter effettuare l'operazione.

Si dividono in:

- *Intere*
Da tipi interi piccoli (come **char**) a **int**.
- *Floatin point*
Da **float** a **double**.
- *Costanti di enumerazione*
Vengono convertite al tipo intero piú piccolo che possa contenerle.

3. Conversioni standard

Sono tutte quelle conversioni implicite che non fanno parte né della prima né della seconda categoria (ad esempio da **int** a **long**, da **char** a **double** e così via).

4. Definite dall'utente

Sono tutte quelle conversioni che vengono definite ad hoc dai programmatori.

Possono avere un uso **implicito** (ad esempio usando un costruttore unario che richiede un tipo di argomento differente da quello fornito) o uso **esplicito** tramite operatori di conversione.