11_Overloading

Table of contents

Overloading

Qual'ora sorgesse il problema per il programmatore di dare nomi esaustivi a sufficienza, il concetto di **overloading** viene in aiuto. Piuttosto che dare nomi diversi, a funzioni che pressapoco hanno lo stesso fine, forniamo nomi uguali, cambiando il resto.

```
// funzioni 'sqrt' che hanno scopi diversi in base al parametro
float sqrt(float arg);
double sqrt(double arg);
long double sqrt(long double arg);
```

I passi che portano alla decisione dell'overloading o meno, per il compilatore, sono:

1. individuazione delle funzioni candidate;
2. scartate le funzioni che sicuramente non possono intromettersi ("not viable");
3. scelta delle migliori funzioni invocabili ("viable").

Fase1: le funzioni candidate

Le candidate sono quelle con lo stesso nome della funzione chiamata, visibili nel punto della chiamata. Per capire in modo semplice, quale delle tante funzioni va considerata, ci ricordiamo che il compilatore osserva solo il *tipo statico* della classe, piuttosto che il dinamico.

```
struct S {};
struct T : struct S {};

// crea oggetto in heap tipo T, restituisce puntatore a tipo T
S* ptr = new T;
ptr -> foo(); // la ricerca di 'foo()' avviene a partire da S
```

La ricerca delle funzioni candidate inizierà nel *corrispondente* `namespace`.

```
namespace N { void foo(int); }
void foo(char); // non visibile dalla chiamata
int main() {
          N:foo('c'); // ricerca cominciata nel namespace N
}
```

Porre attenzione al fatto che l'*hiding* entra in gioco.

```
struct S { void foo(int); };
struct T : public S { void foo(char); };
T t;
t.foo(5); // 'foo()' verrà cercata nello scope di T
```

E che le direttive di `using` modificano la visibilità.

```
using S::foo;
```

Inoltre, la regola di Argument Dependent Lookup (ADL), stabilisce che: in caso di chiamata non qualificata, se vi sono argomenti aventi tipo definito dall'utente e il tipo suddetto è definito nel namespace `N`, allora la ricerca delle funzioni candidate viene prima effettuata anche all'interno del namespace `N`.

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

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

Fase2: le utilizzabili

Se nella 1ª fase era soltanto considerato il nome, in questa entrano in gioco i parametri e argomenti. Per capire se una funzione e' utilizzabile, vengono seguiti i punti:

1. il numero degli argomenti (in chiamata) compatibili con il numero di parametri (in dichiarazione);
2. ogni argomento abbia tipo compatibile con il corrispondente parametro.

Tenendo conto delle conversioni implicite:

1. corrispondenze esatte;
2. promozioni;
3. conversioni standard
   (trasformazione lvalue + promozione/conversione standard + qualificazione);
4. conversioni definite dall'utente
   (sequenza conversione std + conversione utente + sequenza conversione std)

```
double d = 3.1415;
void foo(int) {};

foo(d); // conversione standard (`double` to `int`)
```

Va posta attenzione anche:

- ai possibili valori di default per i parametri;
- all'argomento implicito (`this`) nelle chiamate dei metodi statici;
- al fatto che una funzione `private` o `public` o `protected`, ha sempre possibilita' di essere scelta, indipendentemente dalla sua visibilita' nella classe.

Fase3: la migliore utilizzabile

Le funzioni utilizzabili possono essere:

- $n = 0$, quindi ritornare errore;
- $n = 1$, l'unica utilizzabile è la migliore;
- $n > 1$, se la classificazione porta a un solo metodo migliore, allora questo verrà chiamato, altrimenti verrà restituito errore, perché troppa ambiguità.

E' come se le nostre funzioni, stessero gareggiando: per decidere se la funzione x e' preferibile rispetto y, si confrontano entrambe su tutte le classifiche corrispondenti degli m argomenti. x e' preferibile a y se:

- non perde in nessuno degli m confronti;
- vince almeno in un confronto.

15/03/2023