11 - Risoluzione dell'overloading

L'overloading in C++ è una caratteristica che permette di definire più funzioni con lo stesso nome, distinguendole in base al numero e/o al tipo dei loro parametri, particolarmente utile per mantenere leggibile e coerente il codice quando si implementano funzionalità simili per tipi diversi.
cpp
float sqrt(float arg);
double sqrt(double arg);
long sqrt(long arg);

Il compilatore si occupa di scegliere automaticamente quale versione invocare in base al tipo dell'argomento.
Anche gli operatori supportano l'overloading, il che rende possibile estendere questi meccanismi anche ai tipi definiti dall'utente.

Risoluzione dell'overloading

La risoluzione di una chiamata a funzione sovraccaricata avviene *staticamente*, ovvero a tempo di compilazione. Il compilatore segue un processo ben definito per determinare quale funzione chiamare:

1. *individuazione delle funzioni candidate*: si selezionano tutte le funzioni con il nome appropriato e visibili nel punto di chiamata;
2. *selezione delle funzioni utilizzabili*: si verifica se, tra le funzioni candidate, esistono quelle i cui parametri possono accettare gli argomenti della chiamata, eventualmente applicando conversioni implicite;
3. *scelta della migliore funzione utilizzabile*: si confrontano le funzioni utilizzabili per stabilire quale richiede il minor numero di conversioni o quelle meno invasive (come promozioni rispetto a conversioni standard o definite dall'utente).

Fase 1: Funzioni candidate

L'insieme delle funzioni candidate è costituito dalle funzioni dichiarate all'interno dell'unità di traduzione che soddisfano questi requisiti:

- **nome corrispondente**: la funzione deve avere lo stesso nome della funzione chiamata.
  - tenere presente che per gli operatori la sintassi della chiamata di funzione può variare:
    - *sintassi operatore*:
      - operatore prefisso `- -r`
      - operatore infisso `r1 + r2`
      - operatore postfisso `vect[5]`
    - *sintassi funzionale*:
      - `r.operator++()`
      - `operator+(r1, r2)`
      - `vect.operator[](5)`
- **visibilità**: la funzione deve essere visibile nel punto della chiamata.
  - *chiamate a metodi di una classe*: se un metodo è chiamato tramite `obj.metodo()` (o `ptr->metodo()`), la ricerca avviene nello scope della classe del tipo **statico** dell'oggetto/puntatore.

```
struct S { void foo(); };
struct T : public S { void foo(int); };

S* ptr = new T;
ptr->foo(); // Si cerca in S, non in T.
```

- *funzioni con qualificatore*: se una funzione viene chiamata usando un qualificatore (`namespace::funzione`), la ricerca inizia nello scope del namespace qualificato.

```
namespace N { void foo(int); }
void foo(char);

int main() {
    N::foo(42); // La funzione globale foo(char) non è visibile.
}
```

- *hiding (nascosta dall'overloading)*: una funzione dichiarata in una classe derivata può nascondere una funzione con lo stesso nome nella classe base,

```
struct S { void foo(int); };
struct T : public S { void foo(char); };

T t;
t.foo(5); // Errore: la funzione S::foo(int) è nascosta.
```

  a meno che non venga esposta esplicitamento con `using`

```
struct T : public S {
    using S::foo;
    void foo(char);
};

T t;
t.foo(5); // Funziona: le due funzioni vanno in overloading.
```

- **Argument Dependent Lookup (ADL)**: la regola ADL amplia la ricerca delle funzioni candidate, includendo i namespace associati ai tipi degli argomenti. Quindi
  - se la chiamata di funzione non è qualificata;
  - se vi sono (uno o più) argomenti di un tipo definito dall'utente (cioè hanno tipo struct/classe/enum S, o riferimento a S o puntatore a S, possibilmente qualificati);
  - se il tipo suddetto è definito nel namespace N
    ⇒ allora la ricerca delle candidate viene effettuata anche all'interno del namespace N.

```
namespace N {
    struct S {};
    void foo(S);
}

int main() {
```

```
    N::S s;
    foo(s); // Trova automaticamente N::foo grazie all'ADL.
}
```

Fase 2: Funzioni utilizzabili

Una funzione candidata è **utilizzabile** se soddisfa entrambe queste condizioni:

- **numero degli argomenti**: il numero degli argomenti nella chiamata deve corrispondere a quello dei parametri della funzione, considerando anche:
  - parametri con valori di default;
  - l'argomento implicito `this` nei metodi non statici.
- **compatibilità dei tipi**: ogni argomento della chiamata deve poter essere convertito nel tipo del corrispondente parametro della funzione. Le conversioni implicite ammesse includono:
  - **conversioni esatte** (identità, qualificazioni, trasformazioni di lvalue);
  - **promozioni** (es. da `int` a `long`);
  - **conversioni standard** (es. da `float` a `double`);
  - **conversioni definite dall'utente** (operatori di conversioni o costruttori).

Fase 3: Funzione migliore

Se esistono più funzioni utilizzabili, il compilatore seleziona quella che richiede le conversioni meno invasive. Le regole per determinare la "migliore" funzione si basano sul *rank* delle conversioni richieste:

1. **conversioni esatte** (identità) sono preferite su tutto;
2. **promozioni** sono preferite sulle conversioni standard;
3. **conversioni standard** sono preferite alle conversioni definite dall'utente.
   Se due funzioni sono equivalenti per tutti gli argomenti, il compilatore genera un errore di ambiguità.

```
void foo(int);
void foo(double);

foo(42); // Match perfetto su foo(int), nessuna ambiguità.
foo(42.0); // Match perfetto su foo(double), nessuna ambiguità.
foo('a'); // Ambiguità: 'a' può essere promosso a int o convertito a double.
```

Osservazioni

- le regole viste non prendono in considerazione la presenza di funzioni candidate ottenute istanziando template di funzione;
- quando si scelgono le funzioni *candidate*, il numero e il tipo dei parametri non sono considerati;
- nel caso di invocazione di un metodo di una classe, il fatto che il metodo sia dichiarato con accesso `public`, `private` o `protected` non ha nessun impatto sul processo di risoluzione dell'overloading; infatti la miglior funzione utilizzabile ma non accessibile non viene sostituita da un'altra funzione utilizzabile e accessibile.

Esercizio

1. Mostrare il processo di risoluzione dell'overloading per le seguenti chiamate di funzione. Per ogni chiamata, indicare l'insieme delle funzioni candidate, l'insieme delle funzioni utilizzabili e, se esiste, la migliore funzione utilizzabile.

```
namespace NB {
    class D{};
} //namespace NB

namespace NA {
    class C{};

    void f(int i); //funzione #1
    void f(double d, C c = C()); //funzione #2

    void g(C c = C(), NB::D d = NB::D()); //funzione #3

    void h(C c); //funzione #4

    void test1() {
        f(2.0); //chiamata A
    }
} //namespace NA

namespace NB {
    void f(double d); //funzione #5

    void g(NA::C c = NA::C(), D d = D()); //funzione #6

    void h(NA::C c, D d); //funzione #7

    void test2(double d, NA::C c) {
        f(d); //chiamata B
        g(c); //chiamata C
        h(c); //chiamata D
    }
} //namespace NB

void f(NA::C c, NB::D d); //funzione #8

void test3(NA::C c, NB::D d) {
    f(1.0); //chiamata E

    g(); //chiamata F
    g(c); //chiamata G
    g(c,d); //chiamata H
}
```

CHIAMATA A:

- fun. candidate:
  - #1
  - #2
    //le altre funzioni con nome `f` non sono visibili

- fun. utilizzabili:
  - #1
  - #2
- fun. migliore:
  - #2

  CHIAMATA B:
- fun. candidate:
  - #5
- fun. utilizzabili:
  - #5
- fun. migliore:
  - #5

  CHIAMATA C:
- fun. candidate:
  - #6
  - #3
- fun. utilizzabili:
  - #6 //tecnicamente non è un match perfetto, perchè la chiamata passa un lvalue mentre la funzione richiede un rvalue (il parametro della funzione viene passato per valore non per riferimento)
  - #3
- fun. migliore:

  non esiste ⇒ ambiguità

  CHIAMATA D:
- fun. candidate:
  - #7
  - #4
- fun. utilizzabili:
  - #4
- fun. migliore:
  - #4

  CHIAMATA E: //non si applica la ADL perchè non contiene nessun tipo definito dall'utente
- fun. candidate:
  - #8
- fun. utilizzabili:

  non ha funzioni utilizzabili
- fun. migliore:

  non esiste

  CHIAMATA F: //le uniche funzioni g esistenti sono all'interno di namespace
- fun. candidate: non esiste
- fun. utilizzabili: non esiste
- fun. migliore: non esiste

  CHIAMATA G:
- fun. candidate:
  - #3

- fun. utilizzabili:
  - #3
- fun. migliore:
  - #3

  CHIAMATA H:
- fun. candidate:
  - #3
  - #6
- fun. utilizzabili:
  - #3
  - #6
- fun. migliore:

  non esiste ⇒ ambiguità

2. Mostrare il processo di risoluzione dell'overloading per le seguenti chiamate di funzione. Per ogni chiamata, indicare: l'insieme delle funzioni candidate; l'insieme delle funzioni utilizzabili; la migliore funzione utilizzabile (se esiste); il motivo di eventuali errori di compilazione.

```
#include <string>

namespace N {
    class C {
    private:
        std::string& first(); //funzione #1

    public:
        const std::string& first() const; //funzione #2

        std::string& last(); //funzione #3
        const std::string& last() const; //funzione #4

        C(const char*); //funzione #5
    }; //class C

    void print(const C&); //funzione #6
    std::string& f(int); //funzione #7

} //namespace N

class A {
public:
    explicit A(std::string&); //funzione #8
}; //class A

void print(const A&); //funzione #9

void f(N::C& c) //funzione #10
{
    const std::string& f1 = c.first(); //chiamata A
    std:string& f2 = c.first(); //chiamata B
    const std::string& l1 = c.last(); //chiamata C
```

```
    std::string& l2 = c.last(); //chiamata D
}

void f(const N::C& c) //funzione #11
{
    const std::string& f1 = c.first(); //chiamata E
    std::string& f2 = c.first(); //chiamata F
    const std::string& l1 = c.last(); //chiamata G
    std::string& l2 = c.last(); //chiamata H
}

int main() {
    N::C c("begin"); //chiamata I
    f(c); //chiamata L
    f("middle"); //chiamata M
    print("end"); //chiamata N
}
```

	f. candidate	**f. utilizzabili**	**f. migliore**	**motivo:**
A	#1, #2	none	none	`c.first()` invocherebbe la funzione #1, però è privata quindi non può essere utilizzata. Invocherebbe la #2, però `c` non è costante, quindi non è possibile invocarla
B	#1, #2	none	none	idem chiamata A
C	#3, #4	#3	#3	viene chiamata la #3 perché anche se `l1` è un riferimento costante ad una stringa, `c` non lo è
D	#3, #4	#3	#3	
E	#1, #2	#2	#2	`c` è un riferimento const, quindi è possibile chiamare la #2 marcata const
F	#1, #2	none	none	ragionamento simile alla E, viene restituito un riferimento const a stringa che non può essere convertito in riferimento non const
G	#3, #4	#4	#4	
H	#3, #4	none	none	stesso motivo della F
I	#5	#5	#5	
L	#10, #11, #7	#10, #11	#10	chiamiamo la #10 perchè la `c` creata nella chiamata I non è const
M	#10, #11	#10, #11	ambiguità	non è presente nessuna funzione utilizzabile, perchè le chiamate aspettano come argomento un oggetto di tipo `C`
N	#9	none	none	non è presente nessuna funzione utilizzabile, perchè le chiamate aspettano come argomento un oggetto di tipo `A`

12 - Conversioni implicite