03_Scope

Table of contents

Scope

Lo **scope**, o campo d'azione, è il punto del programma in cui una variabile è visibile.
Sapere dove il nome è visibile è importante, ci sono certi casi in cui vogliamo che nomi uguali siano visibili in uno stesso punto (come *overloading*), altri altrimenti.

Scope di `namespace`

Una dichiarazione di nome, che non sta in classe/struct/parametri di funzione, resta a scope di `namespace`.
Quando diciamo che il `namespace` globale esiste, stiamo dicendo qualcosa del tipo:

```
namespace N {
        void foo() { // foo() è visibile da quì fino alla fine di N
                bar(a); // ERROR: chi sono bar() e a ? Non li conosco
        }
        int a; // a è visibile fino alla fine di N
        void bar(int n) {
                a += n; // CORRECT
        }
}
```

Scope di blocco

Porzione di codice racchiusa in parentesi graffe.

- per funzione

    ```
    void foo()
            // codice
            {
                    // blocco1
            } // fine scope di blocco
    ```

- per blocco iterativo

    ```
    for (int i=0; i =/= 10; ++i) {
            // i in scope del blocco for()
    }
    ```

- per parametro di funzione

```
if (T* ptr = foo()) {
  // ptr quì visibile
} else {
  // ptr anche quì visibile
}
```

Scope di classe

> ⚠ `struct` vs `class`
>
> In C++ possiamo usare degli *specificatori di accesso* (`private`, `public`, `protected`). L'unica differenza tra i due è che lo `struct` è di default `public`, mentre `class` ha accesso `private`.

I membri di una classe (entità dichiarate all'interno della stessa) sono visibili all'interno dell'intera classe, indipendentemente da dove vengano dichiarati.

```
struct S {
        void foo() {
                bar(a); // CORRECT: bar() e a sono visibili ovunque nella classe
        }
        int a;
        void bar(int n) { a += n; }
};
```

> Modi di referenziare oggetti nella classe
>
> - usando l'operatore punto `.` `s.foo()`
> - usando puntatore `->` `ps -> foo()`
> - usando l'operatore di scope `::` `S::foo()`

Scope di funzione

Lo scope di funzione è diverso dallo scope di blocco. Un esempio è l'operatore `goto`, visibile ovunque nella funzione.

```
void foo() { // scope di funzione
        { // scope di blocco
                inizio:
                        // codice
                        goto fine;
        }
        fine:
                // altro codice
                goto inizio;
}
```

Scope delle costanti di enumerazione

Come dichiarate per C++ 2003

```
enum Colors { red, blue, green };
```

hanno come scope quello del corrispondente tipo di enumerazione `Colors` (sono visibili fuori dalle parentesi graffe). Questo potrebbe dare problema di *name clash*: 2 oggetti si chiamano `red`, uno con valore diverso

dall'altro, che rende la leggibilità pessima.

```
enum Colors { red, blue, green };
enum StopLight { red, green, blue };

void foo() { std::cout << red; } // ERROR: a quale 'red' mi devo riferire?
```

Con lo standard vecchio, per ovviare al problema, si usava introdurre `namespace` sulla nostra costante di enumerazione:

```
namespace Colors { enum Colors_Enum { red, blue, gree }; }
```

Una variante introdotta con C++ 2011, risolve il problema senza rimuovere il supporto per la precedente usando un cast esplicito su `red`:

```
enum class Colors { red, blue, green };

void foo() {
        std::cout << static_cast<int>(Colors::red);
}
```

Hiding

Anche a nome "mascheramento", se ne parla quando una dichiarazione nello scope interno nasconde un'altra dichiarazione con nome uguale dello scope esterno.

```
int a = 1; // globale esterna
int main() {
        std::cout << a << std::endl; // 1
        int a = 5; // interna
        std::cout << a << std::endl; // 5
}
```

Si può avere anche per i membri ereditati da una classe, perché lo scope della classe derivata è considerato essere incluso nello scope della classe base:

```
struct Base {
        int a;
        void foo(int);
};
struct Derived : public Base {
        double a; // hiding di Base::a
        void foo(double d); // hiding del metodo Base::foo()
};
```

Estensioni della visibilità del nome

Dichiarare funzioni con lo stesso nome non sempre è un problema, ma lo può essere

```
// quali di queste funzioni usare in uno scope?
Struct Base {
        void foo(int);
        void foo(float);
};
```

La nostra `struct` ha 2 metodi che possono essere utilizzati, ma che con la derivazione della classe non vengono presi in considerazione (niente overloading ma hiding presente). Se volessimo prenderle, e tenerle nel caso ci servissero in secondo momento, usiamo:

```
struct Derived : public Base {
        using Base::foo;       // ora visibili tutti i metodi foo()
        void foo(double d);    // overloading con foo(int) e foo(float)
}
```

Direttive di `using`

```
void foo() {
        using namespace std;
        cout << "Hello" << endl;
}
```

Nella definizione di `foo()`, il compilatore ha la possibilità di guardare dentro il `namespace std`: quando trova il nome, se dichiarato, lo usa, altrimenti usa la direttiva di `using`. Nell'esempio sotto, viene stampato anche 42 anziché "Hello" e basta.

```
#include <iostream>
void foo() {
        int endl = 42;
        using namespace std;
        cout << "Hello" << endl;
}
```

28/02/2023