Scope (campo d'azione)

Entità

Ogni dichiarazione presente in una unità di traduzione introduce un nome per una entità.

Tale nome può essere utilizzato solo in alcuni punti dell'unità di traduzione: le porzioni di codice in cui il nome è "visibile" sono dette essere il *campo di azione* (in inglese, **scope**) per quel nome. L'ampiezza dello scope per un nome varia a seconda della tipologia di dichiarazione e del contesto in cui questa appare. Si distinguono diverse tipologie di scope.

Torna all'indice

Scope di Namespace (incluso lo scope globale)

I namespaces sono utilizzati per organizzare il codice in gruppi logici, allo scopo di evitare conflitti di nomi e migliorare la leggibilità del codice. Una dichiarazione che non è racchiusa all'interno di una `struct`/`class` e/o all'interno di una funzione ha scope di `namespace`.

> Si noti che lo scope globale è anche esso uno scope di `namespace` (al quale ci si può riferire usando il qualificatore di scope `::`).

Il nome è visibile, all'interno di quel `namespace`, a partire dal punto di dichiarazione e fino al termine dell'unità di traduzione (in particolare, NON è visibile prima del punto di dichiarazione).

In sostanza, questo è il motivo per il quale le inclusioni degli header file sono collocate all'inizio dei file sorgente.

```
namespace N {
    void foo() {
        // ERRORI: `bar` e `a` non sono visibili in questo punto
        // (vengono dichiarate dopo)
        bar(a);
    }

    int a; // definizione di 'a'
    void bar(int n) {
        a += n;
        // OK: 'a' è visibile in questo punto (dichiarata prima)
        // della funzione 'bar'
    }
} // ! namespace N
```

Torna all'indice

Scope di Blocco

Un nome dichiarato in un *blocco* (porzione di codice all'interno del corpo di una funzione racchiusa tra parentesi graffe) è locale a quel blocco.

Anche in questo caso, la visibilità inizia dal punto di dichiarazione e termina alla fine del blocco.

```
void foo() {
    // ...
    { // inizio blocco

        // ...
        int j /* inizio scope di blocco per j */ = expr;
        // ...
        std::cout << j;
        // ...
```

```
    } // fine dello scope di blocco per j
}
```

Vi sono alcune regole speciali per i costrutti `for`, `while`, `if`, `switch` e per i blocchi `try`/`catch`:

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

if (T* ptr = foo()) {
    // ptr è visibile qui (e vale ptr != nullptr)
} else {
    // ptr è visibile anche qui (e vale ptr == nullptr)
}
```

> E se avessimo dichiarato `ptr` fuori dal costrutto `if`? Avrebbe portato ad una estensione dello scope del puntatore, aumentando la possibilità di errori e/o comportamenti non voluti dal programmatore.

```
switch (int c = bar()) {
    case 0:
        break;

    case 1:
        do_something(c); // c è visibile qui
        break;

    case 2:
        do_something_different();
        break;

    default:
        std::cerr << "unexpected value c = " << c;
        break;
}

try {
    int a = 5;
    // ...
}
catch (const std::string& s) {
    std::cerr << s; // 's' è visibile qui
    // ATTENZIONE: 'a' NON è visibile qui
}
```

Torna all'indice

Scope di Classe

Qual è la differenza tra `struct` e `class`? Che le classi garantiscono l**'information hiding** (l'utente può gestire metodi e attributi con `public` e `private`), mentre le struct no. Per le struct la visibilità di default è *public* mentre nelle classi è *private*.

I membri di una classe (tipi, dati, metodi) sono visibili all'interno della classe indipendentemente dal punto di dichiarazione.

```
struct S {
    void foo() {
        bar(a); // OK: 'bar' e 'a' sono visibili anche se dichiarati dopo
    }

    int a;
    void bar(int n) { a += n; }
};
```

I membri di una classe posso essere acceduti dall'esterno della classe nei modi seguenti:

```
    s.foo();   // usando l'operatore punto, se `s` ha tipo (riferimento a) `S`
    ps->foo(); // usando l'operatore freccia, se `ps` ha tipo puntatore a `S`
    S::foo;    // usando l'operatore di scope
```

I membri di una classe `S` possono essere acceduti anche da classi che sono derivate (anche indirettamente) dalla classe `S` (in quanto sono ereditati dalle classi derivate). In caso di *overloading* di metodi si può accedere a quelli della classe base usando il risolutore di scope `::`.

Torna all'indice

Scope di Funzione

Le etichette (*label*) di destinazione delle istruzioni `goto` hanno scope di funzione: sono visibili in tutta la funzione che le racchiude, indipendentemente dai blocchi.

```
void foo() {
    int i;
    {
        inizio: // visibile anche fuori dal blocco

        i = 1;
        while (true) {
            // ...
            if (condizione)
                goto fine; // fine è visibile anche se dichiarata dopo
        }
    }

    fine:

    if (i > 100)
        goto inizio;
    return i;
}
```

L'uso dei `goto` e delle etichette è considerato cattivo stile e andrebbe limitato ai casi (pochissimi) in cui risultano essenziali.

Torna all'indice

Scope delle costanti di enumerazione: un caso speciale.

Le costanti di enumerazione dichiarate secondo lo stile C++ 2003.

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

Hanno come scope quello del corrispondente tipo enumerazione `Colors` (ovvero, sono visibili "fuori" dalle graffe che le racchiudono).

Questo può causare problemi di conflitto di nomi:

```
enum Colori { rosso, blu, verde };
enum Semaforo { verde, giallo, rosso };

void foo() { std::cout << rosso; } // a quale rosso si riferisce?
```

Nel C++ 2011 sono state introdotte le enum class, che invece limitano lo scope come le classi, costringendo il programmatore a qualificare il nome e evitando potenziali errori:

```
enum class Colori { rosso, blu, verde };
enum class Semaforo { verde, giallo, rosso };

void foo() {
  std::cout << static_cast(Colori::rosso);
}
```

> Il cast è necessario perché le `enum class` impediscono anche le conversioni implicite di tipo verso gli interi.

Torna all'indice

Riduzioni ed estensioni dello scope di una dichiarazione

Quello introdotto precedentemente è il cosiddetto scope potenziale di una dichiarazione.
Lo scope potenziale può essere modificato da alcuni costrutti del linguaggio.

Hiding di un nome

Quando si annidano campi di azione, è possibile che una dichiarazione nello scope interno nasconda un'altra dichiarazione (con lo stesso nome) dello scope esterno.
Si parla di hiding di un nome.

```
int a = 1; // scope globale

int main() {
 std::cout << a << std::endl;   // stampa 1
 int a = 5;
 std::cout << a << std::endl;   // stampa 5
 {
  int a = 10; // la 'a' esterna viene nascosta
     std::cout << a << std::endl; // stampa 10
 } // lo scope della 'a' esterna riprende da questo punto
 std::cout << a << std::endl;   // stampa 5
}
```

Si può avere hiding 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 del data member Base::a
 void foo(double d); // hiding del metodo Base::foo()
};
```

Torna all'indice

Estensioni della visibilità di un nome

Per accedere ad un nome dichiarato in uno scope differente, è spesso possibile utilizzare la versione qualificata del nome. Per esempio, dentro la classe `Derived` vista sopra, si può accedere ai dati e ai metodi della classe `Base` scrivendo `Base::a` e `Base::foo`. Lo stesso dicasi nel caso dello scope di namespace (si ricordi l'uso di `std::cout`).

Se però un nome deve essere utilizzato molto spesso in una posizione in cui non è visibile senza qualificazione, può essere scomodo doverlo qualificare in ogni suo singolo uso.
Per evitare ciò, si possono usare le dichiarazioni di using (*using declaration*):

```
void foo() {
 using std::cout;
 using std::endl;
 cout << "Hello" << endl;
 cout << "Hello, again" << endl;
 cout << "Hello, again and again and again ..." << endl;
}
```

Nota

Una dichiarazione di *using* può rendere disponibili solo nomi che erano stati precedentemente dichiarati (o resi visibili) nel namespace indicato.

In particolare, nel caso precedente, è comunque necessario includere l'header file `iostream`, altrimenti si ottiene un errore.

La dichiarazione di *using* rende disponibile (nel contesto in cui viene inserita) il nome riferito, che da lì in poi potrà essere usato senza qualificazione. Chiaramente, nel caso di un nome di tipo o di una variabile, è necessario che nello stesso contesto *NON* sia già presente un'altra entità con lo stesso nome.

```
void foo() {
 int cout = 5;
 using std::cout; // error: 'cout' is already declared in this scope
}
```

La cosa è invece legittima nel caso di funzioni, perché in quel caso entra in gioco il meccanismo dell'overloading. Nell'esempio seguente, la dichiarazione di *using* crea l'overloading per i metodi di nome `foo` (evitando l'*hiding*):

```
struct Base {
 void foo(int);
 void foo(float);
};

struct Derived : public Base {
 // rendo visibili in questo scope tutti i metodi
 // di nome "foo" presenti in Base
 using Base::foo;

 // foo(double) va in overloading con foo(int) e foo(float)
 void foo(double d);
};
```

Torna all'indice

Direttive using

Cosa ben distinta rispetto alle dichiarazioni di using sono le direttive di using (*using directive*). La sintassi è la seguente:

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

La direttiva di using *NON* introduce dichiarazioni nel punto in cui viene usata; piuttosto, aggiunge il namespace indicato tra gli scope nei quali è possibile cercare un nome per il quale *NON* si trovino dichiarazioni nello scope corrente.

Per capire la differenza, consideriamo l'esempio seguente:

```
#include

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

Vedendo l'uso del nome `cout`, il compilatore lo cerca nello scope corrente (il blocco della funzione).
Non trovandolo, continua la ricerca negli scope che racchiudono la funzione `foo` e, grazie alla direttiva di *using*, anche nello scope del `namespace std` (trovandolo).

Vedendo l'uso del nome `endl`, il compilatore lo cerca nello scope corrente e trova la dichiarazione della variabile intera, completando la ricerca. La direttiva di *using* in questo caso *NON* entra in gioco e la funzione stamperà la stringa "Hello42" (senza andare a capo).

Torna all'indice