3 - Scope

Le dichiarazioni introducono un nome per un entità, questo è **visibile** però solo in alcuni punti dell'unità di traduzione, la parte in cui è visibile viene chiamato **campo di azione (scope)**.
Tipologie:

- scope di namespace (incluso lo scope globale):
  - dichiarazione non racchiusa all'interno di una struct/class/enum o all'interno di una funzione
  - visibile dal punto di dichiarazione fino al termine dell'unità di traduzione, non visibile prima del punto di dichiarazione
- scope di blocco (codice racchiuso fra parentesi graffe):
  - un nome dichiarato in un blocco è locale a quel blocco
  - visibilità dal punto di dichiarazione fino alla fine del blocco
  - regole speciali per `for`, `while`, `if`, `switch`, `catch`; variabili dichiarate all'interno di questi costrutti hanno visibilità solo all'interno di essi.
- scope di classe:
  - i dati membro e i metodi di una classe/struct sono visibili all'interno della classe indipendentemente dal punto di dichiarazione, in modo tale da consentire la definizione di metodi inline.
  - per i tipi membro, valgono le regole dello scope di blocco
  - i membri di una classe possono essere acceduti da classi che sono derivate (perche ereditati) e possono essere acceduti dall'esterno:
    `s.foo(); //se s ha tipo S`
    `ps->foo(); //se ps ha tipo puntatore a S`
    `S::foo(); //operatore di scope`
    differenza tra class e struct:
    quasi nessuna differenza, soltanto l'accesso ai metodi e ai dati membro, in particolare quando non vengono specificati gli access specified (public, private) nelle classi di default è tutto privato, mentre nelle struct di default è tutto pubblico.

- scope di funzione: riguarda soltanto le etichette (label) del `goto`

```
void foo() {
  int i;
  {
  inizio: // visibile anche fuori dal blocco
    i = 1;
    while (true) {
      // ... calcoli ...
      if (condizione)
        goto fine; // visibile anche se dichiarata dopo
    }
  }
fine:
  if (i > 100)
    goto inizio;
  return i;
}
```

- scope delle costanti di enumerazione: con c++11 le `enum class` adottano le regole di scope delle classi, necessitando della qualificazione del nome, per non creare ambiguità, e del cast (necessario perche le enum class impediscono anche le conversioni implicite di tipo verso gli interi).

```
enum class Colori { rosso , blu , verde };
enum class Semaforo { verde , giallo , rosso };
void foo () {
std :: cout << static_cast<int>( Colori :: rosso );
}
```

Scope potenziale e Scope effettivo

Scope potenziale: come si comporterebbe se il programmatore non facesse nulla per andare a modificarlo (ridurlo o estenderlo).
Questo scope può essere modificato e diventa cosi **scope effettivo**.
Modifiche dello scope:

- **name hiding**: scope diversi vengono annidati e una dichiarazione nello scope interno può nascondere un'altra dichiarazione (con lo stesso nome) dello scope esterno. Si può avere hiding anche per i membri ereditati da una classe, perché lo scope della classe derivata è considerato incluso nello scope della classe base.
- **uso di nomi qualificati**: accesso ad alcune entità al di fuori del loro scope può essere ottenuto usando nomi qualificati (`std:endl`). La qualificazione può essere: parziale, punto di partenza lo scope corrente; totale, punto di partenza lo scope globale (FQN, Fully Qualified Name)
- **ADL: (Argument Dependent Lookup)**:
  in una chiamata di funzione `foo(.., arg, ..)` se il nome della funzione (`foo`) non è qualificato e se uno degli argomenti (`arg`) della chiamata è di un tipo dato `N::U` definito dall'utente all'interno del namespace N, allora si considerano come candidate tutte le funzioni con lo stesso nome dichiarate all'interno dello stesso namespace (cioè `N:foo`)

- **dichiarazioni e direttive using**:
  - se un nome deve essere utilizzato spesso in una posizioni in cui non è visibile si utilizza la **using declaration** (`using std::cout`). Nel caso di una using declaration per un tipo o di una variabile, è necessario che nello stesso scopo non sia già presente un'altra entità con lo stesso nome. Questa cosa invece è legittima in caso di funzioni → **overloading**.
  - **using directive**: non introducce dichiarazioni nel punto in cui viene usata, ma aggiunge il namespace indicato tra gli scope nei quali è possibile cercare un nome (`using namespace std;`).

  N. B. :
- preferire le using declaration rispetto alle using directive, perche introducono meno nomi.
- limitare al massimo lo scope delle using declaration / directive
- non usarle a scope di namespace/globale, in particolare in un header file