# 7 - One Definition Rule (ODR)

Un programma è suddiviso in più unità di traduzione (compilazione separata) che per interagire correttamente devono concordare su un interfaccia comune, per ridurre il rischio di inconsistenza dell'interfaccia si segue la regola **DRY** (Don't Repeat Yourself):
le dichiarazioni dell'interfaccia vengono scritte una volta sola, in uno o più header file
le unità di traduzione includono gli header file di cui hanno bisogno (senza ripeterle).
La **ODR** è una regola fondamentale che stabilisce come gestire correttamente definizioni e dichiarazioni all'interno di un programma. Serve a garantire che il comportamento del programma sia deterministico ed evita ambiguità o errori durante la compilazione e il linking.

> recall:
> **linker**: strumento che si occupa di collegare insieme tutti i file oggetto per creare il programma finale eseguibile.

**ODR**

La ODR stabilisce:

1. ogni **unità di traduzione** che forma un programma può contenere *non più di una definizione* di una data variabile, funzione, classe, enumerazione o template;
2. ogni **programma** deve contenere *esattamente una definizione* di ogni variabile e di ogni funzione non-inline usate nel programma;
3. ogni **funzione inline** deve essere definita in ogni unità di traduzione che la utilizza;
4. in un **programma** vi possono essere *più definizioni* di una classe, enumerazione funzione inline, template di classe e template di funzione (in unità di traduzione diverse, come stabilito nel punto 1) a condizione che:
   4a. tali definizioni siano sintatticamente identiche;
   4b. tali definizioni siano semanticamente identiche.

## Dettagli

### Punto 1: definizione multipla di tipo in una unità di traduzione

Un'unità di traduzione (file sorgente .cc e tutti gli header file inclusi) può contenere una sola definizione di una data variabile, funzione, classe, ecc. Esempio di violazione:

```
struct S { int a; };
struct S { char c; double d; };
```

Esempio corretto:

```
struct S { int a; }; //una sola definizione
extern S s; //dichiarazione della variabile
S s; //definizione della variabile
```

### Punto 2: una definizione per variabili e funzioni globali

- caso banale: uso di variabile / funzione che è stata dichiarata ma non è mai stata definita, la compilazione in senso stretto va a buon fine ma verrà segnalato un errore al momento di generare il codice eseguibile
- definizioni multiple (a volte incosistenti) in traduzioni diverse:

```
//foo.hh
int foo(int a);

//file1.cc
#include "foo.hh"
int foo(int a) { return a + 1; }

//file2.cc
#include "foo.hh"
int foo(int a) { return a + 2; }

//file3.cc
#include "foo.hh"
int bar(int a) { return foo(a); }
```

Questo caso porta a errori di linking o comportamenti imprevedibili, perchè il linker potrebbe usare una definizione o l'altra (tra quella del file1 e quella del file2) senza avvisare.

### Punto 3: definizioni di funzioni inline

Le funzioni inline devono essere definite in ogni unità di traduzione in cui vengono chiamate. Questo è necessario perchè il compilatore sostituisce la chiamata alla funzione con il corpo della funzione stessa durante la compilazione; infatti è per questo che conviene scriverne il corpo una sola volta in un header file.

## Violazioni più avanzate

### Punto 4: definizioni sintatticamente o semanticamente diverse

- *sintatticamente diversa*: due definizioni di una classe o tipo hanno una struttura diversa. Esempio:

```
// file1.cc
struct S { int a; int b; };  // Definizione 1

// file2.cc
struct S { int b; int a; };  // Definizione 2 (ordine diverso)
```

- *semanticamente diversa*: anche se sintatticamente identiche, le definizioni hanno un significato diverso. Esempio:

```
// file1.cc
typedef int T;
struct S { T a; T b; };
```

```cpp
// file2.cc
typedef double T;  // Cambia il significato di T
struct S { T a; T b; };  // Sintassi identica, semantica diversa
```

## Risolvere le violazioni della ODR

Per evitare errori legati alla ODR, si seguono queste pratiche:

1. **usare gli header file per dichiarazioni condivise**: tutte le dichiarazioni comuni (classi, funzioni, tipi) devono essere messe in un unico header file, incluso dove necessario.
2. **proteggere gli header file con guardie contro le inclusioni multiple**:
   Si utilizzano:
   - **guardie del preprocessore**:

     ```cpp
     #ifndef HEADER_NAME
     #define HEADER_NAME
     // Contenuto dell'header
     #endif
     ```

     esempio senza guardie:

     ```cpp
     // header.h
     struct MyStruct {
         int a;
     };

     // file1.cpp
     #include "header.h"

     // file2.cpp
     #include "header.h"

     // main.cpp
     #include "header.h"  // Incluso direttamente
     #include "file1.cpp" // header.h viene incluso di nuovo!
     #include "file2.cpp" // header.h viene incluso ancora una volta!
     ```

     risultato: quando il compilatore processa il file main.cpp, troverà più definizioni di `MyStruct` e segnalerà un errore.
   - oppure `#pragma once`: più semplice, ma non standard.
3. **non definire variabili o funzioni negli header file**: negli header file si mettono solo dichiarazioni, mentre le definizioni vanno nei file .cc .
4. **seguire la regola DRY**: evitare duplicazioni di definizioni, utilizzando gli stessi header file in tutte le unità di traduzione.

## Costrutti ammessi e vietati negli header file

**Ammessi**:

- dichiarazioni di variabili, funzioni, classi, template;

- definizioni di funzioni inline o template;
- alias di tipi ( `typedef` o `using` );
- namespace nominati.
  **Vietati**:
- definizioni di variabili e funzioni non-inline;
- namespace anonimi;
- `using namespace` (potrebbe creare conflitti di nomi).

## Esempio pratico:

Consideriamo un programma che deve effettuare calcoli matematici e che necessita di usare:

- una classe per i numeri razionali;
- una classe per i polinomi a coefficienti razionali;
- una funzione che usa sia i razionali, sia i polinomi;
  **Razionale.hh**:

```
class Razionale {
    //definizione della classe
};
```

**Polinomio.hh**:

```
#include "Razionale.hh"
class Polinomio {
    //definizione della classe, usa Razionale
}
```

**Calcolo.cc**:

```
#include "Razionale.hh"
#include "Polinomio.hh" //violazione ODR punto 1
Razionale valuta(const Polinomio& p, const Razionale& r) {
    //calcola il valore di p in x
}
```

**Analisi dell'errore**: quando viene compilata l'unità di traduzione corrispondente al file sorgente `Calcolo.cc` si viola la ODR perché l'unità conterrà due definizione della classe `Razionale` :

- la prima dalla direttiva di inclusione;
- la seconda è ottenuta, indirettamente, dalla seconda direttiva di inclusione `#include "Polinomio.hh` .
  Se provassimo a modificare il file `Calcolo.cc` , per risolvere il problema, eliminando l'inclusione di `Razionale.hh` :
- viene a crearsi una *dipendenza indiretta* (nascosta);
- diminuirebbe la leggibilità del codice;
- verrebbe a complicarsi la sua manutenzione.
  La soluzione corretta sarebbe utilizzare le direttive del processore che impediscono di includere

l'header file più di una volta, come `#pragma once`, oppure le guardie contro l'inclusione ripetuta.
Quindi:
**Razionale.hh**:

```
#ifndef RAZIONALE_HH
#define RAZIONALE_HH

class Razionale {
  // Definizione della classe
};

#endif
```

**Polinomio.hh**

```
#ifndef POLINOMIO_HH
#define POLINOMIO_HH

#include "Razionale.hh"

class Polinomio {
  // Definizione della classe, usa Razionale
};

#endif
```

**Calcolo.cc**

```
#include "Razionale.hh"
#include "Polinomio.hh"

// Funzioni e codice che usano Polinomio e Razionale
```

8 - Passaggio argomenti