ODR (One Definiton Rule)

Enea Zaffanella

Metodologie di programmazione
Laurea triennale in Informatica
Università di Parma
enea.zaffanella@unipr.it

ODR: motivazione

- il codice di un programma è suddiviso in più unità di traduzione (compilazione separata)
- per interagire correttamente, le unità di traduzione devono concordare su una interfaccia comune, formata da dichiarazioni di tipi, variabili, funzioni, ecc.
- per ridurre il rischio di inconsistenza dell'interfaccia, si cerca di seguire la regola **DRY** ("Don't Repeat Yourself"), nota anche come "Write Once":
  - 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 ripetere le corrispondenti dichiarazioni)
- meccanismo intuitivamente semplice, ma se usato senza cautela può dare luogo a problemi che, in ultima analisi, sono violazioni della "One Definition Rule"

ODR: definizione

La ODR (regola della definizione unica) stabilisce quanto segue

- La ODR (regola della definizione unica) stabilisce quanto segue:
  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, in virtù del punto 1) a condizione che:
     (a) tali definizioni siano sintatticamente identiche
     (b) tali definizioni siano semanticamente identiche
- nota: quella descritta è una versione semplificata della regola, che omette vari casi speciali
- nel seguito si forniscono esemepi di violazioni della regola

Violazione del punto 1

- definizione multipla di tipo in una unità di traduzione

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

- definizione multipla di variabile in una unità di traduzione

```
int a;
int a;
```

- il motivo dell'errore è evidente: le due definizioni con lo stesso nome creano una ambiguità

Esempi che NON sono violazioni del punto 1

- nota bene: si considera il nome **completamente qualificato**, per cui la seguente **NON** è una violazione del punto 1:

```
namespace N { int a; }
int a;
```

  perché le variabili `N::a` e `::a` sono distinte

- analogamente, per le funzioni è lecito l'**overloading**, per cui anche la seguente **NON** è una violazione:

```
int incr(int a) { return a + 1; }
long incr(long a) { return a + 1; }
```

  perché le funzioni `int ::incr(int)` e `long ::incr(long)` sono distinte

Altri esempi che NON sono violazioni del punto 1

- notare che si parla di **definizioni**
- è lecito avere **più dichiarazioni** della stessa entità, a condizione che solo una di esse sia una definizione (le altre devono essere dichiarazioni pure)

```
struct S;              // dichiarazione pura
struct S { int a; };   // definizione
struct S;              // dichiarazione pura

S a;         // definizione
extern S a; // dichiarazione pura

void foo();         // dichiarazione pura
void foo() { }      // definizione
extern void foo(); // dichiarazione pura
```

Violazione del punto 2: caso banale

- uso di una variabile o funzione che è stata dichiarata ma non è stata mai definita nel programma (zero definizioni):
  - la compilazione in senso stretto andrà a buon fine
  - il linker segnalerà l'errore al momento di generare il codice eseguibile

Violazione del punto 2: caso più interessante

- definizioni multiple (magari pure inconsistenti) in unità di traduzione diverse

```
/* nel file foo.hh */
int foo(int a);

/* nel file foo1.cc */
#include "foo.hh"
int foo(int a) { return a + 1; }

/* nel file foo2.cc */
#include "foo.hh"
int foo(int a) { return a + 2; }

/* nel file user.cc */
#include "foo.hh"
int bar(int a) { return foo(a); }
```

- il linker forse segnalerà l'errore (lo standard dice "no diagnostic required")

Violazione del punto 3

- il punto 3 dice che le funzioni **inline** devono essere definite ovunque sono usate; il senso della regola è chiaro, se si è compreso il meccanismo dell'inlining delle funzioni, che prevede che la chiamata di funzione possa essere sostituita con l'espansione in linea del corpo della funzione (a scopo di ottimizzazione)
- l'espansione è effettuata durante la fase di compilazione in senso stretto, per cui il corpo della funzione deve essere presente in ogni unità di traduzione che contiene una chiamata

Una nota tecnica sul linker

- il linker deve:
  - segnalare come errore il caso di una funzione non-inline che è definita in più unità di traduzione
  - nella stessa situazione, non deve segnalare errore se la funzione è inline
- come fa a distinguere i due casi (considerato che vede solo il codice oggetto prodotto dalla compilazione)?

Una nota tecnica sul linker (ii)

- possiamo intuire la risposta osservando l'output del comando `nm`

```
/* File aaa.cc */
inline int funzione_inline() { return 42; }
int funzione_non_inline() { return 1 + funzione_inline(); }
```

- compilando (con l'opzione `-c`) e invocando `nm` sull'object file generato si ottiene:

```
$ nm -C aaa.o
0000000000000000 W funzione_inline()
0000000000000000 T funzione_non_inline()
```

- l'etichetta `W` (weak symbol) ci dice che la funzione inline è definita debolmente: possono esistere più definizioni e il linker può prenderne una qualunque, perché il programmatore garantisce che tutte le definizioni esistenti sono identiche

Violazione del punto 4a

- il punto 4 della ODR è il più interessante: si applica a classi, enumerazioni, funzione inline, template di classe e template di funzione, ma è sufficiente considerare le definizioni di classi
- esempio di violazione della regola 4a:

```
/* unita1.cc */
struct S { int a; int b; }; // def. di S
S s; // def. di variabile di tipo S

/* unita2.cc */
struct S { int b; int a; }; // def. di S  con sintassi diversa
extern S s; // dich. pura della s definita in file1.cc
```

Violazione del punto 4b

- piccola variante del'esempio precedente:

```
/* unita1.cc */
using T = int;
struct S { T a; T b; }; // def. di S

/* unita2.cc */
using T = double;
struct S { T a; T b; }; // stessa sintassi, semantica diversa
```

- il problema si può presentare anche a causa di un uso improprio delle macro del preprocessore (cioè, non dipende solo dall'uso degli alias di tipo)

Linee guida per soddisfare la ODR

- la linea guida principale, già nominata, è la DRY (Don't Repeat Yourself): scrivere una volta sola le dichiarazioni e/o definizioni negli header file e includere questi dove necessario
- questo approccio (da solo) non risolve tutti i problemi
- per convincerci, 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

Suddivisione del codice

```
/* Header file Razionale.hh */
class Razionale { /* codice */ };

/* Header file Polinomio.hh */
#include "Razionale.hh"
class Polinomio { /* codice (usa anche Razionale) */ };

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

Analisi dell'errore ottenuto

- quando si compila l'unità di traduzione corrispondente al file sorgente `Calcolo.cc`, si ottiene una violazione della ODR perché l'unità conterrà infatti **due** definizioni della classe `Razionale`:
  - la prima ottenuta dalla prima direttiva di inclusione
  - la seconda ottenuta (indirettamente) dalla seconda direttiva di inclusione
- quale è il modo corretto di risolvere questa situazione?
- un modo sicuramente **sbagliato** (dal punto di vista metodologico) è quello di modificare `Calcolo.cc`, eliminando l'inclusione di `Razionale.hh`:
  - crea un **dipendenza indiretta** (cioè nascosta)
  - diminuisce la leggibilità del codice
  - rende complicata la sua manutenzione

Soluzioni adeguate

- come detto, occorre consentire ad ogni unità di traduzione di includere tutti gli header file di cui necessita (cioè di avere **dipendenze esplicite**),
- soluzioni alternative del problema delle inclusioni multiple dello stesso header file:
  - usare direttive del processore (se supportate) che impediscono di includere l'header file più di una volta:

    ```
    #pragma once
    ```

  - le **guardie contro l'inclusione ripetuta** sono una soluzione più verbosa, ma sempre supportata

Uso delle guardie contro l'inclusione ripetuta

- l'header file `Razionale.hh` viene modificato in questo modo:

```
/* Header file Razionale.hh */
#ifndef RAZIONALE_HH_INCLUDE_GUARD
#define RAZIONALE_HH_INCLUDE_GUARD 1

class Razionale { /* codice */ };

#endif /* RAZIONALE_HH_INCLUDE_GUARD */
```

- nota bene: tutti gli header file devono essere modificati in modo analogo, facendo attenzione ad usare un nome distinto della guardia per ogni header file

Perché le guardie funzionano?

- la prima volta che l'header file viene incluso, la "guardia" (cioè il flag `RAZIONALE_HH_INCLUDE_GUARD`) NON è ancora definita e quindi il preprocessore procede con l'inclusione
- se capita, durante il preprocessing di quella stessa unità di traduzione, di ritentare altre volte l'inclusione di `Razionale.hh`, il preprocessore trova la guardia già definita; la condizione di `#ifndef` valuta a falso e quindi non avviene nessuna inclusione ripetuta
- nota: le guardie contro l'inclusione ripetuta sono utilizzate anche negli header file della libreria standard distribuiti con g++. Per esempio, nell'header file `iostream`:

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1
// ... codice
#endif /* _GLIBCXX_IOSTREAM */
```

Cosa è sensato trovare in un header file?

(si veda il Capitolo 15 del testo di Stroustrup)

- direttive del preprocessore (inclusione, guardie, macro, . . .)
- commenti
- dichiarazioni/definizioni di tipo
- dichiarazioni pure di variabili
- definizioni di costanti
- dichiarazioni pure di funzioni
- definizioni di funzioni inline
- dichiarazioni/definizioni di template
- namespace dotati di nome
- alias di tipi

Cosa NON si dovrebbe trovare in un header file?

(si veda il Capitolo 15 del testo di Stroustrup)

- definizione di variabili (non locali)
- definizione di funzioni (non inline)
- dichiarazioni di using (a namespace scope)
- direttive di using (a namespace scope)
- namespace anonimi