ODR

ODR: One Definition Rule

Quando il codice di un programma deve essere suddiviso in più unità di traduzione, si pone il problema di come fare interagire correttamente le varie porzioni del programma. Intuitivamente, le varie unità di traduzione devono concordare su una interfaccia comune. Tale interfaccia è formata quindi da dichiarazioni di tipi, variabili, funzioni, ecc.

Per ridurre il rischio che una delle unità di traduzione si trovi ad operare con una versione diversa (inconsistente) dell'interfaccia, si cerca di seguire la regola DRY ("Don't Repeat Yourself"): le dichiarazioni dell'interfaccia vengono scritte una volta sola, in uno o più header file.

Le varie unità di traduzione includeranno gli header file di cui hanno bisogno, evitando di ripetere le corrispondenti dichiarazioni. Il meccanismo è intuitivamente semplice, ma se usato senza la necessaria cautela, può dare luogo a problemi che, in ultima analisi, si possono ricondurre a violazioni della "One Definition Rule".

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 a condizione che:
    - tali definizioni siano sintatticamente identiche;
    - tali definizioni siano semanticamente identiche.

Torna all'indice

Esempi

Forniamo alcuni esempi per chiarire i vari punti della ODR; in particolare, ci concentreremo sulle possibili 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. Si noti che si considera il nome completamente qualificato, per cui la seguente NON è una violazione, perché si tratta di variabili diverse (`N::a` e `::a`).

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

Analogamente, per le funzioni è lecito l'overloading, per cui anche la seguente NON è una violazione, perché si tratta di funzioni diverse, `int ::incr(int)` e `long ::incr(long)`:

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

Si noti inoltre che si parla di definizioni. È lecito avere più dichiarazioni della stessa entità, a condizione che solo una di esse sia una definizione (ovvero, le altre devono essere dichiarazioni pure). L'esempio seguente NON contiene violazioni ODR:

```
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
```

Torna all'indice

Violazione del punto 2

Un caso banale è quello dell'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, ma il linker segnalerà l'errore al momento di generare il codice eseguibile.

Più interessante è il caso delle definizioni multiple (magari pure inconsistenti) effettuate in unità di traduzione diverse:

```
// Siamo in foo.hh

int foo(int a);

// Siamo in file1.cc

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

// Siamo in file2.cc

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

// Siamo in file3.cc

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

Il linker, al momento di collegare (l'object file prodotto dalla compilazione di) `file3.cc` con il resto del programma potrebbe indifferentemente invocare la funzione foo definita in `file1.cc` o quella definita in `file2.cc`, ottenendo effetti non prevedibili.

Tipicamente, il linker segnalerà l'errore, ma in realtà le regole del linguaggio dicono che non è tenuto a farlo (la formula usata nello standard è "no diagnostic required") e, nel caso, la colpa dell'errore ricade sul programmatore.

Torna all'indice

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). Tale 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.

> Per quanto detto, il linker deve segnalare come errore il caso di una funzione non-inline che è definita in più unità di traduzione, mentre non deve segnalare errore se la funzione è inline. Come fa a distinguere questi due casi? La risposta la possiamo intuire usando lo strumento `nm` che visualizza il contenuto di un object file.

```
// Siamo in 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, vediamo quanto segue:

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

La funzione non inline è marcata come simbolo definito (etichetta `T`). La funzione inline, invece, è marcata con l'etichetta `W` (weak symbol). Dal manuale di `nm`:

```
"W"
"w" The symbol is a weak symbol that has not been specifically tagged as a weak object symbol.
When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error.
When a weak undefined symbol is linked and the symbol is not defined, the value of the symbol is determined in a system-specific manner without error.
On some systems, uppercase indicates that a default value has been specified.
```

Torna all'indice

Violazione del punto 4

Il punto più interessante della ODR è il 4. Si applica a classi, enumerazioni, funzione inline, template di classe e template di funzione, ma per comprendere il problema è sufficiente considerare il caso delle definizioni di classi.

Esempio di violazione della 4a

```
// Siamo in file1.cc

struct S {
 int a;
 int b;
};     // definizione del tipo S
S s;          // definizione di variabile di tipo S

// Siamo in file2.cc

struct S { int b; int a; }; // definizione del tipo S (sintassi diversa)
extern S s;  // dichiarazione pura della s definita in file1.cc
```

Quando il compilatore lavora su `file2.cc`, viene ingannato dalla diversa definizione del tipo `S`, ma non se ne può accorgere. Non è nemmeno detto che se ne accorga il linker (in ogni caso, non è tenuto a farlo).

Con una piccola variante, è possibile ottenere una violazione della 4b, nella quale i tipi sono sintatticamente identici, ma semanticamente diversi:

Esempio di violazione della 4b

```
// Siamo in file1.cc

typedef T int;
struct S { T a; T b; }; // definizione del tipo S

// Siamo in file2.cc

typedef T double;
struct S {
 T a;
 T b;
};       // definizione del tipo S
         // (sintassi identica, ma semantica diversa)
```

Torna all'indice

Il tipo Razionale

Avendo analizzato le clausole che compongono la ODR, rimane da capire cosa va fatto all'atto pratico per garantirne la soddisfazione.

La linea guida principale, già nominata, è la DRY (don't repeat yourself), a volte detta anche *"write once"*: scrivere una volta sola le dichiarazioni e/o definizioni, inserendole negli header file, e includere questi dove necessario.

L'esempio seguente mostra che l'approccio non risolve tutti i problemi.

Supponiamo di avere un programma che effettua calcoli matematici. Il programma necessita di utilizzare numeri razionali, per i quali si crea un tipo apposito (`Razionale`), la cui definizione è messa nell'header file `Razionale.hh`:

```
// Siamo in Razionale.hh

class Razionale {
```

```
  // ... le cose giuste
};
```

Una seconda parte del programma deve gestire polinomi a coefficienti razionali, per cui si crea un altro header file:

```
// Siamo in Polinomio.hh

#include "Razionale.hh"
class Polinomio {
  // ... le cose giuste, usando anche Razionale
};
```

In una terza parte del programma, si deve scrivere un algoritmo che effettua calcoli sui polinomi, usando anche i razionali:

```
// Siamo in Calcolo.cc

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

// Codice che usa i tipi Polinomio e Razionale
```

Quando compileremo (l'unità di traduzione corrispondente a) `Calcolo.cc`, otterremo un errore dovuto alla violazione della clausola 1 della ODR. L'unità conterrà infatti *due* definizioni della classe `Razionale`, la prima ottenuta dalla prima direttiva di inclusione e la seconda ottenuta (indirettamente) dalla seconda direttiva di inclusione.

Torna all'indice

Qual 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`:

```
// Siamo in Calcolo.cc

#include "Polinomio.hh"

// Codice che usa i tipi Polinomio e Razionale
```

Questo approccio *sembra* funzionare nella situazione contingente, ma è destinato a creare più problemi di quanti non ne risolva.

In primo luogo, diminuisce la leggibilità del codice, perché non è più così evidente che `Calcolo.cc` dipende anche dall'header file `Razionale.hh` (affidandosi all'inclusione indiretta).

In secondo luogo, occorre considerare cosa succede se il responsabile dello sviluppo della classe `Polinomio` decidesse di modificare l'implementazione della sua classe, per esempio per utilizzare il tipo di dato `Frazione` al posto di `Razionale` (eliminando l'inclusione di questo header file): la compilazione di `Calcolo.cc` fallirebbe anche se nessuno ne ha modificato il file sorgente (cosa che non accadrebbe se venisse mantenuta in `Calcolo.cc` l'inclusione di `Razionale.hh`).

Occorre quindi consentire ad ogni unità di traduzione di includere tutti gli header file di cui necessita, trovando una soluzione alternativa al problema delle inclusioni multiple dello stesso header file.

Alcuni compilatori prevedono l'utilizzo di direttive speciali per il processore (che non sono standard), come `#pragma once` che, inserita nell'header file, comunica al preprocessore di evitarne l'inclusione multipla.

Una soluzione che invece è garantita funzionare per qualunque compilatore si basa sull'uso delle "guardie contro l'inclusione ripetuta". In pratica, l'header file `Razionale.hh` viene modificato in questo modo:

```
// Siamo in Razionale.hh

#ifndef RAZIONALE_HH_INCLUDE_GUARD
#define RAZIONALE_HH_INCLUDE_GUARD 1

class Razionale {
  // ... le cose giuste
};

#endif /* RAZIONALE_HH_INCLUDE_GUARD */
```

La prima volta che il file viene incluso, la *"guardia"* (ovvero, il flag del preprocessore `RAZIONALE_HH_INCLUDE_GUARD`) NON è ancora definita e quindi il preprocessore procede con l'inclusione. Come prima cosa, viene definita la guardia stessa e poi si includono nell'unità di traduzione tutte le dichiarazioni e definizioni (potenzialmente

elaborando altre direttive di inclusione). Se capitasse, durante il preprocessing di quella stessa unità di traduzione, di ritentare altre volte l'inclusione di `Razionale.hh`, il preprocessore troverebbe la guardia già definita; la condizione di `#ifndef` valuterebbe a falso e quindi non avverrebbe nessuna inclusione ripetuta.

> NOTA: questa soluzione, per funzionare, deve essere applicata sistematicamente su *tutti* gli header file che fanno parte del programma (`Polinomio.hh`, ecc). Inoltre, occorre prestare attenzione ed evitare di usare la stessa guardia (ovvero, flag del preprocessore con lo stesso nome) per file di inclusione distinti.

> NOTA: verificare che le guardie contro l'inclusione ripetuta sono utilizzate negli header file della libreria standard distribuiti con `g++`.

Per esempio, nell'header file `iostream`:

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

// ...

#endif /* _GLIBCXX_IOSTREAM */
```

Torna all'indice

Costrutti del linguaggio

Dopo avere discusso della ODR e delle sue possibili violazioni, può essere utile fare un elenco dei costrutti del linguaggio che è ragionevole trovare all'interno di un header file:

- direttive del preprocessore (inclusione, guardie, macro, ...)
- commenti
- dichiarazioni o definizioni di tipo
- dichiarazioni di variabili
- definizioni di costanti
- dichiarazioni di funzioni
- definizioni di funzioni inline
- dichiarazioni o definizioni di template
- namespace dotati di nome: `namespace N { ... }`
- alias di tipi: `{ typedef std::vector<int> Ints; using Ints = std::vector<int>; }`

I seguenti costrutti, invece, NON si dovrebbero trovare in un header file:

- definizione di variabili
- definizione di funzioni

Perchè?

Siccome gli header file sono pensati per essere inclusi in più unità di traduzione, si avrebbe una violazione della ODR (clausola 2).

Anche i seguenti costrutti NON si dovrebbero trovare in un header file:

- namespace anonimi: `namespace { ... }`
- dichiarazioni di `using`
- direttive di `using`

In questo caso, però, non c'entra la *ODR*. Direttive e dichiarazioni di `using` annullano il meccanismo di protezione dai conflitti di nomi che viene fornito dai `namespace`.

È lecito farlo in contesti molto limitati e sotto il nostro controllo (ad esempio, all'interno della definizione di una funzione o all'interno di una unica unità di traduzione). È considerato cattivo stile invece farlo in un `header file` (a meno che non si sia all'interno del corpo di una funzione inline o di un template di funzione), perché il potenziale problema si propagherebbe a tutte le unità di traduzione che includono il nostro `header file` (direttamente o indirettamente).

Torna all'indice