Lezione4

Table of contents

Riferimenti vs Puntatori

1. Quando viene creato un riferimento, dobbiamo *per forza inizializzarlo*; per il puntatore possiamo anche non farlo (wild/dangling/zero).

2. Creato il riferimento, per tutta la sua vità si *riferirà a un specifico oggetto*, quindi non posso creare un riferimento e cambiare l'oggetto a cui si riferisce; i puntatori possono puntare oggetti diversi.

3. Con operazioni su riferimento, lavoriamo sempre sull'oggetto riferito; il puntatore è un oggetto diverso e abbiamo 2 tipologie di operazioni possibili: *operazioni sul puntatore*, *operazioni sull'oggetto puntato*.

```
*p // operator* prefisso
p -> a // operator -> infisso, equivalente a (*p).a
```

4. Quando usiamo `const` a chi ci stiamo riferendo?
Tutto quello che sta a sinistra si riferisce all'oggetto puntato, tutto quello che sta sulla destra si riferisce al puntatore.

```
int i = 5;
const int ci = 5; // CORRECT: non modificabile
int& r_i = i; // CORRECT: posso modificare 'i' con 'r_i'
const int& cr_i = 1; // CORRECT: posso modificare 'i' usando 'cr_i'

int& r_ci = ci; //ERROR: riferimento non 'const' ad oggetto 'const'

const int& cr_ri = ci; // CORRECT: accesso in sola lettura
```

```
int* p_i; // CORRECT: entrambi 'p_i' e '*p_i' sono modificabili
const int* p_ci; // CORRECT: puntatore costante
int* const cp_i = &i; // CORRECT: 'cp_i' non modificabile, '*cp_ci' modificabile
const int* const cp_ci = &i; // niente è modificabile
```

Riferimenti & Puntatori

1. Al termine del tempo di vita di un puntatore, cosa accade?
Il puntatore quando muore *non fa succedere nulla all'oggetto puntato*, cosa a cui dobbiamo porre attenzione per memory leak.
Al termine del tempo di vita di un riferimento, anche lui *non tocca l'oggetto a cui si riferisce*.

2. Non esistono i riferimenti nulli, ma possono esistere i *riferimenti dangling*: il riferimento sopravvive all'oggetto riferito che invece scompare.

```
// restituiamo il riferimento di un oggetto che oramai è stato distrutto
struct S { /* ... */ };
S& foo {
   S s;
   // ...
   return s;
}
```

   L'esempio sopra è un classico esempio di copia temporanea non necessaria: viene creato un oggetto di tipo `S` che alla fine non serve niente se non a essere distrutto. Da questa idea nascono nel C++, per ovviare al problema di copie inutili, *riferimenti a r-value* e *costruttori di spostamento* sono nati.

One Definition Rule (ODR)

"Posso avere tante dichiarazioni ma una sola definizione" sarebbe la frase che spiega la ODR, ma è imprecisa: le unità di traduzione possono comunicare tra di loro, e a queste servono regole per precisare come sono fatti i dati e le operazioni.
Negli header file per esempio vengono definiti i tipi di dato e le dichiarazioni delle funzioni: tante unità di traduzione possono avere modi diversi di calcolare un fattoriale.

Le unità di traduzione devono concordare con un'interfaccia (contenente le dichiarazioni di dati o funzioni); per evitare che un'unità si trovi a lavorare per sbaglio con un tipo/funzione errato, seguiamo una regola precisa: Don't Repeat Yourself (DRY), le dichiarazioni vengono scritte una volta sola, in un *header file*.

La One Definition Rule detta che:

1. ogni unità di traduzione (il risultato della fase di preprocessing su un file sorgente(`HelloWorld.preproc,cpp`)) che forma il programma, può contenere *non più di una definizione di una data variabile, funzione, classe, enumerazione o template*;
2. ogni programma, deve contenere *esattamente 1 definizione di ogni variabile e di ogni funzione non-inline* usate nel programma;
3. TODO
4. in un programma possono esserci *più definizioni di classe, enumerazione, funzione inline, template di classe e di funzione*, a condizione che:
   1. devono essere *sintatticamente identiche*;
   2. devono essere *semanticamente identiche*.

Esempi di violazione ODR

Punto 1

```
// definizione multipla
// ERRORE: ambiguità
struct S { int a; };
struct S { char c; double d };
int a;
int a;
```

Punto 2

```
// ERROE: foo() dichiarata ma non definita

// 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); }
```

01/03/2023