Riferimenti vs puntatori

Enea Zaffanella

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

Tipi riferimento e tipi puntatore

- tipi riferimento e tipi puntatore vengono spesso confusi
  - dai programmatori che si concentano sugli aspetti implementativi
  - rappresentati internamente utilizzando l'indirizzo dell'oggetto riferito/puntato
- visione a più alto livello mette in evidenza le differenze
  - a livello **sintattico**
  - a livello **semantico**

A livello intuitivo

- una intuizione utile, anche se formalmente non esatta:
  - un puntatore è un **oggetto** il cui valore (un indirizzo) si può riferire ad un altro oggetto
  - un riferimento **non** è un oggetto vero e proprio, ma è una sorta di "nome alternativo" che consente di accedere ad un oggetto esistente
- da queste due "definizioni" seguono alcune osservazioni ...

1: inizializzazione

- quando viene creato un riferimento, questo **deve** essere inizializzato, perché si deve sempre riferire ad un oggetto esistente

```
int i;        // ok
int& r1 = i; // ok
int& r2;      // errore
```

- in altre parole, non esiste il concetto di "riferimento nullo"
- invece un puntatore può non essere inizializzato (rischioso), oppure essere inizializzato con il letterale `nullptr` (o con il puntatore nullo del tipo corretto, mediante conversioni implicite) nel qual caso **NON** punterà ad alcun oggetto

2: modificabilità

- una volta creato un riferimento, questo si riferirà sempre allo stesso oggetto; non c'è modo di "riassegnare" un riferimento ad un oggetto differente
- invece, durante il suo tempo di vita, un oggetto puntatore (che non sia qualificato `const`) può essere modificato per puntare ad oggetti diversi o a nessun oggetto

```
int i, j;
int* p = &i; // ok, p punta ad i
p = &j;      // ok, ora p punta a j
```

3: accesso a oggetto riferito/puntato

- ogni volta che si effettua una operazione su un riferimento, in realtà si sta (implicitamente) operando sull'oggetto riferito
- nel caso dei puntatori, invece, abbiamo a che fare con **due oggetti diversi**: l'oggetto puntatore e l'oggetto puntato
  - operazioni di lettura e scrittura (comprese le operazioni relative all'aritmetica dei puntatori) applicate direttamente al puntatore accederanno e potenzialmente modificheranno l'oggetto puntatore
  - per lavorare sull'oggetto puntato, invece, occorrerà usare l'operatore di dereferenziazione

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

4: qualificazione const

- un qualificatore `const` aggiunto al riferimento si applica sempre all'oggetto riferito e non al riferimento stesso
- nel caso del puntatore, invece, avendo due oggetti (puntatore e puntato), è possibile specificare il qualificatore `const` (o meno) per ognuno dei due oggetti
- nella dichiarazione di un puntatore è possibile scrivere **due volte** il qualificatore const:
  - se `const` sta a sinistra di `*`, si applica all'oggetto puntato
  - se `const` sta a destra di `*`, si applica all'oggetto puntatore
- la cosa si può complicare nel caso di **annidamento di puntatori** (esempio, `int***`)

Esempio (parte 1)

```
int i = 5;          // oggetto modificabile
const int ci = 5;   // oggetto non modificabile

int& r_i = i;          // ok: posso modificare i usando r_i
const int& cr_i = i; // ok: non modificherò i usando cr_i
int& r_ci = ci; // errore: non posso usare un non-const &
                // per accedere ad un oggetto const
const int& cr_ci = ci; // ok: accesso in sola lettura
```

Esempio (parte 2)

```
int* p_i;         // p_i e *p_i entrambi modificabili
const int* p_ci; // p_ci modificabile, *p_ci non modificabile
int* const cp_i = &i; // cp_i non modificabile, *cp_i modificabile
const int* const cp_ci = &i; // cp_ci e *cp_ci non modificabili

int& const cr = i; // errore: const va (solo) a sinistra di &

p_i = &i;  // ok
p_i = &ci; // errore: non posso inizializzare un non-const *
           // usando un indirizzo di un oggetto const

p_ci = &i;  // ok: non modificherò i usando *p_ci
p_ci = &ci; // ok

cp_i = nullptr; // errore: cp_i non modificabile
cp_ci = &ci;    // errore: cp_ci non modificabile
```

Alcune somiglianze: tempo di vita

- al termine del tempo di vita di un puntatore (ad esempio, quando si esce dal blocco di codice nel quale era stato definito come variabile locale) viene distrutto l'oggetto puntatore, ma **NON** viene distrutto l'oggetto puntato
- analogamente, quando un riferimento va fuori scope, l'oggetto riferito non viene distrutto
- esiste però il caso speciale del riferimento inizializzato con un temporaneo: in questo caso, l'oggetto temporaneo finisce il ciclo di vita insieme al riferimento

Alcune somiglianze: riferimenti “dangling”

- come per il dangling pointer (puntatore non nullo che contiene l’indirizzo di un oggetto non più esistente), è possibile creare un dangling reference:

```
struct S { /* ... */ };

S& foo() {
  S s;
  // ...
  return s;
}
```

- grave errore di programmazione: l’oggetto riferito, allocato automaticamente, è distrutto quando si esce dal blocco
- soluzione: modificare l’interfaccia della funzione affinché ritorni per valore

Riferimenti a rvalue

- nota bene: abbiamo considerato esclusivamente i riferimenti a lvalue (`T&`)
- molte delle osservazioni fatte sono valide anche per il caso di riferimenti a rvalue (`T&&`), il cui approfondimento è rimandato ad un altro momento