5/09/25

PUNTATORI, VARIABILI E REFERENZE

Immaginiamo di partire da una variabile ordinaria, ad esempio una stringa: std::string String = "iAmAString";

Questa dichiarazione crea in memoria un oggetto std::string. La variabile *String* non è altro che un **nome simbolico** che il compilatore associa a un indirizzo di memoria. In altre parole, quando scrivo *String*, io sto accedendo direttamente al valore contenuto in quello spazio di memoria. Non c'è intermediazione: il valore e la variabile coincidono concettualmente.

Un puntatore è una variabile "speciale", il cui contenuto non è un valore logico immediato, bensì un **indirizzo di memoria**. Se io scrivo: std::string *stringPTR = &String;

allora *stringPTR* contiene l'indirizzo dove risiede la stringa *String*. Se io dereferenzio con **stringPTR*, vado a recuperare o modificare il contenuto effettivo della variabile a quell'indirizzo.

Il puntatore ha due implicazioni fondamentali:

1. Introduce un livello di indirezione. Per accedere al valore devo "saltare" a quell'indirizzo.
2. Può essere riassegnato, spostandosi a indicare variabili diverse. Inoltre può anche essere nullo, cioè non riferirsi a nessun oggetto valido.

Questo ci permettono di gestire strutture dati dinamiche, passare grandi oggetti senza copiarli, lavorare con la memoria dinamica (new/delete o malloc/free in C), ma allo stesso tempo introduce complessità e rischi (dangling pointers, memory leaks).

C++ aggiunge un ulteriore costrutto: la **referenza**. Una referenza è un meccanismo che serve a creare un **alias** di un oggetto già esistente. Quando io scrivo: std::string &stringREF = nome;

non sto creando un nuovo oggetto, né una nuova variabile autonoma: stringREF diventa un altro nome per *String*. Qualunque operazione compiuta su *stringREF* ha effetto immediato su String, e viceversa.

Ci sono due caratteristiche essenziali delle referenze:

- Devono essere inizializzate al momento della dichiarazione.
- Non possono essere "rindirizzate" verso un altro oggetto: rimangono alias fissi dell'oggetto iniziale.

Questa caratteristica le differenzia radicalmente dai puntatori. La referenza non introduce il rischio di un puntatore nullo o non inizializzato, ed è molto più sicura da usare in molti contesti. È per questo che in C++ le referenze vengono impiegate estensivamente per passare argomenti alle funzioni senza costi di copia, ma senza perdere in chiarezza e sicurezza.

Esempio:

```
void stampa(const std::string &s) {
   std::cout << s << std::endl;
}
```

Qui la funzione non copia la stringa, ma lavora direttamente sull'oggetto chiamante. Se avessi usato un puntatore, avrei dovuto scrivere:

```
void stampa(const std::string *s) {
   std::cout << *s << std::endl;
}
```

che obbliga chi usa la funzione a passare l'indirizzo e dereferenziare, aumentando il rischio di errori.

- La variabile normale rappresenta **direttamente un oggetto in memoria**.
- Il puntatore rappresenta un **indirizzo**, quindi un livello di indirezione, con grande flessibilità ma anche potenziali pericoli.
- La referenza è un **alias sicuro e trasparente**, introdotto in C++ proprio per offrire la praticità del puntatore senza la sua pericolosità.

EX02 Module 01

INIZIO PROGRAMMA

1. Dichiarare una variabile stringa con valore "HI THIS IS BRAIN".
2. Dichiarare un puntatore che contiene l'indirizzo della variabile stringa.
3. Dichiarare una referenza che si riferisce direttamente alla variabile stringa.
4. Stampare:
   a) Indirizzo della variabile stringa.
   b) Indirizzo contenuto nel puntatore.
   c) Indirizzo della referenza.
5. Stampare:
   a) Valore della variabile stringa.
   b) Valore ottenuto tramite il puntatore (dereferenziazione).
   c) Valore ottenuto tramite la referenza.

FINE PROGRAMMA

Scopo didattico dell'esercizio

- Dimostrare che **variabile, puntatore e referenza** possono riferirsi allo stesso oggetto in memoria.
- Evidenziare la differenza tra:
  - **indirizzo** (con & o puntatore),
  - **valore** (con accesso diretto, tramite * o tramite referenza).
- Capire che:
  - Gli indirizzi stampati coincidono.
  - I valori stampati coincidono.

EX 03 MODULE 01

PUNTATORI E REFERENZE

La reference può essere immaginata come un "altro nome" per un oggetto già esistente. Nel momento in cui la si dichiara, è necessario legarla immediatamente a un oggetto, e da quel momento in poi non è più possibile riassegnarla. La reference garantisce sempre l'esistenza di un oggetto valido dietro di sé, non può essere vuota, non può essere spostata altrove. Una reference viene usata quando si vuole modellare una relazione stabile e garantita. Dal punto di vista del programmatore, significa che un oggetto "dipende" da un altro e non può esistere senza di esso. In un costruttore, per esempio, dichiarare un parametro come reference obbliga chi istanzia l'oggetto a fornire subito un argomento valido. È una forma di contratto semantico: "questo oggetto non può nascere senza quell'altro". Non si deve preoccuparsi di verificare la validità della reference, né di controllare se essa punti a qualcosa, perché è sempre garantito che lo faccia. L'uso tipico, quindi, è in situazioni in cui l'associazione non può mancare: un motore dentro un'automobile, un'arma per un guerriero che non può restare disarmato, oppure nei parametri di funzione quando vogliamo passare un oggetto senza copiarlo, ma allo stesso tempo senza il rischio di puntatori nulli.

Ex:

Weapon club = Weapon("crude spiked club");

Weapon &ref = club; // ref è una reference a club

Da questo momento in poi, usare `ref` equivale ad usare direttamente `club`. Non servono operatori speciali per accedere ai membri: se `Weapon` ha un metodo `getType()`, basta scrivere `ref.getType()`.

Il puntatore, invece, rappresenta un livello di astrazione diverso. Un puntatore non è un alias, ma una variabile che contiene un indirizzo di memoria. Proprio per questo motivo può essere inizializzato a `NULL` (o `nullptr` in C++11 e versioni successive), cioè può non riferirsi ad alcun oggetto valido. Inoltre, può cambiare nel corso del tempo: un puntatore oggi può indicare un'arma, domani un'altra, oppure restare temporaneamente vuoto. Il puntatore viene utilizzato quando l'associazione non è obbligatoria, oppure quando può cambiare nel corso della vita dell'oggetto. Permette di esprimere l'idea di optionalità: un guerriero può nascere senza arma e ottenerla solo più tardi, oppure può cambiare arma più volte. Nei programmi più complessi, i puntatori sono fondamentali quando si lavora con strutture dinamiche, come liste collegate, alberi, grafi, o quando occorre gestire risorse che possono esistere o meno. In questi casi, la possibilità di assegnare un puntatore a `nullptr` diventa un modo esplicito per rappresentare l'assenza di un valore. Tuttavia, questo utilizzo porta con sé l'obbligo di controllare sempre che il puntatore sia valido prima di dereferenziarlo, pena errori di esecuzione anche gravi.

Weapon club = Weapon("crude spiked club");

*Weapon *ptr = &club; // ptr punta a club*

si dichiara con l'asterisco `*`. A differenza della reference, un puntatore può essere inizializzato anche a `nullptr`, e non è obbligatorio legarlo subito a un oggetto. Per accedere ai membri dell'oggetto puntato si utilizza l'operatore `->`, oppure bisogna dereferenziare con `*` e poi accedere con il punto.
ptr->getType(); // equivalente a (*ptr).getType()

METODI CONST E DI RITORNO PER RIFERIMENTO COSTANTE

Se un metodo restituisce un oggetto complesso, come una `std::string`, restituirlo per valore implica creare una copia temporanea. Questa copia, per quanto ottimizzata dal compilatore, ha comunque un costo in termini di memoria e prestazioni, soprattutto se il metodo viene invocato di frequente. Restituire invece un riferimento (`&`) significa non creare copie, ma rimandare direttamente all'oggetto originale. Tuttavia, se questo riferimento non fosse dichiarato `const`, il chiamante potrebbe modificare dall'esterno l'oggetto interno della classe, *violando l'incapsulamento.* Per questo si restituisce un `const &`: il chiamante può leggerne il contenuto, ma non può alterarlo. Nel caso di `Weapon::getType()`, il ritorno `const std::string&` permette di accedere al tipo dell'arma senza creare copie e senza rischiare che il chiamante cambi la stringa dall'esterno.

L'altro aspetto riguarda il **qualificatore `const` dopo la dichiarazione della funzione**. In C++, scrivere `... getType() const` significa che quel metodo promette di non modificare lo stato dell'oggetto su cui è invocato. Questa è una garanzia contrattuale: se un metodo è `const`, il compilatore non permette l'uso di operazioni che alterino i membri della classe. È un modo per distinguere nettamente le funzioni di sola lettura da quelle che hanno effetti sullo stato interno. Tornando all'esempio, `Weapon::getType()` è dichiarato `const` perché il suo compito è puramente osservativo: leggere e restituire il tipo di arma, non modificarlo.

ASSOCIAZIONE E POSSESSO

Un'associazione descrive un legame d'uso tra due entità indipendenti: una classe utilizza un'altra senza controllarne il ciclo di vita. Questo implica che l'oggetto referenziato o puntato deve essere gestito dall'esterno e deve vivere almeno quanto chi lo utilizza, altrimenti si rischia di lavorare con riferimenti o puntatori non validi.

In termini di modellazione, dunque, è fondamentale distinguere tra:

- **composizione**, dove l'oggetto contenitore possiede e governa l'esistenza delle sue parti;
- **aggregazione**, dove un oggetto contiene altri ma senza gestirne la vita in modo stretto;
- **associazione semplice**, che esprime una collaborazione senza vincoli di possesso.

Il caso di `HumanA` e `HumanB` con `Weapon` mostra proprio questo: l'arma non appartiene all'umano, ma gli è semplicemente associata. Questo ribadisce il principio generale che le relazioni in OOP possono assumere forme diverse a seconda della responsabilità sul ciclo di vita degli oggetti, e che scegliere il tipo di relazione corretto significa modellare il problema in maniera fedele e sicura.

Significa che in programmazione a oggetti non tutte le volte in cui una classe "usa" un'altra vuol dire che la possiede o la controlla. Nel caso di `HumanA` e `HumanB`, l'oggetto `Weapon` non viene creato o distrutto all'interno della classe "umano", ma esiste autonomamente e viene solo collegato tramite reference o puntatore. Questo esempio serve a dimostrare un principio più ampio: in OOP esistono diversi tipi di relazioni tra classi, e la differenza sta nella **responsabilità sul ciclo di vita degli oggetti**. Se una classe gestisce la vita di un'altra (la crea e la distrugge), si parla di **composizione**; se la include ma non ne governa l'esistenza, si parla di **aggregazione**; se invece la utilizza senza possederla, siamo di fronte a una **associazione semplice**.

Quello che emerge con `HumanA` e `HumanB` è una forma di **polimorfismo statico**, cioè un comportamento polimorfico ottenuto senza ricorrere a ereditarietà o a meccanismi di late binding come il polimorfismo dinamico.

Entrambe le classi dichiarano un metodo con lo stesso nome, `attack()`, ma la loro implementazione riflette la natura del legame con l'arma: `HumanA` sa di avere sempre un'arma, perché la conserva come reference, e quindi il suo attacco non deve gestire casi particolari; `HumanB`, invece, lavora con un puntatore che può anche non essere inizializzato, e la sua logica deve verificare se l'arma esiste prima di procedere.

Il concetto chiave, quindi, è che il polimorfismo non coincide unicamente con l'uso di ereditarietà e metodi virtuali, ma può manifestarsi anche in forma statica: classi distinte che espongono lo stesso contratto comportamentale, interpretandolo ciascuna secondo le proprie caratteristiche interne. In pratica, sia `HumanA` che `HumanB` hanno un metodo `attack()`. Da fuori io chiamo sempre `attack()`, ma il risultato cambia a seconda della classe: uno attacca sapendo di avere sempre un'arma, l'altro deve prima controllare se l'arma c'è. Quindi il concetto chiave è: **stesso messaggio (attack), comportamenti diversi**, anche senza ereditarietà. È questo che rende il codice più flessibile e vicino al modo in cui ragioniamo nel mondo reale.