Programmazione 1

Classi e Oggetti: Le classi sono dei **tipi non primitivi** definiti dall'utente (**user-defined**), essere servono ad **incapsulare** dati, quindi a definire gli **stati e i comportamenti** degli oggetti che verranno creati a partire dalla struttura della classe, che non è altro che uno **stereotipo** dell'oggetto. Un oggetto è un'**istanza** della classe, esso inizializza gli stati presenti nella classe e ne utilizza i metodi. Per ogni classe possono esistere infiniti oggetti e per ognuno di essi vale il **principio di identità**: ogni oggetto ha una propria identità a prescindere dal suo stato, di conseguenza esso sarà diverso da tutti gli altri. Ogni oggetto ha un proprio **ciclo di vita** determinato da tre fasi: **creazione**, **utilizzo** e **distruzione.** Anche per gli oggetti valgono i diversi tipi di **allocazione**, dinamica, nello heap (puntatore a oggetto) e statica nello stack. Se alloco un oggetto con scope globale esso sarà allocato nel segmento data.

Metodi e attributi: Un **metodo** viene creato a partire da una classe ed è una funzione che potrà essere invocata dagli oggetti della classe stessa, esso può intervenire sugli attributi della classe e compiere azioni specifiche. Un **attributo** è una variabile che definisce lo stato di una classe, esso può essere istanziato per ogni oggetto creato ed essere modificato da opportuni metodi. Per **accedere ai metodi** di una classe a partire da un oggetto esistono due metodi: se l'oggetto è **statico** utilizzerò la notazione **obj.metodo(),** se invece voglio usare un **puntatore a oggetto** userò la notazione **pobj->metodo()** o **dereferenziandolo**, userò ***pobj.metodo();**

Metodo costruttore: Il metodo costruttore è un particolare metodo che non prevede tipo di ritorno e assume lo stesso nome della classe, esso serve ad **inizializzare** gli oggetti che vengono creati. La chiamata al costruttore è automatica e avviene **simultaneamente alla creazione dell'oggetto**, essa non può essere evitata, in questo modo la creazione e l'inizializzazione di un oggetto sono inseparabili perché il compilatore forzerà sempre la chiamata. Quando non viene definito un costruttore per la classe, il compilatore ne predispone uno di **default** che non compie particolari azioni se non la creazione dell'oggetto. Una classe può avere anche **più di un costruttore**, a seconda dei parametri passati e del tipo di oggetto che si vuole creare.

Costruttore immediato: esso viene creato utilizzando la **lista di inizializzazione** dei parametri, ad esempio: X(int a, int b)**:x(a),y(b);** In questo modo a e b saranno già legate ad x e y attributi della classe. Quando dichiaro un oggetto come **part of** di un'altra classe, ad esempio il **costruttore di una classe ereditata**, devo obbligatoriamente passare i parametri del costruttore padre tramite lista di inizializzazione. Questo vincolo prende il nome di **relazione di composizione**: la vita di un oggetto non è indipendente ma legata a quella di un oggetto di un'altra classe. (le variabili const e le reference devono per forza essere inizializzate e non possono essere usate tramite lista dei parametri).

Costruttore di copia: quando non creato, quello di default esegue una copia **membro a membro** dell'oggetto da creare **(X a = b;).** Quando utilizzo i **puntatori** non posso però utilizzare il costruttore di default perché creerei due puntatori alla stessa area di memoria creando errori di **segmentation fault**, va quindi creato un costruttore di copia che non riceve parametri ma **reference** all'oggetto da copiare in modo da **non eseguire una copia memberwise** ma una definita dal programmatore. Il costruttore di copia verrà chiamato quando assegnerà alla creazione di un oggetto un'istanza già esistente

Distruttore: Quando alloco dati nell'**heap**, essi saranno indipendenti dal ciclo di vita dell'oggetto in quanto in un'area diversa da quella dello stack riservato all'oggetto, essi infatti non saranno mai distrutti se non manualmente dal programmatore. Per questo devono essere predisposti dei metodi **distruttori**, identificati dalla tilde seguita dal nome della classe **(~Class).** Essi non fanno altro che chiamare la **delete** per tutti i dati allocati nell'heap. Se prima di cancellare i dati nell'heap, cancellassi il puntatore che li punta (i dati dell'heap possono essere accessibili solo tramite puntatore creato da comando **new**), incorrerei in **memory leak**, ovvero i dati non saranno più accessibili in nessun modo e quindi non più cancellabili e sprecherebbero solo memoria.

Modificatori di accesso: In una classe, metodi e attributi possono avere diversi **scope**, a seconda del livello di sicurezza dell'incapsulamento che si vuole ottenere. Quindi ogni componente della classe può essere definito sotto tre modificatori di accesso:

-**private**: gli attributi e le variabili definite sotto questo modificatore, potranno essere usate direttamente solamente dai membri della stessa classe e per essere usati dall'esterno dovranno essere predisposti opportuni metodi detti **getter** e **setter**.

-**public**: gli attributi e i metodi sotto questo modificatore possono essere usati indistintamente dall'interno e dall'esterno, public quindi non limita lo scope. (il costruttore deve sempre essere dichiarato public).

-**protected**: gli attributi e i metodi sotto questo modificatore godono di proprietà particolari, infatti, essi risulteranno public per le classi ereditate dalla classe che li contiene, risulteranno invece private per le classi esterne che non hanno nessuna relazione con la classe in cui sono contenute.

-**unqualified**: per sintassi, quando non viene definito nessun modificatore di accesso per un componente della classe, i metodi e gli attributi dichiarati verranno assunti dal compilatore come private.

Oltre ai modificatori standard, ogni metodo può assumere alti modificatori personali quali static e const:

const:

metodo: è una promessa del programmatore al compilatore, infatti quel metodo non potrà modificare lo stato della classe (di attributi e metodi) ma si limiterà a ritornare informazioni, ne sono un esempio i metodi getter, mentre i setter non potranno mai essere const.

attributo: gli attributi const rendono il valore di inizializzazione non modificabile a run-time.

static:

metodo: i metodi static non sono associati ad una singola istanza della classe ma a tutte quelle create, essi possono accedere solamente ad altri attributi e metodi static e possono essere chiamati direttamente dalla classe, ad evidenziare che non sono vincolati ad un singolo oggetto.

attributo: un attributo static manterrà lo stesso valore per ogni istanza della classe, quindi ad esempio trova impiego come contatore di istanze create, dato che non è vincolato all'oggetto ma alla classe. Le variabili static possono essere create all'interno della classe ma il loro valore va inizializzato obbligatoriamente al di fuori di essa, chiamandole per scope. (tutti i dati static vengono allocati nel segmento **data**).

Metodi friend: una funzione friend **viola il principio dell'incapsulamento**, infatti essa è una funzione che viene definita al di fuori dello scope della classe ma ha comunque accesso a tutti i metodi e gli attributi di essa, public o private che siano. Un metodo friend può essere usato da più di una classe e può anche essere membro di una terza classe. Per definire un metodo friend bisogna operare nel seguente modo: definire a scope globale il metodo e il suo body, e in seguito dichiarare il prototipo all'interno della classe amica, definendo lo **specificatore** friend.

Overloading metodi: L'overloading è una forma di programmazione generica in cui una funzione con lo stesso nome viene utilizzata con tipi di input (signature) diverso nello stesso scope. Posso creare famiglie di overloading per utilizzare lo stesso metodo indifferentemente dai parametri passati, una volta creati gli opportuni metodi con stesso nome e signature diversa, sarà il compilatore ad occuparsene (scope uguale).

Overriding: ridefinizione di un metodo precedentemente definito in una classe base in una classe derivata. Si ha overriding se i due metodi hanno la stessa **intestazione** (scope diverso).

Overloading operatori: a livello implementativo, gli operatori sono **funzioni** a tutti gli effetti, il cui nome assume la forma ***operator@***, dove con @ intendiamo uno qualsiasi tra gli operatori. Il problema principale consiste nella definizione dell'**ambito di visibilità** degli operatori sovraccaricati. Alcuni operatori infatti possono essere sovraccaricati solo come membri di classe (assegnamento), perchè per la loro implementazione è richiesto l'uso del membro **this**, altri solo come funzioni esterne alla classe (ad esempio gli operatori << e >> di inserimento ed estrazione da flusso), altri ancora indifferentemente nell'uno o nell'altro modo. Quando si effettua l'overloading di un operatore, non è possibile ridefinire il numero di parametri (+ ad esempio può essere solo binario). Quando dichiaro un overloading come funzione membro, il parametro di sinistra può essere omesso, dato che il compilatore gli assegnerà il puntatore all'oggetto chiamante (this). Bisogna prevedere anche l'opportuno tipo di ritorno per una funzione overloaddata, inoltre quando voglio che il valore di ritorno possa essere usato come **lvalue**, quindi sia assegnabile, la funzione overloading deve tornare una **reference** all'oggetto e non l'oggetto stesso.

Ereditarietà: Una classe (superclasse) può essere ereditata da un'altra (sottoclasse), in modo tale che la sottoclasse acquisisca tutte le caratteristiche della superclasse. Quando eredito, non faccio altro che copiare il modello della classe padre usandolo come scheletro per la classe figlio.

Gli attributi e i metodi **private** possono essere ereditati ma solo la classe madre potrà manipolarli. Quando eredito una classe, e nella sottoclasse definisco un metodo con lo **stesso nome** di un metodo della superclasse ma signature diversa, questo metodo oscurerà quello istanziato nella superclasse che diventerà accessibile solo tramite **operatore risolutore di scope** o direttiva **using** (using A::f()).

Un membro **protected** della classe padre, risulterà public per la classe figlio e private per l'esterno.

Modificatori di accesso delle classi: quando eredito, uso la seguente sintassi (class B: modificatore A), dove B è la classe ereditata e A la classe padre, a seconda del modificatore succedono le seguenti cose:

-**unqualified**: va considerato come **public**, l'accesso ai membri ereditati rimane invariato nella classe ereditata quindi l'ereditarietà non restringe l'area di azione.

-**protected**: i metodi e le classi public della superclasse vengono ristretti in private nella classe figlio.

-**private**: tutti i metodi e gli attributi ereditati diventano private, restringendo l'interfaccia al massimo.

Polimorfismo: il polimorfismo è l'abilità di creare un metodo o un attributo che ha **più di una forma.** Possiamo parlare di polimorfismo ad esempio quando abbiamo una gerarchia di classi ereditate da un'interfaccia comune, in cui ogni classe possiede un metodo con lo stesso nome ma signature diversa (overriding), quel metodo sarò polimorfo, ovvero avrà forma diversa e nome uguale. Esistono due tipi di polimorfismo:

-**Polimorfismo dinamico:** viene effettuato a **run-time** (overriding + virtual). Esso viene usato in presenza di **puntatori o reference**, in quanto al compilatore non sarà noto il tipo di oggetto referenziato, quindi il compilatore conserverà le informazioni nella **tabella delle funzioni virtuali**, sarà così in grado di identificare il comportamento della specifica funzione a run-time. Questo comportamento è detto **late-binding**. La funzione **virtual** va dichiarata solo nella classe padre e serve al compilatore per creare le tabelle delle funzioni virtuali, in assenza della **keyword**, a prescindere dal tipo di puntatore, il compilatore chiamerà sempre il metodo padre.

-**Polimorfismo statico**: viene effettuato a **compile-time** (overloading). Qui il compilatore applica una serie di **regole di risoluzione dell'overloading** per determinare a compile-time quale metodo evocare.

Esempio: data una classe A e una B ereditata da A e un metodo f definito in entrambe.

```
A* ptr = new B();

ptr->f();
```

-se in A f() è dichiarato virtual, ptr chiamerà f() di B perché data la tabella delle funzioni virtual, lo scope di sarà B anche se ptr è di tipo A.

-se in A f() è invece non virtual, non verrà creata la tabella delle funzioni virtual, quindi il compilatore chiamerà f() di A.

Problemi di costruzione e distruzione polimorfa: un costruttore non può essere dichiarato virtual.

Costruttore: la costruzione di un oggetto di una classe derivata implica:

1. invocazione del costruttore della classe base;
2. invocazione del costruttore della classe derivata.

Distruttore: la distruzione di un oggetto sullo **stack** (tipo noto) implica:

1. Invocazione distruttore classe derivata;
2. Invocazione distruttore classe base.

Deallocatore: la deallocazione di un oggetto sull'**heap** (tipo non noto) implica:

1. Invocazione distruttore classe derivata, solo se nella classe base il distruttore è virtual;
2. Invocazione distruttore classe base.

Principio di sostituzione di Liskov: Un tipo S è un sottotipo di T quando è possibile sostituire tutte le istanze di T con delle istanze di S mantenendo intatto il funzionamento del programma.

Funzione virtuale pura: una funziona virtuale pura (void f() = 0), si utilizza quando è necessario fornire un'implementazione ma il concetto è troppo **astratto** per essere generalizzato.

Classe astratta: una classe con almeno una funzione virtuale pura si dice classe astratta e:

1. Non si possono creare istanze di essa;
2. È possibile creare puntatori e reference di questo tipo;
3. Se una classe derivata da una classe astratta non fornisce l'implementazione di almeno un metodo virtual puro, anch'essa sarà astratta.

Interfaccia: Una classe astratta che contiene solo: **attributi**, **costanti** e **funzioni virtuali pure** si dice interfaccia. In un'interfaccia nessun comportamento è definito ma solamente **dichiarato**. (specifiche senza implementazione).

Down-casting: Il **downcasting** è usato per convertire il tipo di **puntatore**. Ad esempio quando un puntatore è stato allocato a run-time e voglio conoscerne il tipo, utilizzo l'operatore **typeid** e poi eseguo un **dynamic cast** per cambiarne la natura ed effettuare operazioni **safe**.

Classi e funzioni template: I template vengono usati per rendere il **codice riutilizzabile** in quanto operano su tipi generici, specializzandosi in tipi specifici (primitivi o user-defined) solo a run-time, in occorrenza della loro chiamata. Il **binding** da tipo generico a tipo specifico è detto **argument- deduction**, ogni tipo generico deve corrispondere a run-time ad un tipo specifico e soprattutto deve essere precedentemente dichiarato **nella lista dei parametri** del template.

Esempio di classe template:

```
template< class T> class A{ };
```

dove T è il parametro generico assunto dal template a run-time. Applicando la funzione **typeid** a classi di questo tipo, l'output sarà: **nome_classe<tipo>.**

Namespace: I namespace sono **contenitori** di classi e oggetti che possono essere importati per rendere la **programmazione modulare** e favorire il riutilizzo del codice. Per utilizzare gli oggetti presenti in un namespace posso:

1. **Includere tutto il namespace**, in modo da evitare di richiamare lo scope ogni volta. Per includere il namespace uso la direttiva "using namespace nome_template".
2. **Usare l'accesso ad un singolo oggetto** del namespace, con "using namespace::oggetto".
3. **Usare ogni volta l'operatore risolutore di scope** concatenato con "namespace::classe::oggetto".

Ogni volta che importo più di un namespace o uso classi con nomi uguali a quelli presenti nel namespace importato, se non qualifico con attenzione cosa sto usando, posso creare errori di **name-clash.**

Reference: Le reference sono state introdotte per utilizzare i **vantaggi dei puntatori** ma evitarne la sintassi scomoda, avendo l'impressione di star lavorando con tipi primitivi. Quando uso le reference, creo un **alias** della variabile che ha la funzionalità di un puntatore ma non crea l'inconveniente della dereferenziazione. Una reference non può essere di tipo **NULL**, va quindi subito inizializzata con l'oggetto che deve copiare. Quando un metodo lavora su una reference, non crea una copia dei parametri sul suo stack ma lavora sull'oggetto stesso, evitando problemi di **ambiguità**. Ovviamente per usare oggetti dinamici o nulli uso comunque i puntatori.