Riassunti LP1

K

Troppo sotto l'esame

1 Mem & Env

Questi esercizi consisteranno nella traduzione delle operazioni di assegnazione che in c/c++ denotano operazioni di env (che ottiene informazioni rispetto ad una variabile d'ambiente) e di mem (che ne ricava l'indirizzo in memoria).
In generale gli esercizi si baseranno sulla traduzione, attraverso delle semplici regole di traduzione, delle istruzioni da c/c++ a mem&env e viceversa.

Descrizione	c/c++	mem&env
lhs operand	x=?	env(x)
rhs operand	?=y	mem(env(y))
lhs pointer	*x=?	mem(env(x))
rhs pointer	?=*y	mem(mem(env(y)))
lhs pointer^2	**x=?	mem(mem(env(x)))
rhs pointer^2	?=**y	mem(mem(mem(env(y))))
lhs subscript	x[z]=?	env(x) + z
rhs subscript	?=y[z]	mem(env(y)) + z
lhs address-of	&x=?	not allowed
rhs address-of	?=&y	env(y)

1.1 Qualche esercizio con soluzione

↦ Esperimere in termini di mem e env le parti sinistra e destra dell'assegnamento:

1. **x=y[1]**, sapendo che y è un vettore.
   env(x) = mem(env(y) + 1)
2. **x[5]=y[5]**, sapendo che x è un puntatore e che y è un puntatore.
   env(x) + 5 = mem(env(y)) + 5
3. **x[4]=y[*p]**, sapendo che x è un vettore e che y è un vettore.
   env(x) + 4 = mem(env(y) + mem(mem(env(p))))
4. ***(*p + 1) = y[a]**, sapendo che y è un vettore.
   mem(mem(env(p)) + 1) = mem(env(y) + mem(env(a)))
5. ***x = &y**, sapendo che x è un puntatore.
   mem(env(x)) = env(y)
6. ***x = y[5]**, sapendo che x è un puntatore e che y è un vettore.
   mem(env(x)) = mem(env(y) + 5)
7. ***x = &(y[*p])**, sapendo che x è un puntatore e che y è un vettore.
   mem(env(x)) = env(y) + mem(mem(env(p)))
8. ***(*x) = &(y[a])**, sapendo che x è un puntatore e che y è un vettore.
   mem(mem(env(x))) = env(y) + mem(env(a))
9. **x[*p] = *(*y)**, sapendo che x è un vettore e che y è un puntatore.
   env(x) + mem(mem(env(p))) = mem(mem(mem(env(y))))
10. ***x = *(*p + z)**, sapendo che x è un puntatore.
    mem(env(x)) = mem(mem(env(p)) + mem(env(z)))
11. **x = y[*p]**, sapendo che y è un puntatore.
    env(x) = mem(env(y) + mem(mem(env(p))))
12. **x[*p] = *(*p+4)**, sapendo che x è un puntatore.
    env(x) + mem(mem(env(p))) = mem(mem(env(p)) + 4)
13. ***(*x) = y**, sapendo che x è un puntatore.
    mem(mem(env(x))) = mem(env(y))

14. ***(*x) = &(y[*p])**, sapendo che x è un puntatore e che y è un vettore.
    mem(mem(env(x))) = env(y) + mem(mem(env(p)))

↦ Scrivere un assegnamento le cui parti di sinistra e destra corrispondo a:

1. **env(x) + mem(mem(env(p)))** e **env(y) + mem(mem(env(p)))**.
   x[*p] = &(y[*p])
2. **mem(env(x))** e **mem(env(y) + mem(mem(env(p))))**
   *x = y[*p]
3. **mem(env(a)) + 5** e **mem(mem(env(y)))**
   a[5] = *y
4. **mem(env(x))** e **mem(mem(env(y)))**
   *x = *y
5. **env(x) + mem(mem(env(p)))** e **env(y)**
   x[*p] = &y
6. **mem(env(x))** e **mem(env(y))**
   *x=y
7. **mem(env(a)) + 5** e **mem(env(y)) + 1**
   a[5] = y[1]
8. **mem(env(a)) + 4** e **mem(mem(env(y)))**
   a[4] = *y
9. **env(x) + mem(mem(env(p)))** e **mem(env(y) + 2)**
   x[*p] = y[2]
10. **env(x)** e **mem(mem(env(y)))**
    x = *y
11. **env(x) + 1** e **mem(env(y))**
    x[1] = y
12. **mem(env(a))** e **mem(mem(env(y)))**
    *a = *y
13. **env(x) + 1** e **mem(env(y)) + 1**
    x[1] = y[1]

2 Domande Generali

2.1 Vero o Falso

Nelle seguenti domande è richiesto di indicare la veridicità dell'affermazione.

↦ Fortran.

〉 f: Il primo Fortran supportava la ricorsione.
〉 f: Il primo Fortran aveva uno heap.
〉 f: Il primo Fortran supportava un'operazione analoga ad una malloc.
〉 v: Nel primo Fortran l'occupazione della memoria di un programma era nota a tempo di compilazione.
〉 v: Nel primo Fortran l'occupazione di memoria di un programma era nota a compile time.
〉 f: Il primo Fortran aveva uno heap e uno stack di attivazione.

↦ Lisp.

〉 f: Il LISP è un linguaggio imperativo.
〉 f: Il Lisp adotta lo scoping statico.
〉 v: Il primo garbage collector è stato promosso nel LISP.

↦ Prolog.

〉 f: Il Prolog puro supporta i cicli for/next.
〉 f: In Prolog la ricerca del massimo in una lista può essere fatta in fomra iterativa (non ricorsiva).
〉 v: Il predicato Prolog member può enumerare i membri di una lista e generare liste.

↦ SQL.

〉 f: SQL è un linguaggio general purpose.

↦ HTTP.

⟩ f: HTTP (senza scripts) è un linguaggio genera purpose.

↦ C++.

⟩ f: Il C++ ha un garbage collector.

⟩ v: In C++ l'ereditarietà multipla può causare un consumo esponenziale di memoria.

↦ C.

⟩ f: In C, l'identificatore x in y=x rappresenta env(x).

⟩ v: In C, l'identificatore &x in y=&x rappresenta env(x).

⟩ v: C è debolmente tipato.

⟩ f: Un linguaggio fortemente e staticamente tipato può avere le union del C.

⟩ f: In C, il sistema dei tipi adotta solo la name equivalence.

⟩ f: In C, il sistema dei tipi adotta solo la structural equivalence.

⟩ v: In C, il sistema deitipi adotta la name equivalence e la structural equivalence.

⟩ f: Il C adotta sempre le structural equivalence tra tipi.

⟩ f: Il C adotta la structural equivalence per le struct e le name equivalence per tutti gli altri tipi.

↦ Java e JVM.

⟩ v: Se in java si usa unicamente il polimorfismo parametrico allora tutti i controlli di tipo avvengono a tempo di compilazione.

⟩ f: In java, se l'unico polimorfismo universale usato è quello per inclusione, allora tutti gli eventuali errori di tipo sono segnalati a tempo di compilazione.

⟩ v: Java non ha metodi analoghi alla funzione free() di C e C++

⟩ v: Se in Java si usa unicamente il polimorfismo parametrico allora tutti i controlli di tipo avvenogno a tempo di compilazione.

⟩ v: La JVM può cambiare(?) classi a host diversi.

⟩ v: La JVM può caricare classi a host diversi.

⟩ v: Subito dopo l'esecuzione del costruttore alle variabili di istanza sono assegnati i valori di default.

⟩ f: I modificatori che precedono una variabile di tipo array si applicano alla variabile array ed anche ai suoi elementi.

⟩ f: Non è possibile dichiarare un attributo senza inizializzarlo.

⟩ v: E' possibile dichiarare un attributo senza inizializzarlo.

⟩ v: Un attributo può essere contemporaneamente static e private.

⟩ v: Un attributo può essere contemporaneamente static e final.

⟩ v: Un attributo static può essere acceduto mediante un riferimento a un oggetto della sua classe di appartenenza.

⟩ f: Un attributo static non può essere acceduto mediante un riferimento a un oggetto della sua classe di appartenenza.

⟩ v: Un attributo static può essere acceduto mediante il nome della sua classe di appartenenza.

⟩ f: Non è possibile dichiarare un array senza indicarne la dimensione.

⟩ f: La lunghezza di un array può essere variata dopo la sua creazione.

⟩ v: La lunghezza di un array non può essere variata dopo la sua creazione.

⟩ v: La lunghezza di un array non può essere variata dopo la sua costruzione.

⟩ f: La dimensione di un array deve essere indicata al momento della dichiarazione dell'array.

⟩ v: Il minimo numero di elementi che un array può contenere è 0.

⟩ v: Un array ha una sola superclasse.

⟩ v: Un array ha un'unica superclasse.

⟩ v: Un array vuoto non può avere riferimento null.

⟩ f: Un array non ha un'unica superclasse.

⟩ f: Un array con riferimento null ha lunghezza zero.

⟩ f: Un array non possiede dei membri.

⟩ f: Un array vuoto può avere riferimento null.

⟩ v: E' possibile dichiarare un array senza indicarne le dimensioni.

⟩ f: In Java non tutti i tipi numerici sono con segno.

⟩ v: Una classe non interna non può essere dichiarata private.

⟩ v: Una classe può essere dichiarata private.

⟩ f: Una classe non interna può essere dichiarata private.

⟩ f: Una classe non può essere dichiarata private.

⟩ f: Nell'overloading due metodi non possono avere lo stesso nome ma diverso tipo di parametri.

⟩ f: Due metodi non possono avere diverso nome ma lo stesso ipo di parametri.

⟩ v: Due metodi non possono differire per il tipo di ritorno.

⟩ v: Quando due metodi hanno lo stesso nome si ha overloading.

⟩ f: Due metodi possono differire per il tipo di ritorno.

⟩ v: Due metodi possono avere diverso nome ma lo stesso tipo di parametri.

⟩ f: Non tutti i tipi di eccezione estendono la classe Throwable.

⟩ f: Non tutti i tipi di eccezione estendono la classe Object.

⟩ v: Non tutti i tipi di eccezione estendono la classe RuntimeException.

⟩ v: Tutti i tipi di eccezione estendono la classe Object.

⟩ f: Tutti i tipi di eccezione estendono la classe RuntimeException.

⟩ v: Un oggetto può non esistere dopo la sua dichiarazione.

⟩ f: Un oggetto può non esistere dopo la sua creazione.

⟩ v: La dichiarazione di un oggetto e la sua creazione possono essere svolte contemporaneamente.

⟩ f: La dichiarazione di un oggetto e la sua creazione possono essere svolte solo in tempi diversi.

⟩ v: La dichiarazione di un oggetto e la sua creazione possono essere svolte in tempi diversi.

⟩ f: La dichiarazione di un oggetto e la sua creazione possono essere svolte solo contemporaneamente.

⟩ f: Un oggetto esiste sempre dopo la sua dichiarazione.

⟩ v: Ogni valore in memoria non è associata ad un tipo di dato particolare.

⟩ f: Ogni valore in memoria è associato ad un tipo di dato particolare.

⟩ f: L'operatore new non restituisce un riferimento all'oggetto appena creato.

↦ Generiche.

⟩ v: L'aliasing coniste nel riferiresti alla stessa locazione con nomi diversi.

⟩ v: Se A è un istanza di B e B è una sottoclasse di allora A è un istanza di C.

⟩ v: Se implementiamo le liste mediante template, i membri di una lista hanno tutto lo stesso tipo (o nel caso object oriented delle sottoclassi di quel tipo.

⟩ v: Tra i compiti del supporto a run time c'è la gestione della memoria.

⟩ f: Un linguaggio puramente funzonale ha i cicli while.

⟩ f: Nel paradigma funzionale si possono usare cicli for e while.

⟩ v: Nel paradigma funzionale puro non ci sono gli assegnamenti.

⟩ v: Nel paradima funzionale puro non ci sono gli assegnamenti.

⟩ v: La funzione membro (elemento, lista) nel paradigma funionale può essere usaa anche per generare tutti gli elementi della lista.

⟩ v: La promozione automatica dei tipi numerici è una forma di polimorfismo ad hoc.

⟩ v: Nei linguaggi funzionali env(x) restitisce il valore di x.

⟩ f: Nei linguaggi funzionali env(x) restituisce una locazione.

⟩ v: In un linguaggio funzionale puro, un idenificatore x rappresenta env(x).

⟩ v: Nei linguaggi funzionali l'ambiente associa direttamente gli identificatori al loro valore.

⟩ f: Nei linguaggi imperativi l'ambiente associa direttamente gli idenificatori al loro valore.

⟩ f: I linguaggi funzionali non sono computazionalmente completi.

⟩ f: I linguaggi funzionali puri non sono computazionalmente completi.

⟩ f: Compilatore e programmi oggettop sono di norma eseguiti dalla stessa macchina.

⟩ f: Nei linguaggi fortemente tipati il type checking avviene tutto a tempo di compilazione.

⟩ f: In un linguaggio fortemente tipato il controllo dei tipi avviene sempre intermente a tempo di compilazione.

⟩ v: In un linguaggio fortemente tipato il controllo dei tipi avviene durante la compilazione e l'esecuzione.

⟩ v: In un linguaggio staticamente tipato il controllo dei tipi avviene sempre interamente a tempo di compilazione ed esecuzione.

⟩ v: In un linguaggio dinamicamente tipato il controllo dei tipi avviene intermente a tempo di eseuzione.

⟩ v: Se il linguaggio è dinamicamente tipato, allora il tipo di una variabile può cambiare durante l'esecuzione del programma.

⟩ f: In un linguaggio debolmente tipato il controllo dei tipi avviene sempre interamente a tempo di compilazione.

⟩ v: Il controllo di correttezza dei downcast richiede controlli a runtime.

⟩ f: Il polimorfismo per inclusione è il più indicato per la definizione di collezione di oggetti omogenei.

⟩ v: Il polimorfismo che permette più controlli a tempo di compilazione è quello parametrico.

⟩ v: Si può accedere alle variabili non locali di una procedura in tempo costante, indipendentemente da quanti record di attivazione devono attraversare.

⟩ f: Il codice oggetto deve essere eseguito da un interprete diverso dala macchina hardware.

⟩ f: Il polimorfismo parametrico puro può generare errori di tipo a runtime.

⟩ f: Le macro hanno un proprio ambiente locale implementato con un record di attivazione.

⟩ f: In un programma che usa le union o altre forme di record varianti il controllo dei tipi può essere fatto interamente a tempo di compilazione.

⟩ v: Il polimorfismo parametrico puro (template) è più adatto del polimorfismo per inclusione per rappresentare insiemi di oggetti omogenei.

⟩ f: Il polimorfismo parametrico puro (template) è più adatto del polimorfismo per inclusioone per rappresentare insieme di oggetti eterogenei.

⟩ v: Il polimorfismo parametrico puro (template) ammette il controllo dei tipi completo a tempo di compilazione.

⟩ f: Il polimorfismo parametrico puro (template) può generare errori di tipo a runtime.

2.2 Domande con risposta

Le seguenti domande erano (forse) parte di questionari a risposta multipla. Verrà pertanto **indicata** solo la risposta esatta.

↦ Nei linguaggi a oggetti l'ambiente non locale di un metodo si può trovare **nello heap e nella zona statica**.

↦ ML supporta l'equivalenza strutturale sulle dichiarazione fatte con **type**

↦ E' vero che nel paradigma funzionale puro non ci sono gli assegnamenti? **sì**.

↦ Il C supporta l'equivalenza per nome **sui tipi primitivi**.

↦ Gli attributi statici di una classe Java sono memorizzati nello Heap? **sì**.

↦ Il controllo di correttezza dei downcast richiede controlli a runtime? **sì**.

↦ L'accesso a una variabile non locale nei linguaggi imperativi può essere realizzato in tempo costante, indipendentemente dalla dimensione dello stack? **?**.

↦ Esistono linguaggi a oggetti senza la gerarchia dei tipi? **no**.

↦ In C le variabili non locali di funzioni e procedure si possno trovare **nello stack**.

↦ Il C supporta l'equivalenza strutturale **sui tipi primitivi**.

↦ Java ha metodi analoghi alla funzione free() di C e C++? **no**.

↦ Nel paradigma imperativo, l'ambiente non locale di procedura e funzioni si può trovare nello heap.

↦ E' vero che in C gli int sono lunghi 32bit? **dipende** (dall'elaboratore).

↦ Le macro (come ad es. le #define del C) hanno un proprio ambiente locale implementato con un record di attivazione? **no**

↦ I parametri IN passati per riferimento **non possono essere modificati**.

↦ In un linguaggio fortemente tipato il controllo dei tipi avviene sempre interamente a tempo di compilazione? **no**.

↦ Il predicato Prolog member può essere utilizzato sia per verificare se un dato elemento appartiene a una lista sia come "generatore", cioè per enumerare tutti i membri della lista uno a uno? **si**.

↦ In un linguaggio staticamente tipato il controllo dei tipi avviene sempre interamente a tempo di compilazione? **sì**.

↦ Il primo Fortran aveva uno stack di attivazione? **no**

↦ C adotta lo scoping statico? **sì**

↦ Il primo Fortran aveva un heap? **no**.

↦ In Java le variabili non locali dei metodi si possono trovare

↦ La dimensione degli oggetti nello heap può essere esponenziale nella altezza della gerarchia delle classi (ovvero nel numero di superclassi di una data classe) **in C++**.

↦ Il comando malloc in C alloca memoria **nello heap**.

↦ Il primo Fortran aveva uno Heap? **no**.

↦ In Java l'ambiente non locale di una classe interna a un metodo si può trovare **sullo stack**.

↦ Il C adotta sempre la structural equivalence tra i tipi? **no**.

↦ Si può accedere alle variabili non locali di una procedura in tempo costante, indipendentemente da quanti record di attivazione si devono attraversare? **sì**.

↦ Un linguaggio fortemente e *staticamente* tipato può avre le union del C (o il costrutto equivalente detto record con varianti di Pascal)? **no.**

↦ Il controllo di correttezza

↦ La dimensione degli oggetti nello heap può essere esponenziale nell'altezza della gerarchia delle classi (ovvero nel numero di superclassi di una data classe) **in C++**.

↦ Il comando new in Java alloca memoria **nello heap**.

↦ L'ambiente associa gli identificatori direttamente al loro valore nei linguaggi **funzionali**.

↦ Quali forme di polimorfismo supporta Java? **Ad hoc, per inclusione, parametrico**

↦ Il C++ l'ereditarietà multipla può causare un consumo esponenziale di memoria? **si**.

↦ Nel primo Fortran l'occupazione di memoria di un programma era nota a tempo di compilazione? **si**.

↦ Java adotta lo scoping statico? **si**.

↦ Il primo Fortran aveva l'equivalente di una malloc? **no**.

↦ L'ambiente associa gli identificatori direttamente al loro valore nei linguaggi **funzionali**.

↦ Nei linguaggi funzionali, env(x) restituisce **il valore di x**.

↦ Compilatore e programmi oggetto sono di norma eseguiti dalla stessa macchina? **no**.

↦ Il C supporta l'equivalence strutturale **sui tipi primitivi**.

↦ Il C++ ha una implementazione **compilata**.

↦ Quando viene invocata una macro (come le #define in C), viene inserito un record di attivazione nello stack? **no**

↦ L'ambiente non locale di una classe interna ad un'altra classe si può trovare **nello heap e nella zona statica**.

↦ Un tipo di dato astratto è **un tipo perfettamente incapsulato**.

↦ Nei linguaggi funzionali env(x) restituisce **il valore di x**.

↦ In Java l'ambiente non locale dei metodi si può trovare **nello heap e nella zona dove sono memorizzate le classi**

↦ Quali forme di polimorfismo supporta Java? **Ad hoc, per inclusione e parametrico**.

↦ La proprietà caratterizzante del paraigma ad oggetti è **la gerarchia dei tipi**.

↦ Un tipo di dato astratto è **un tipo perfettamente incapsulato**.

↦ I parametri IN OUT passati per copia possono **essere letti prima di essere inizializzati** e **possono essere modificati**.

↦ Il polimorfismo che permette più controll a tempo di compilazione è quello **parametrico**.

↦ Esistono linguaggi a oggetti senza a gerarchia di tipi? **no**.

↦ L'ambiente associa gli identificatori a una locazione nei linguaggi **imperativi**.

↦ In Java le variabili non locali dei metodi si possono trovae **nello Heap** e **nella zona statica**.

↦ Esitono linguaggi fortemente e staticamente tipati dove gli identificatori non vanno necessariamente dicharati prima dell'uso? **no**.

↦ La corrttezza dei downcast si può verificare interamente a tempo di compilazione? **no**.

↦ Il predicato Prolog member può essere utiizzato sia per verificare se un dato elemento appartiene a una lista sia come "generaore", cioè per enumerare tuti i membri della lista uno a uno? **si**.

↦ E' vero che in un linguaggio imperativo, un identificatore x in un assegnamento rappresenta env(x)? **dipende**

↦ La dimensione degli oggetti nello heap può essere esponenziale nella altezza della gerarchia delle classi (ovvero nel numero di superclassi di una data classe) in **C++**.

↦ Se A è una sottoclasse di B, le istanze di A sono **anche istanze di b**.

3 Domande sul Codice

4 Indica quale degli assegnamenti e' corretto a tempo di esecuzione

Alcuni suggerimenti:

↦ Quando si upcasta si sale nella gerarchia dell'ereditarietà, quando si downcasta si scende. Ad esempio $Ab \rightarrow A$ è un upcast, mentre $A \rightarrow Ab$ è un downcast.

↦ Non è possibile castare classi di *discendenza* diversa, ad esempio Ab:A e Ac:A;

↦ L'upcast non genera mai errori, se eseguito su classi della corretta discendenza.

↦ Il downcast potrebbe throware ClassCastException nel caso la classe non sia un oggetto della discendenza selezionata.

5 Indica quale e' l'output del programma

Alcuni suggerimenti:

↦ Controlla che non ci siano catch superflui rispetto alle operazioni effettuate nei blocchi try.

↦ Controlla che non ci siano uncatched throw in try/catch/finally.

6 Passaggio Parametri

Negli esercizi di passaggio parametro sarà sufficiente disegnare uno stack di attivazione, per ogni singola funzione richiamata dal programma. Questi esercizi si basano perlopiù sulla logica di esecuzione con variazioni di "visibilità" (scope) e di modalità di passaggio delle variabili: esecuzione statica o dinamica con passaggio in/out/in-out per valore o riferimento.

6.1 Scope delle variabili

Sarà importante tenere in considerazione lo scope delle variabili qual'ora si stesse cercano di utilizzare una variabile non presente a livello locale. In poche parole con lo scope vengono indicate quali siano le variabili di fall-back da utilizzare. Per questi esercizi, esistono solo due tipologie possibili di visibilita: statico e dinamico.

↦ **Dinamico**, lo scope dinamico è il più semplice da rispettare tra i due in quanto la variabile *mancante* è da ricercare all'interno della funzione ad un livello più in alto nello stack di attivazione: il più delle volte, sara sufficiente controllare che sia il chiamante ad avere tale variabile.

↦ **Statico**, per determinare lo scope statico delle variabili è bene prestare attenzione all'indentazione del codice: nel caso si cerchi di accedere ad una delle variabili mancanti nello scope locale, è bene individuare quale sia lo scope che lo contiene *fisicamente*.

6.2 Modalità di passaggio delle variabili

Il passaggio delle variabili cambia in modo signficativo l'esecuzione del programma, in alcuni casi potrebbe addirittura renderne impossibile l'esecuzione. Esistono sei diverse combinazioni di modalità di passaggio delle variabili.

↦ Per **Valore**, **IN**, in questo le variabili che verranno passate alla sottofunzione verranno copiate in una variabile locale, con lo stesso o con un identificativo diverso, qualsiasi modifica effettuata sarà locale. Questa modalità non presenta in alcun caso problemi di compilazione.

↦ Per **Valore**, **OUT**, in questo caso le variabili locali non saranno inizializzate, teoricamente è possibile leggerle ma non è possibile predirre il valore che assumeranno: al termine della procedura, il valore verrà ricopiato nella variabile passata come destinataria.

↦ Per **Valore**, **IN-OUT**, in questo caso le variabili verranno sì copiate all'interno di una variabile locale nella funzione oggetto ma, al ritorno da suddetta funzione, il valore verrà assegnato alle variabili-parametro. E' impossibile avere errori in questa modalità.

↦ Per **Riferimento**, **IN**, le variabili passate in questa modalità permettono la lettura ma NON la scrittura, risultando spesso in errori qual'ora si tentasse di modificarne il valore. Controllare che non ci siano scritture su suddette variabili.

↦ Per **Riferimento**, **OUT**, si comporta identicamente a valore-out: in teoria, si andrebbe ad assegnare direttamente il valore sulla variabile passata come parametro ma, nella pratica dell'esercizio, sarà sufficiente controllare che non vengano effettuate operazioni di lettura su suddetta variabile.

↦ Per **Riferimento**, **IN-OUT**, le variabili passate attraverso questa modalità avranno la peculiarità di modificare il valore di suddette variabili direttamente nello stack di attivazione del chiamante: qualsiasi modifica effettuata sulla variabile in scope locale sarà riflettuta nello scope della funzione chiamante. E' impossibile avere errori in questa modalità.

7 UML

Gli schemi UML forniscono una vista strutturale, cioè statica, del sistema in termini di **Classi**, composte da Attributi ed Operazioni, e **Relazioni**, come associazioni, generalizzazioni….

E' considerabile un'estensione dei diagrammi ER (Entity-Relationship), introducendo agli attributi anche le operazioni.

Una relazione tra una classe A e una classe B in un class diagram implicherà una relazione tra ogni istanza di classe A con un'istanza di classe B.

7.1 Prospettiva

Un diagramma UML può rappresentare diverse prospettive:

↦ Concettuale, che non si preoccupa della vera e propria implementazione ma si limita a studiare i concetti di ciò che si sta andando a rappresentare.

↦ Di Specifica, che studia, a livello di software, le interfacce ma non le implementazioni.

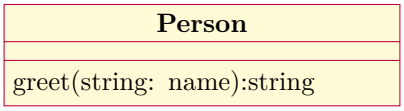
↦ Implementativa, che fa riferimento, cioè, alle classi realmente impiegate e implementate con un linguaggio di programmazione Object Oriented, indicando le strutture dati effettivamente impiegate.

7.2 Classe

Una classe descrive un gruppo di oggetti con caratteristiche comuni: attributi e comportamenti. Gli oggetti (istanze) di una stessa classe differiscono tra loro per i valori degli attributi e per le relazioni che li legano ad altri oggetti.

Essa è rappresentabile come un rettangolo suddiviso in tre fasce orizzontali:

1. La prima fascia ne indica il nome di quella classe.
2. La seconda ne indica tutti gli attributi con le loro caratteristiche in uno schema **nome:tipo=val. default**.
3. La terza ed ultima fascia indica piuttosto le operazioni effettuabili da quella classe attraverso l'utilizzo di uno schema **nome(argomenti):type**.



7.3 Relazioni

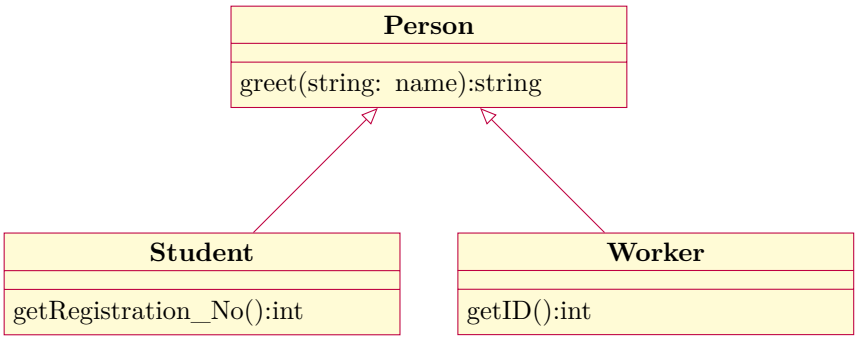
Come già detto, in UML, oltre alle classi, vengono rappresentate le relazioni.

↦ **Generalizzazione is-a**, ogni istanze della classe è anche istanza di tutte le superclassi.

Nei linguaggi Object Oriented, questo meccanismo è descrivibile come **Ereditarietà**, incorporando struttura e comportamento di elementi più generali.

Le classi *figlie* o sottoclassi ereditano gli attributi dalla classe *padre* o superclasse. Essi ereditano anche i metodi che, a differenza degli attributi, potranno essere ridefiniti nella classe figlia (concetto di **Polimorfismo**).

Sarà, inoltre, possibile aggiungere nuovi metodi e attributi alla sottoclasse.



E' bene notare che con una generalizzazione è sempre possibile sostituire il padre con il figlio ma non il viceversa: se stessi lavorando con la classe rettangolo, ereditata da poligono, posso sempre dire di star lavorando con un poligono; viceversa, se stessi lavorando con un poligono potrei non star lavorando con un rettangolo.

↦ **Assocazione**, individua una **connessione** logica tra le classi coinvolte.

Un'associazione è caratterizzata da:

⟩ **Cardinalità**, che indica il numero di istanze che possono essere associate all'altra classe. Sono rappresentate solitamente con un numero (o un simbolo, in caso di cartinalità N) da indicare su una delle due estremità del segmento della relazione.

E' possibile riscontrare alcuni casi particolari di cardinalità:

* Opzionale, da indicare con **0**, potrebbe o meno sussistere una relazione tra una classe di istanza A e un istanza di classe B.
* Multipla, da indicare con *****, che sta ad indicare un numero illimitato di relazioni tra una istanza di classe A e un istanza di classe B.
* Range di Variabilità, da indicare con **i..f**, dove **i** ed **f** sono indicate numericamente (o simbolicamente). E' possibile, mediante l'utilizzo di **virgole**, indicare con più precisione il range (e.g. 1..3,5), **escludendo** determinate cardinalità dal range espresso. (nell'e.g, 4 non è una cardinalità valida)

⟩ **Navigabilità**, che indica il **verso** privilegiato dell'associazione mediante l'utilizzo di una freccia **da-a**, il cui significato è da interpretare in un determinato modo a seconda del contesto:

* **Specifica**, istanza A dipende da istanza B, ma non viceversa.
* **Implementazione**, istanza A possiede un puntatore a istanza B.
* **Concettuale**, in questo caso la dicitura puà benissimo essere ignorata.

La navigabilità può essere unidirezionale o bidirezionale, nel secondo caso è possibile omettere (o meno) la freccia.

⟩ **Ruolo**, una classe può partecipare ad un'associazione con un ruolo specifico. Il ruolo dev'essere **sempre** indicato quando le classi sono coinvolte più volte nella stessa associazione: un esempio tipico potrebbe essere una relazione ricorsiva (istanza classe A in relazione con un'altra istanza classe A).

⟩ **Vincoli Aggiuntivi**, talvolta è necessario esplicitare dei vincoli aggiuntivi in modo da rendere più comprensibile il diagramma delle classi. I vincoli vanno definiti sui legami o sul valore degli attributi e vanno indicati tra **{graffe}** a margine della classe o relazione interessata.

7.4 Classe d'associazione

, è possibile che alcune relazioni possano necessitare di attributi che altrimenti andrebbero scaricate nelle classi coinvolte: Una classe associativa è permette la creazione di una classe derivata da un associazione.

La loro introduzione implica un vincolo aggiuntivo e cioè che per ogni coppia di oggetti associati può esserci un'unica istanza della classe di associazione.

Gli attributi da memorizzate nella classe d'associazioni hanno la particolarità di non essere altrimenti attribuibili a nessuna delle classi coinvolte.

7.5 Aggregazione

, è un caso particolare di associazione molto comune che indica serve ad indicare che un'istanza A è un insieme di elementi di istanze di classa B (e.g. un automobile è composta da vari componenti e, qualora fosse demolita, suddetti componenti continueranno ad esistere). Nel diagramma è indicato con un **rombo vuoto** ad una delle due terminazioni del segmento della relazione.

7.6 Composizione

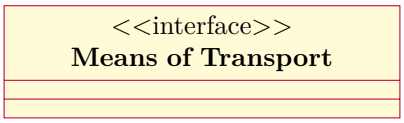
, è considerabile una variazione dell'aggregazione ed implica che i componenti non possano esistere senza il contenitore. Implicitamente, i componenti devono essere associati esclusivamente ad un unico contenitore in una relazione con molteplicità 1.

La distruzione dell'oggetto contenitore implica anche la distruzione degli oggetti contenuti. Solitamente, la composizioni vengono organizzate in una struttura **albero** per questioni di pulizia del diagramma.

7.7 Interfaccia - Realizzazione

. Un'interfaccia viene modellata in modo simile ad una classe ma, in particolare, è composta da soli metodi (e che quindi non presenta alcun attributo).

In UML, un'interfaccia può essere rappresentata da una classe con dicitura **«Interface»** oppure, attraverso la notazione lollipop, un piccolo cerchio con affianco il nome dell'interfaccia.



Una classe relazionata ad un'interfaccia viene definita realizzazione e, a seconda della rappresentazione scelta in precedenza, è rappresentata da una linea tratteggiata terminata da con un triangolo oppure da una semplice linea nel caso della notazione lollipop.

7.8 Visiblità

, indica quali attributi sono accessibili da altri oggetti, viene indicato attraverso un simbolo che precede l'attributo in questione. In particolare le tipologie di visibilità sono:

↦ Pubblica, indicata con il simbolo del +, rappresenta un attributo accessibile da qualsiasi altro oggetto dotato di riferimento all'oggetto contenente l'attributo in questione.

↦ Privata, indicata con il simbolo del **-**, rappresenta un attributo accessibile solo dalla classe di appartenenza.

↦ Protetta, indicata con il simbolo del #, rappresenta un attributo accessibile da tutte le classi che ereditano da quella in cui l'attributo è dichiarato.

↦ Package, indicata con il simbolo del , rappresenta un attributo accessibile solo da classi appartenenti allo stesso package (o una qualche annidazione dello stesso).

Nonostante il concetto della visibilità sia una delle caratteristiche caratterizzanti dei linguaggi Object Oriented, i linguaggi ne fanno un uso *proprio*. Per questo motivo, la visiblità di un diagramma UML deve sempre far riferimento alle regole dello specifico linguaggio che si sta tenendo in considerazione.

7.9 Cosa rappresenta una classe: errori comuni

Una classe deve sempre rappresentare un concetto autonomo e importante, cercando di trovare il giusto livello di astrazione:

↦ Un indirizzo fisico non ha molto senso se non associato ad un entità, come ad esempio un luogo o ad una persona. In questo caso è preferibile indicarlo come un attributo.

↦ E' bene sbilanciarsi nella creazione dei metodi in base all'utilizzo che se ne dovrà fare: se lo scopo è quello di sapere l'indirizzo di una persona, è una decisione ben più saggia ritornare l'intero indirizzo, attraverso un metodo, piuttosto che elencare i singoli componenti dell'indirizzo come oggetti atomici (via, cap, numero civico, città, …), potrebbe risultare più **corretto**.

8 Sequence Diagram

Il diagramma di sequenza descrive uno **scenario** che raccoglie una determinata sequenza di azioni in cui tutte le scelte sono già state effettuate: non possono comparire, quindi, scelte o flussi alternativi. Esso prende anche il nome di **Diagramma degli Eventi** o **Scenario Eventi**.

Un sequence diagram descrive in particolare le relazioni che intercorrono, in termini di messaggi, tra Attori, Oggetti di Business, Oggetti o Entità del sistema che si sta rappresentando.

8.1 Messaggio

Un messaggio è un'informazione che viene scambiata tra due oggetti: solitamente chi invia è l'attore.

Un messaggio può essere di due tipi:

↦ Sincrono, se il mittente rimane in attesa di una risposta.

↦ Asincrono, se il mittente non attende la risposta che può avvenire in un secondo momento, continuando con altre operazioni.

Il messaggio generatosi come risposta ad un messaggio inviato precedentemente è detto **mesaggio di risposta**.

Se il mittente del messaggio è nello stesso tempo il destinario, allora il messaggio si dice ricorsivo.

8.2 Rappresentazione

Un sequence diagram fa uso di elementi di geometria semplici per la sua rappresentazione.

↦ **Le entità** coinvolte sono indicate con dei rettangoli da cui discende una **linea temporale** (orig. lifelines), linee tratteggiate parallele. Sulla linea temporale vengono disegnati, per indicare lo stato delle attività di elaborazione, dei rettangoli.

↦ **I messaggi** sono indicati da delle frecce continue con il contenuto del messaggio sopraindicato. La frecce possono presentare una o due *facciate* a seconda se il messaggio sia asincrono o sincrono.

↦ **I messaggi di risposta** sono indicati da delle frecce tratteggiate con caratteristiche simili a quanto indicato in precedenza.

↦ Vi è inoltre la possibilità di rappresentare l'invio da parte di un entità non ben definita attraverso un cerchio annerito.