Sicurezza dei tipi nei linguaggi

H1 # Introduzione

La gestione dei tipi nei linguaggi è importante per la prevenzione di errori e per la sicurezza del sistema.

H2

> Si definisce **tipo** un insieme di valori omogenei e operazioni da fare:
>
> - Tipi semplici
>   - `int`
>   - `char`
>   - ...
> - Tipi strutturati
>   - classi
> - Funzioni e metodi

Vediamo ora alcuni errori che si possono compiere nella loro gestione.

Errori

H3 ## Errori a livello HW

H2

Confondere dati con programmi

È possibile senza controllo sui tipi (ad esempio in C) usare un dato in una variabile per invocare una funzione, ad esempio:

H4

- definisco una variabile intera x: `int x` ;
- chiamo la funzione `x()` ;

Confondere tipi di dato semplici

È possibile che routine HW che si aspettano certi tipi di dato ne ricevano altri, portando a degli errori nei valori registrati, ad esempio:

H4

- invoco la funzione `float_add(3, 4.5)` con 3 di tipo `int`
- la CPU interpreta `3` come sequenza di bit di tipo `float`
- il risultato può essere errato

H3 Errori semantici

Alcuni esempio possono essere:

- operazioni con tipi primitivi errati
  - `int_add(3, 4.5)`
- gestione dell'ereditarietà tra classi
  - `class Quadrato extends Figura` (`Quadrato` sottoclasse di `Figura`)
  - `Figura a1 = new Quadrato()` ✔
  - `Quadrato b1 = new Figura()` ✘

    `Quadrato` potrebbe avere dei metodi in più che potrei invocare (`b1` è di tipo `Quadrato`) ma non trovare, poiché i metodi si cercano nella classe istanziata (che è `Figura` in questo caso).

Type safety

H2

> Un linguaggio di programmazione L si dice ***type safe*** se non esiste un programma scritto in L che possa violare la *distinzione dei tipi* di L

dove alcuni esempi di **violazione dei tipi** sono:

- confondere `int` e `float`
- chiamare una funzione attraverso un intero
- accedere a una zona di memoria sbagliata
- ...

Un linguaggio molto diffuso che **NON è type safe** è C/C++, poiché prevede:

- tpye casting
- aritmetica dei puntatori
- accesso a zone di memoria errate
- . . .

Mentre linguaggi come Java e LISP sono **type safe**, in quanto effettuano un controllo completo dei tipi.

Bisogna tenere in considerazione che vi è un trade-off tra prestazioni e sicurezza, infatti il C/C++ è più veloce e flessibile di linguaggi type safe.

H3 Controllo dei tipi (*type checking*)

Tra i linguaggi **type safe** distinguiamo due diverse categorie in base a quando viene effettuato il controllo dei tipi:

- ***run-time***
  - Il controllo avviene *durante l'**esecuzione***
    - Esempio in LISP:
      quando esegue l'istruzione `(head x)` dove `head` è una funzione che restituisce il primo elemento di una lista, prima controlla che `x` sia una lista
- ***compile-time***
  - Il controllo avviene *durante la **compilazione***
    - Esempio in ML:
      quando compila una funzione `f(x)` controlla che `f` sia `A -> B` e che `x : A`

Java

Java applica **compile-time**, però quando il compilatore non è sicuro introduce un controllo **run-time**.

H4

```
Figura b = new Quadrato();
Quadrato a = (Quadrato) b;
```

La conversione di `b` in `Quadrato` è corretta solo se `b` è stata effettivamente istanziata come `Quadrato` (cosa che è vera in riga 1). Java non può eseguire questo controllo in compilazione, pertanto viene introdotto in fase di esecuzione.

H3 Pro e contro

Entrambi gli approcci evitano errori di tipo, ma

- il controllo a **run-time** rallenta l'esecuzione, in quanto bisogna controllare le conversioni di tipo ogni volta
- il controllo a **compile-time** limita la flessibilità dei programmi, in quanto tutte le istruzioni devono essere corrette prima dell'esecuzione

Alcuni programmi non corretti a **compile-time** sono invece corretti una volta eseguiti, cioè in **run-time**.

Ad esempio in Java:

```
int x;
if (0 > -1)
    x++;
else
    x = "ciao";
```

Questo non supererebbe la compilazione, in quanto sto assegnando a un intero una stringa, ma se venisse eseguito non causerebbe errori in quanto 0 è sempre maggiore di 1 e quindi non entrerebbe mai nel ramo `else`.

C/C++ e la sicurezza

Come abbiamo visto C/C++ non è un linguaggio type safe. Vediamo quindi in dettaglio le sue debolezze e alcune tecniche per evitare errori.

H2

H3 Dereferenziazione di *null*

Se accedo ad una cella puntata da un puntatore `null` ho `segmentation fault`.

```
int main(){
    int* ptr;
    ptr = 0;
    *ptr = 2;
}
```

Se eseguissi questo codice, otterrei `seg fault`, in quanto il valore del puntatore `ptr` è `null`, dal momento che `0` non è una locazione di memoria valida da poter assegnare a un puntatore. Quando quindi cerco questa locazione, l'OS termina l'esecuzione per accesso a locazioni di memoria errate.

H3 Casting non safe

Il C permette la **conversione non controllata** da un tipo a un altro:

- possibile perdita di informazioni per casting da un sopratipo a un sottotipo
- da intero a funzione (come visto qui)

```
double d;
int i;
i = d;
```

Queste istruzioni sono corrette ma potrei perdere informazioni nel passaggio da `double` a `int`.

H3 Aritmetica dei puntatori

Mediante l'aritmetica dei puntatori possiamo puntare a zone di memoria con tipi *diversi*.

```
int* p;
int x;
x = *(p + i);
```

Supponendo che nella locazione `p+i` vi sia un valore di tipo diverso da `int`, col codice scritto esso diventerà un `int`. Questo significa che posso convertire una variabile in qualsiasi tipo se conosco la sua locazione.

H3 Memoria non safe

Mediante i puntatori si può *accedere a memoria in modo non corretto.*

```
void f(int* p, int i, int v){
    p[i] = v;
}
```

Il codice scrive il valore in `v` nella locazione `p+i`, e questo è un esempio della facilità con cui è possibile accedere in memoria.

Questo tipo di situazione è alla base dell'attacco di buffer overflow.

H3 Casting e puntatori

I puntatori in C sono assimilati a interi, quindi *tramite un cast da interi a puntatori posso accedere a una zona di memoria a piacere*.

Deallocazione esplicita e dangling pointer

Una zona di memoria puntata da un puntatore `ptr` può essere liberata esplicitamente, ma il puntatore `ptr` continuerà a puntare lì! Se l'OS usasse quella locazione, io potrei accedere ai dati dell'OS usando `ptr`.

Dangling pointer sullo stack

Possibilità di avere dangling pointer che puntano a celle dello stack, che potrebbero essere sfruttabili accedendo tramite quel puntatore.

Riassunto violazioni sicurezza del C/C++

- Errori *spaziali* di accesso in memoria
  - Out of bound access
  - Buffer overflow
  - . . .
- Errori *temporali* di accesso in memoria
  - Dangling pointers
- Errori di casting
- Memory leaks