Gestione della memoria

In un linguaggio di programmazione sequenziale emergono diverse problematiche nella gestione della memoria:

- Formazione del **Garbage** (memoria non deallocata non più utile, la cui utilità è *indecidibile* a livello teorico).
- Puntatori a memoria non allocata (dangling pointers).
- Riutilizzo improprio di memoria deallocata. Si suddivide in due casi:
  - Accesso alla memoria già deallocata.
  - Ri-deallocazione della memoria già deallocata (double free).
- Frammentazione della memoria.
- Memoria non allocata.

Nell'ambito concorrente/parallelo, a queste problematiche se ne aggiungono altre. Un esempio significativo è la **Race Condition**, situazione in cui il risultato della computazione dipende dall'ordine di esecuzione dei programmi. Questo può verificarsi in diversi scenari, come quando un thread opera in scrittura mentre altri thread leggono contemporaneamente lo stesso dato.
Il programmatore deve gestire attentamente questi casi, implementando meccanismi di sincronizzazione come i vari sistemi di locking.

Consideriamo ora un esempio di funzione in *C*:

```
T f(..., T x);
```

dove `T` rappresenta un tipo di dato generico. In assenza di informazioni sul tipo di dato, possono presentarsi diverse problematiche inerenti alla memoria:

1. Il dato `T` in output è interamente allocato ad ogni chiamata.

   Non esistono puntatori che collegano input ad output (o viceversa), né puntatori dal dato a variabili globali (dichiarate a top-level, accessibili senza averle prese in input e disponibili anche dopo l'esecuzione della funzione) o statiche (simili alle variabili globali, ma con visibilità limitata all'interno della funzione).

   Solo il chiamante della funzione ha accesso al dato. Sarà quindi responsabilità del chiamante determinare se il dato è ancora utile.

2. L'input è condiviso con l'output.

   Esistono puntatori dall'output all'input. In questo caso, l'input non può essere considerato garbage finché l'output non lo diventa.

3. L'output è condiviso con l'input (caso opposto al precedente).

   Esistono puntatori dall'input all'output. In questo caso, l'output non può essere considerato garbage finché l'input non lo diventa.

4. Combinazione del punto *2* e del punto *3*. Input e output devono diventare garbage contemporaneamente.

5. La funzione mantiene un puntatore all'output (o a parte di esso).

   Esempi di questo caso sono gli oggetti *Singleton* e il pattern *Memoization* (dove i risultati per diversi input vengono memorizzati all'interno di una hash table).

   L'output non può essere considerato garbage finché non lo sono tutti gli output precedenti e quell'input non verrà più richiesto.

Tutti questi casi possono verificarsi nella funzione descritta precedentemente.

Esistono diverse tecniche per la gestione della memoria:

- Nessuna gestione automatica della memoria (come in C).
- Garbage collection automatica.

  Il run-time del linguaggio cerca di approssimare l'utilità di un dato e dealloca i dati che considera garbage.

  Viene implementata un'*euristica* di garbage detection, la cui efficacia varia in base all'algoritmo scelto. In alcuni casi, il programmatore può assistere l'euristica per ottimizzare il riconoscimento del garbage.

  Questa tecnica si divide in due principali categorie:

  - Reference Counting
  - Mark & Sweep
- Gestione esplicita da parte del programmatore (come in Rust, C++).

  Non è presente alcuna euristica di garbage detection, ma è il programmatore che esplicitamente descrive al compilatore gli invarianti di accesso ai dati.
  Il compilatore inserisce quindi il codice necessario per gestire allocazione/deallocazione in base alle specifiche fornite dal programmatore.

Reference Counting
L'euristica del reference counting si basa sul principio che: *se un dato boxed ha 0 puntatori entranti, allora il dato è garbage.*

Per applicare questa euristica è necessario tenere traccia, per ogni dato boxed, del numero di puntatori entranti.

A tal fine, nella prima cella del dato boxed, insieme al tag e al numero di celle (dimensione del dato), viene memorizzato anche un **Reference Counter** (**RC**), ovvero un numero intero che conteggia i puntatori entranti.
Questo valore deve essere maggiore di 0, altrimenti il dato viene considerato garbage e deallocato.

L'euristica utilizzata in questo caso, pur essendo efficace, non è ottimale in tutte le situazioni.

È importante introdurre il concetto di **radice**. Le radici sono celle di memoria sempre accessibili al programma, come lo stack e i registri.

Un problema significativo del reference counting si verifica quando il contatore non può arrivare a 0, impedendo la deallocazione, come nel caso di strutture dati con riferimenti

ciclici (ad esempio, una lista doppiamente linkata). In questi casi si crea garbage ogni volta che si perde il puntatore alla prima cella della lista.

Per risolvere il problema delle strutture dati cicliche, viene introdotto il concetto di **weak pointer**, distinguendolo dai normali puntatori ora denominati **strong pointer**.

Nel caso di liste doppiamente linkate, i puntatori strong vengono utilizzati per riferirsi alla cella successiva nella lista, mentre i puntatori weak vengono impiegati per riferirsi all'elemento precedente.

Grazie a questa distinzione, nel reference counter vengono conteggiati solo i puntatori strong, consentendo una gestione più efficace della garbage detection.

Questo metodo, tuttavia, non risolve completamente le problematiche menzionate in precedenza. L'utilizzo dei puntatori weak richiede una gestione attenta da parte del programmatore.

Inoltre, è necessario prestare particolare attenzione ai puntatori weak che potrebbero riferirsi a celle già deallocate. In questo caso, sono necessarie strutture dati aggiuntive per verificare se il puntatore weak si riferisce ancora a un dato allocato.

Analizziamo ora le operazioni di allocazione e gestione dei dati:

- La memoria è frammentata: sono necessari algoritmi efficaci per la gestione del pool di aree di memoria libera (come Best Fit, First Fit, ecc.). Questo processo ha un costo significativo in termini di spazio/tempo. Una possibile implementazione può essere:

  ```
  p = alloc(size + 1);        // + 1 per la presenza della cella RC
  (*p)[0] = 1;
  ```

  Quando viene allocata una cella, il RC è inizializzato a 1, indicando che esiste un solo riferimento entrante.

- Copia di un puntatore. Una possibile implementazione può essere:

  ```
  if(q != null) {
      (*q)[0]--;

      if((*q)[0] == 0) {
          dealloc(q);
      }
  }

  q = p;
  (*p)[0]++;
  ```

  È necessario aggiornare il valore RC di `q` quando un puntatore viene copiato in quella variabile. Questo può causare una deallocazione a cascata.

  Una possibile implementazione di `dealloc()` può essere, in pseudo codice:

```
dealloc(q) {
    for i = 1 to (*q)[0].size do {
        if(boxed (*q)[i]) {
            (*q)[i]--;

            if((*q)[i] == 0) {
                dealloc((*q)[i]);
            }
        }
    }
    free(q);
}
```

Il costo computazionale della copia di un puntatore risulta proporzionale alla lunghezza della catena di deallocazioni, potenzialmente pari al numero totale di operazioni eseguite dal programma fino a quel momento (definito come **unbounded**).

Il costo in tempo è quindi **O(n)**, dove `n` rappresenta il numero di passi del programma.

Mark & Sweep
L'euristica del mark & sweep, nella sua versione base, afferma che: *un dato non raggiungibile dalle radici viene considerato garbage.*

La fase di **Mark** parte dalle radici e *marka* con dei bit tutto quello che è raggiungibile. Tutto il resto viene considerato garbage.

La fase di **Sweep** invece effettua una sorta di deframmentazione. Tutte le celle markate vengono spostate e deframmentate, mentre le celle non spostate vengono deallocate.

Continuare con lezione 11 marzo.