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 *marca* con dei bit tutti gli oggetti raggiungibili. Tutto ciò che non viene marcato è considerato garbage.

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

Descritta in questo modo, sembra che vengano effettuati due passaggi consecutivi. Questa scelta non sarebbe ottimale, in quanto aumenterebbe notevolmente il costo computazionale. In realtà, l'algoritmo viene implementato come un'unica operazione.

Vengono quindi definiti, oltre alle *radici* citate in precedenza (inizialmente stack e registri), due Heap differenti: uno **Heap di Lavoro** ed uno **Heap Vuoto**.

All'interno dell'Heap di Lavoro vengono allocati i dati utilizzati durante l'esecuzione, mentre l'Heap Vuoto rimane inutilizzato.

Si lavora sempre all'interno dell'Heap di Lavoro finché non si attiva l'algoritmo di garbage collection, che sposterà i *dati vivi* dall'Heap di Lavoro all'Heap inutilizzato.
Effettuato lo spostamento, i due Heap si **scambiano di ruolo**.

Con questa implementazione, il 50% della memoria viene inizialmente riservata ma non utilizzata. Inoltre, gli Heap hanno una dimensione iniziale più piccola, che viene incrementata in base alle esigenze.

L'algoritmo, nella sua versione base, funziona nel seguente modo (un possibile scenario):

- Inizialmente si hanno dei puntatori nelle radici che puntano a celle nello Heap di Lavoro. Queste celle a loro volta possono puntare ad altre celle nello stesso Heap. Sono

inoltre presenti anche celle non più raggiungibili (potenzialmente anche con collegamenti ciclici).

- Definito questo scenario, si itera sulle radici e per ogni cella si controlla se il dato sia *boxed* o *unboxed*. Se il dato è boxed si segue il puntatore per localizzare il dato nello Heap.

  Viene quindi allocato il dato appena trovato all'interno dell'Heap Vuoto. L'Heap Vuoto viene trattato come uno Stack, con un Heap Pointer posizionato in basso che sale man mano che vengono salvati nuovi dati al suo interno. Questo permette di evitare la frammentazione all'interno del nuovo Heap.

  All'interno del dato appena copiato vengono mantenuti anche i puntatori presenti al suo interno (se presenti). I puntatori vengono aggiornati per rispecchiare la nuova posizione del dato spostato nel nuovo Heap.

- Terminata l'iterazione sui diversi grafi che partono dalle radici, tutti i dati non spostati vengono considerati - come dall'euristica - garbage.

  L'Heap di Lavoro viene quindi considerato vuoto, invertendo i ruoli dei due Heap.
  Per "considerato vuoto" non si intende svuotare lo heap fisicamente, ma semplicemente riportare l'heap pointer alla posizione della prima cella.

Terminato l'algoritmo, si avrà l'Heap di Lavoro con tutte le celle allocate sotto l'heap pointer. Questo permette di allocare nuovi dati all'interno dello heap in maniera molto efficiente, come se fosse uno stack.

I linguaggi che allocano molto frequentemente, come i linguaggi funzionali, utilizzano spesso Mark & Sweep.

Questa prima implementazione presenta però dei problemi, come ad esempio i puntatori ciclici che potrebbero portare ad allocazioni infinite delle celle, oppure la perdita della condivisione (sharing) delle celle all'interno del nuovo Heap (dove in precedenza lo sharing era presente).

Per risolvere queste problematiche, viene salvato all'interno del dato che deve essere copiato (nello Heap di Lavoro attuale) il puntatore al nuovo dato copiato sul secondo Heap. Il puntatore viene salvato nella prima cella del dato, dove solitamente si trovano tag e dimensione del dato stesso. Questo permette anche di verificare se la cella è stata già visitata in precedenza.

Analizziamo ora il costo computazionale di questa versione finale dell'algoritmo.
Il costo computazionale in spazio è $\mathbf{O(0)}$, non dovendo allocare nuova memoria aggiuntiva.
Il costo computazionale in tempo è $\mathbf{O(n + m)}$, dove `n` è il numero delle radici ed `m` il numero di dati vivi (dati non considerati garbage).

Il costo in tempo non è ottimale, considerando che il numero di dati vivi presenti possa essere elevato. È quindi necessario applicare alcune modifiche per migliorare questo aspetto.

L'algoritmo Mark & Sweep si attiva in base a diverse strategie, come in momenti di basso utilizzo della CPU o quando l'heap pointer raggiunge un valore elevato nella memoria.

Verifichiamo ora se lo spostamento dei dati in memoria è compatibile con la semantica del nostro linguaggio di programmazione. Esaminiamo quindi diverse operazioni possibili sui puntatori:

- Dereferenziazione di un puntatore: in questo caso non ci sono problemi.
- Somma scalare all'interno dei blocchi: in questo caso non ci sono problemi.
- Sottrazione all'interno dei blocchi: in questo caso non ci sono problemi.
- Confronto di uguaglianza tra due puntatori: in questo caso non ci sono problemi.
- Operazioni di $<$, $>$, $<=$, $>=$: in questo caso **ci sono problemi**.

  Un esempio concreto di utilizzo problematico sono gli alberi binari di ricerca.
- Funzioni hash: in questo caso **ci sono problemi**.

  Il puntatore non può essere utilizzato in modo affidabile come chiave dell'hash.

Gli ultimi due punti ci indicano che non è possibile utilizzare i puntatori come chiavi nei dizionari quando si implementa il Mark & Sweep.

Per risolvere questo problema, è necessario aggiungere al dato boxed un ID univoco, da utilizzare come chiave del dizionario. Questo implica che per leggere il valore della chiave bisogna dereferenziare il dato, il che comporta un costo aggiuntivo di indirezione.

Mark & Sweep Generazionale
Mark & Sweep ha un alto costo computazionale in tempo quando esiste molta memoria viva. Per migliorare questo aspetto, sarebbe utile sapere in modo più preciso quale memoria è destinata a rimanere viva a lungo - e quindi non spostarla - e quale memoria è destinata a diventare garbage rapidamente - per poterla gestire in modo diverso.

Viene quindi utilizzata un'analisi statistica per migliorare questo algoritmo, formulando l'**ipotesi generazionale**.

L'ipotesi generazionale afferma, nella sua forma estrema, che: *un dato x è destinato a rimanere reachable per un periodo molto lungo, oppure è destinato a diventare unreachable rapidamente.*

Viene quindi escluso il caso intermedio, in cui un dato rimane reachable per un periodo medio. L'obiettivo è determinare a quale delle due categorie appartengono i diversi dati, per poterli collocare in maniera ottimale.

Esaminiamo ora una versione dell'algoritmo Mark & Sweep che implementa l'ipotesi generazionale.

Vengono definiti, oltre alle *radici* menzionate in precedenza (inizialmente stack e registri), tre Heap differenti: uno **Heap Young in uso**, uno **Heap Young non in uso** ed uno **Heap Old (Major)**.

Idealmente, i dati che rientrano nella prima categoria (reachable per un lungo periodo di tempo) devono essere spostati nello Heap Old.
L'algoritmo di Mark & Sweep viene utilizzato quindi principalmente tra i due Heap Young.

Generalmente, i due Heap Young hanno dimensioni più ridotte rispetto allo Heap Old.

Esaminiamo ora come i dati si spostano tra i diversi Heap:

- Quando l'algoritmo Mark & Sweep sposta i dati dall'Heap Young in uso a quello Young non in uso, questo passaggio prende il nome di **Minor Collection**.

  Dopo questo primo passaggio, i dati saranno ricompattati all'interno del secondo Heap.

  Il suo heap pointer sarà quindi posizionato sopra tutti i dati già spostati. Questo ci permette di determinare se un dato è recente o meno (verificando se si trova sotto o sopra l'heap pointer).

  Viene quindi posizionato un nuovo puntatore, chiamato **High Water Mark**.
- Successivamente, come in precedenza, i due Heap vengono invertiti, considerando il precedentemente utilizzato come vuoto.
- Nel nuovo Heap attivo, man mano che nuovi dati vengono inseriti, l'heap pointer verrà incrementato. I dati posizionati tra l'*high water mark* e l'*heap pointer* saranno considerati dati più recenti.
- Al secondo passaggio di Minor Collection il comportamento sarà diverso rispetto al passaggio precedente.

  Se un dato si trova sotto l'High Water Mark, verrà spostato nello Heap Old. Altrimenti, verrà spostato nel nuovo Heap Young che diventerà attivo.

Cosa succede se lo Heap Old raggiunge la saturazione?
Se i tre heap hanno la stessa dimensione, si può effettuare una sorta di minor collection tra Old e lo heap vuoto.
Altrimenti, lo Heap Old deve avere una dimensione maggiore rispetto agli altri, richiedendo l'applicazione di una deframmentazione sullo heap stesso.

Analizziamo ora il costo computazionale di questo algoritmo.

All'interno delle radici si possono avere puntatori sia agli Heap Young che allo Heap Old. L'obiettivo è quello di visitare solo i puntatori nelle aree Young. Si può determinare se un puntatore punta allo Heap Old grazie al suo intervallo all'interno della memoria.

Si vogliono quindi visitare solo i grafi presenti negli Heap Young, ignorando i puntatori allo Heap Old.

Il costo computazionale in tempo è quindi $\mathbf{O(n + m)}$, dove `n` è il numero delle radici ed `m` il numero di dati nello Heap Young.
Questo costo risulta essere più efficiente rispetto al precedente, escludendo una parte di dati.

È necessario però fare attenzione ad un caso particolare: una cella nello Heap Old che punta ad un dato nello Heap Young. Questo scenario può verificarsi solo in linguaggi di programmazione con dati mutabili.

Riprendendo l'euristica di Mark & Sweep: *un dato non raggiungibile dalle radici viene considerato garbage.*

Data questa definizione, viene aggiunta al gruppo di *radici* una lista di celle nello Heap Old, ovvero quelle celle che contengono puntatori dallo Heap Old a dati nello Heap Young.

Questo comporta costi maggiori per l'assegnamento di un puntatore, dovendo aggiornare ogni volta la lista precedentemente citata. Questo overhead viene chiamato **Write Barrier**.

Come discusso in precedenza, l'insieme delle radici non è limitato. Un'altra fonte di radici sono i **binding**.

Il binding tra due programmi scritti in linguaggi diversi serve per permettere la comunicazione tra le due parti. Nel caso in cui un linguaggio mantiene un puntatore ad un dato in utilizzo nell'altro linguaggio, quel dato diventa una radice e non deve essere spostato in memoria.

In generale, la sincronizzazione tra due linguaggi rappresenta una fonte potenziale di garbage che richiede particolare attenzione.