5. Storage

5.5. Automatic memory management
Memory management

- Stack Based
- Escape Analysis
- Manual
- Dangling Reference
- Memory Leak
- Automatic
- Garbage Collection
- Copying Collector
- Reference Counting
- Memory Compaction
- Handles
- Manual
- Cautions
- Memory Leak
- Null Pointer Reference
- Automatic
- Algorithms
- Mark and Sweep
Reference counting

Idea a “reference count” (RC) field in every variable

Invariant
- RC is the number of references to the variable.
- The RC of all live variable is positive

Initially In allocation such as:\(^1\):
```java
Thingy t = new Thingy(); // JAVA syntax
```

Set RC of the newly created `Thingy` to 1.

Maintenance
```
Object o1 = new Object();
// Denote the newly allocated object by `O1`;
// Set RC(`O1`) ← 1;
Object o2 = new Object();
// Denote the newly allocated object by `O2`;
// Set RC(`O2`) ← 1
...
Object o2 = o1 // RC(`O1`)++; RC(`O2`) --;
```

De-allocation After each decrement, if `RC(\(O\)) = 0`: (i) de-allocate `O`; (ii) decrement RC for all children of `O`; and, (iii) recursively de-allocate
Pros & cons of reference counting

**Pros**
- Predictable performance
- Smooth execution without interruptions
- Implementable in Manual Memory Management System via smart pointers, or even as part of the language semantics.
- Automatic Memory Management System as part of the garbage collection system.
- Cost is proportional to actual computation, not to memory size

**Cons**
- Cannot deal with circular structures
- Is generally slow, incurring a huge “write barrier”

---

**Fact (Write barrier)**
The formidable write barrier excludes the universal application of RC for memory management

---

\(^2\)the amount of work that needs to be done in each memory write
What is garbage collection?

Definition (Mark & sweep GC algorithm)

*Invented by John McCarthy around 1959 as an enabling technology for *Lisp* implementation*, Garbage Collection (GC) is a part of the program semantics and runtime, which automatically claims back all unused memory.*

In simple words, de-allocation becomes the responsibility of the PL’s runtime system, rather than the programmer’s.

- Programmer never de-allocates memory
- When memory becomes scarce, a GC procedure is applied to collect all unused variables
- *Mark & sweep*: the simplest GC algorithm

*found in Java, Smalltalk, Python, Lisp, ML, Haskell, and most functional, or modern OO languages.*
Why garbage collection?

GC prevents

- Dangling references
- Memory leak
- Heap corruption
- Heap de-fragmentation (with a compacting collector).

Also, GC makes first-class functions value possible.
This is our storage bank, which contains many “cells”. Our interest, though, lies only with “allocated” cells. An allocated cell is called a variable. Some variables follow value semantics; others contain references. Some belong in the runtime stack; others are “global”; the rest are heap allocated. The heap is primarily a list of free blocks! But, the heap also maintains another list, which keeps references to all heap allocated cells! So, we have a list of free blocks and a list of allocated cells. Together, the two lists make the Heap Data Structure.

Now, our variables reference each other, and have many null references, which we will not always show. For garbage collection, we define the Root Set, which contains global variables, and the runtime stack. Starting from the root set, we conduct a mark phase. We first mark all variables in the root set, and then follow their references, to mark variables referenced from the root set. Again, we follow references, to mark variables two references away from the root set. We keep following references, until we mark all variables reachable from the root set.

Consider now on the entire set of variables. Which variables are garbage? In the sweep phase, we collect all variable blocks which are not marked. Recall the “List of Allocated Cells”? Let’s iterate over it! Start at the first node, and iterate, and iterate, and iterate, 1st cell for recycling, and iterate, 2nd cell for recycling, and iterate, and iterate, and iterate, and iterate, and iterate, 3rd cell for recycling, and iterate, 4th cell for recycling, until we are done! All that remains is, to remove the cells destined for recycling from the “List of Allocated Cells”, to claim back the memory they occupy, and to add these memory blocks to the back to the "List of Free Cells". This diagram depicts the main points.

Mark & sweep garbage collection
Summary: mark & sweep garbage collection

Mark  mark all cells as unused

Sweep  unmark all cells in use (stack, global variables), and cells which can be accessed, directly or indirectly, from these

Release  all cells which remain marked
Delicate issues of the marking process

- Do not visit an object more than once
- Do not get stuck in a loop.
- Typical implementations:
  - Breadth-first search
  - Depth-first search
- Marking:
  - Can be done by “raising” a bit in each object
  - More efficient procedure:
    - Initially, all objects are “0”
    - In first collection, marking is by changing the bit to “1”
    - In second collection, marking is by changing the bit to “0”
    - In third collection, marking is by changing the bit to “1”
Stop & copy garbage collection

- Divide the heap into two regions:
  Region I takes all allocations
  Region II is put on hold
- When region I is exhausted, copy live (reachable) variables to region II
- Switch the roles of the two regions
Defragmentation

- Can be done whenever the GC detects defragmentation
- Can be done in each collection cycle:
  - Presumably slower
  - Often performs better due to caching and “locality of reference”.
Predicaments of garbage collection

Memory/Time Resources could be saved using programmers’ knowledge.

Decreased Performance of the[Real]core program

Uneven Performance with “embarrassing pauses” for GC cycles

Unpredictable Performance the program can never know when a GC cycle may start

Not for Real Time which requires predictable performance

Not for Transactions a transaction may time out with no good reason

Hinder Interactiveness pauses can lead to user abandonment

Incompatible with “Resource Allocation Is Initialization” cannot rely on the destructor of a file object to close the file
Some responses to these predicaments

**Generational GC** collects variables at the nursery first, where mortality is high

**Incremental GC** Can perform some computation and resume it later.

**Concurrent GC** Can run concurrently to the program.

**Realtime GC** Obeys time constraints

Concurrency, predictability, etc., always incur a performance toll.
Memory leak in garbage collection?

- GC can only claim reachable variables
- If a programmer forgets to nullify references, then a pseudo memory leak may occur

Define a class `Leak`

```java
public class Leak {
    ... 
}
```

whose contents is:

```java
private Leak next;
private int[] data;

private Leak(Leak next) {
    this.next = next;
    this.data = new int[1<<25];
}

private static Leak cons(Leak l) {
    return new Leak(l);
}
```

```java
public static void main(String[] args) {
    Leak l = new Leak(null);
    final Runtime r = Runtime.getRuntime();
    for (int i = 0; i < 100; ++i) {
        System.out.println(i + ": " + r.freeMemory());
        l = cons(l);
    }
}
```
Output of the above program

0: 123021432
1: 123086952
2: 123152472
3: 123217992
4: 123283512
5: 123349032
6: 123414552
7: 123480072
8: 123545592
9: 70526952

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at Leak.<init>(Leak.java:16)
  at Leak.cons(Leak.java:12)
  at Leak.main(Leak.java:8)
Semantical memory leak\(^3\)

Definition (Semantical garbage)

A variable which the program will never use again, but still keeps a reference to it, is called semantic garbage.

```java
class Huge {
    Huge() { // Constructor:
        // Allocates lots of data and stores
        // it in the newly created object
    }
}
```

```java
void f() {
    Huge semanticGarbage = new Huge();
    heavy.computation(new Indeed(100);
    System.exit(1);
}
```

The semantic garbage predicament

All sophisticated GC algorithms contend in vain against semantic garbage.

\(^3\)Note that the previous example exhausted memory for the sake of demonstration; it did not really create semantic garbage.
GC & the stack: escape analysis

- GC is always slower than stack-based memory management.
- In a pure GC, there are no automatic variables.
- In Java, local variables are:
  - Stack allocated: builtin, atomic types: \texttt{int}, \texttt{double}, \texttt{boolean} etc. (Java forbids
  - Stack allocated: References to classes and arrays.
  - Heap allocated: Classes and arrays (accessed only by references)

Seemingly Innocent Program

```java
void foo() {
    int a[] = new int[1 << 20];
    List<Integer> b = new ArrayList<Integer>();
    // does a gets assigned to global variables?
}
for (int i = 0; i < 1<<20; i++)
    f(); // Lots of GC activity
```

With escape analysis a smart compiler can determine that variables \texttt{a} and \texttt{b} never “escape” function \texttt{foo()}, and then can be safely claimed when this function terminates.