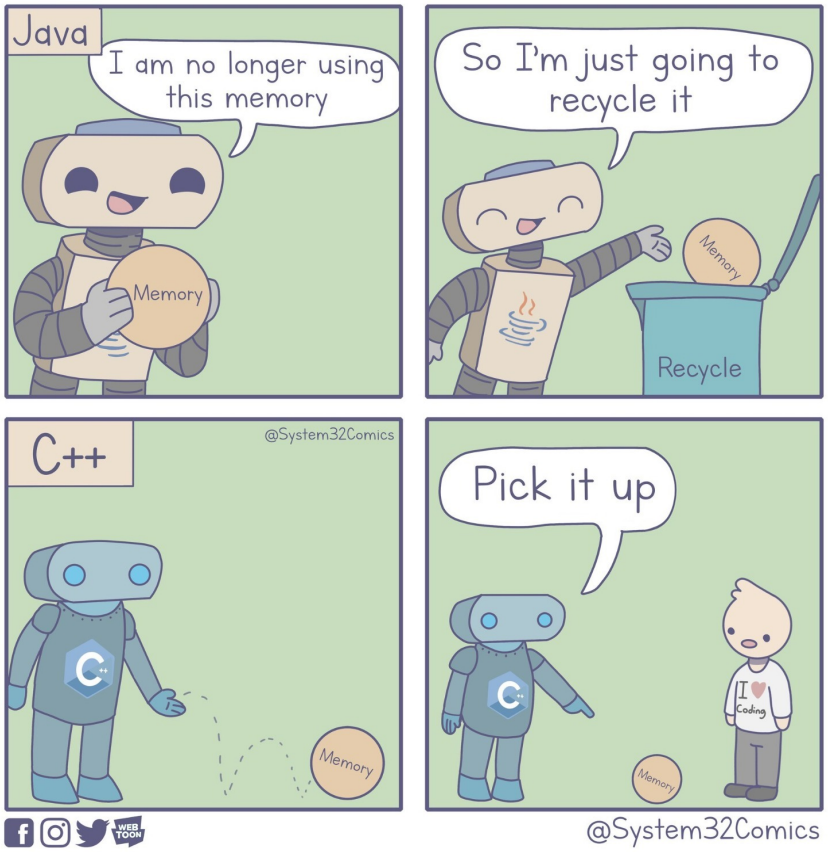
# Recollida de memòria brossa

Facultat d'Informàtica de Barcelona - UPC
Llenguatges de programació




Quadrimestre de tardor, 2020
Hash autor: 80608

# Índex

# 1 Introducció

Els recollidors de memòria brossa (*garbage collectors* en anglès) son un mecanisme propi d'alguns llenguatges de programació que, de forma transparent al programador, gestionen la memòria. Aquest mecanisme de gestió implicita de la memòria va ser dissenyat per primer cop per simplificar la gestió de memòria al llenguatge de programació LISP.

Per una petita aproximació a aquets algoritmes, prenem com a model l'execucció d'un programa en C, representat a la figura 1. Podem veure com la memòria dinàmica d'un programa està gestionada a través de crides al sistema operatiu amb una interfície senzilla.

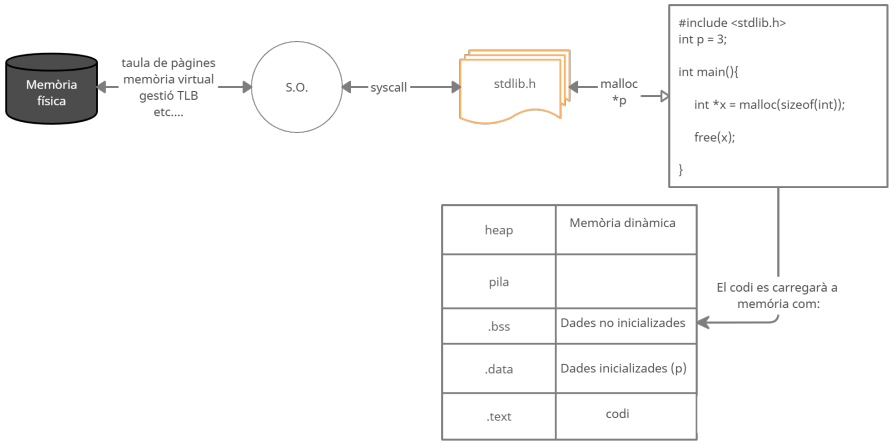



Figura 1: Execucció d'un codi en C/C++

En un codi similar a aquets podem tenir errors en temps d'execucció, per exemple, si provem d'accedir al punter un cop s'ha alliberat. Errors similars, deguts a una mala programació, són els que ataquen els sistemes de recollida de memòria brossa.

Com a contrapartida d'aquesta automatització de la gestió de la memòria, el rendiment pot veure's afectat, doncs els GC no són un algoritme que s'executa alhora que el codi que s'ha programat. En aquest treball s'exposaran els errors que ens sol·lucionen aquets mecanismes.

# 2 Motivació

Per veure els problemas que ens resolen els *garbage collectors*, prenem com exemple el llenguatge C, on la gestió de la memòria és explícita i observem els problemes que ens podem trobar. Ens centrem en la memòria dinàmica del programa, doncs és la que controla el programador.

1. Accés invàlid a memòria : Error al intentar llegir/escriure sobre una regió no reservada.

```
struct alfa{ int d1; int d2;};
char *cP =  (char*) malloc(sizeof(char));
free(cP);
strcpy(cP, "helloWorld");
```

   En aquest codi, estem intentant escriure a una regió de memòria que el SO no té assignada al programa. Com a resultat tenim un error de *Violació del segment*.

2. Fuges de memòria (*Memory leaks*): Si no s'allibera l'espai, el sistema omple la seva regió de *heap* i atura l'execucció del programa.

3. Reserva/alliberament incompatible: Les operacions i mètodes de cada llibreria de cada llenguatge té la seva pròpia gestió, si barregem crides de diferents llibreries, podem reservar i alliberar diferents quantitats d'espai de memòria.

4. Alliberar una regió lliure : Si per error, realliberem una regió ja alliberada obtenim un error.

```
int * iP = malloc(sizeof(int));
free(iP);
free(iP);
```

   S'avorta l'execucció del programa amb un missatge d'error similar a *free(): double free detected in tcache 2*

5. Accés a regions no inicializades: Si no inicializem les dades d'una regió, el resultat pot ser qualsevol valor que estigui anteriorment a la memòria.

```
struct alfa{ int d1; int d2;};
struct alfa * aP = (struct alfa *) malloc(sizeof(struct alfa));
printf("@%x, valor: %i\n",&aP, *aP);//@61fe10 valor: 7213648
int resultat = aP->d1 + 24;
printf("resultat %i \n", resultat); //resultat 7213648
free(aP);
```

   Al codi, veiem que el valor de resultat és arbitrari, ja que el valor mai ha estat inicialitzat.

6. Accessos a altres direccions: Si intentem accedir fora del nostre rang de direccions, el sistema operatiu ens ho impedirà.

Aquets errors són els que no cal parar atenció en llenguatges de programació amb sistemes de GB i, per tant, redueixen els potencials errors en temps d'execucció dels programes.

# 3 Ús dels garbage collector

Vegem un exemple de com un llenguatge amb *garbage collector* com a gestor fa que treballar amb objectes i memòria dinàmica sigui més fàcil pel programador. Veurem exemples d'ùs en Java (llenguatge orientat a objectes pur) i Python (llenguate d'*scripting* amb suport per l'orientació a objectes).

## 3.1 Java

A Java, els objectes son referenciats per les variables del codi (les variables son apuntadors a objecte). L'operador '=' assigna a la variable la direcció de l'objecte. Mirant la referència d'ús del Java[1] els objectes es poden utilitzar així:

```
import java.util.*;

public class myClass{
    int atb1, atb2;

    myClass(int a, int b){
        atb1 = a;
        atb2 = b;
    }
    int add(){return atb1+atb2;}

    public static void main(String []args){
        myClass instance = new myClass(1,2);
        myClass arrayOBJ[] = new myClass[2];
        List<myClass> l = new ArrayList<myClass>();
        System.out.println(x.add());
        int a = arrayOBJ[1].add();  //java.lang.NullPointerException
        x = null;
        System.out.println(x.add());//java.lang.NullPointerException
    }
}
```

[1] *Creating objects (The JAVA™ tutorials)*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/objects.html. Capituls Creating objects, using objects (visitat: 13.12.2020), Creating Objects, Using objects.

Al exemple tenim una definició de classe i un programa per utilitzar-la.

Tots els objectes a Java s'instancien i inicialitzen cridant al constructor amb la paraula clau *new*. Es pot declarar un objecte sense instanciar ni inicialitzar, fet que el llenguatge tracta llançant una excepció si accedim a aquest objectes.

Al exemple s'ha declarat un *array* de dos objectes que no s'instancien mai, si provem de cridar als seus atributs o mètodes, l'execucció del programa acaba amb una *NullPointerException*. En el cas del llenguatge C no teniem aquest comportament si no inicialitzavem les variables o objectes.

La destrucció dels objectes es fa automàticament (o via crida a System.gc()) via *garbage collector*. Un objecte es triable per ser destruït en el moment que l'objecte no té cap referència a memòria.

```
public class Main{
    public static void main(String []args){
        myClass x = new myClass(1,2);
        xx = x;
        System.out.println(x.add());
        x = null;
        System.gc(); //Runtime.getRuntime().gc(); es equivalent
    }
}
```

Al segon exemple, tot i cridar al GC, l'objecte creat no ha de ser eliminat ja que la variable 'xx' apunta encara l'objecte.

## 3.2 Python

A Python, totes les dades del programa estan representades per objectes, que tenen un tipus, un valor i una identitat(id). Aquest id es l'adreça on esta emmagatzemat l'objecte. Com Python es un llenguatge interpretat tots els objectes son dinàmics. Els objectes es creen amb l'operador '=' seguit d'un constructor i es poden eliminar de l'execució amb la crida *del*.

```
>>> x = 23
>>> type(x)
<class int>
>>> hex(id(x))
'0x7ffc7c571960'
>>> del x
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Al exemple, creem un objecte "x", del tipus *int*, comprovem el seu id (direcció on es troba l'objecte) i l'eliminem del programa. Com podem veure, la gestió dels objectes es fa amb una interfície molt fàcil. Sota aquesta crida a *del* hi ha un **garbage collector** que es pot configurar al gust del usuari amb la interfície[4].

# 4 Algorismes i Implementacions

Existeixen diverses implementacions dels recollidors de memòria brossa per diferents casos d'ùs; totes les implementacions segueixen el mateix algoritme base:

## 4.1 Algoritme Bàsic

Totes les implementacions **marquen** les regions de memòria candidates a ser eliminades i les **eliminen**.

Per definir els objectes que es poden marcar, definim els següents estats dels objectes:

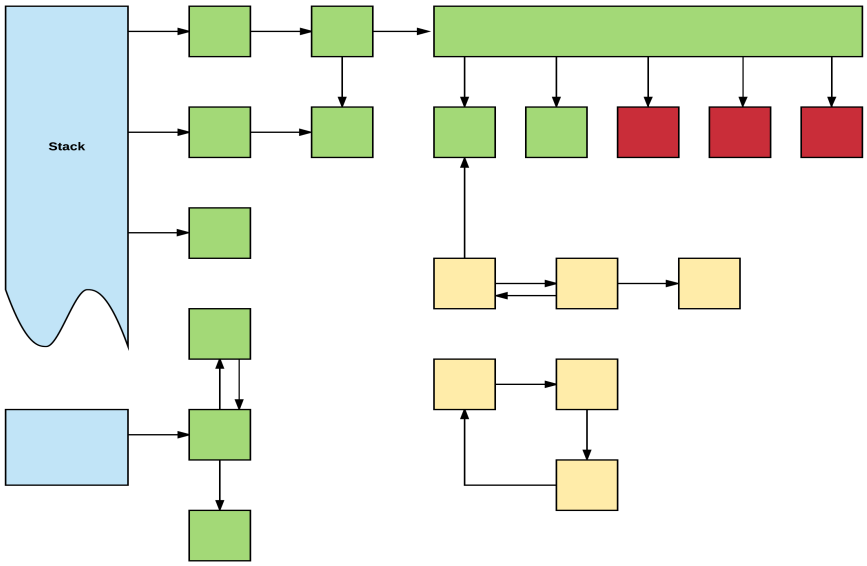



Figura 2: Exemple d'un estat dels objectes en un programa.

Al exemple, extret de blog *The danger of garbage-collected languages*[12], tenim objectes 'vius' pintats de verd, objectes 'morts' pintats de groc i objectes que no es faran servir més pintats de vermell.

Els objectes morts (no tenen cap referència) són els que els algoritmes de GC marquen com a triats. D'altra banda, els objectes que no es faran servir més no poden ser marcats, ja que segueixen sent referenciats.

Un cop marcats els objectes, el següent pas és eliminar els objectes (o regions de memòria). Existeixen dues estratègies:

- Esborrar : S'esborren els objectes i l'assignador de memòria mantè una llista d'espais lliures.
- Esborrar + Compactar: S'esborren els objectes i es compacten a memòria, l'assignador de memòria només manté l'inici de l'espai lliure.

## 4.2 Implementacions a Java

Java té diverses implementacions de GC, per diferents casos d'ús. Totes elles divideixen el heap de la JVM[2] en generacions:

1. Generació jove: Lloc on s'assignen els nous objectes. Tots els nous objectes es crean en la regió *eden*, dins de la generació jove.
2. Generació de supervivents: Dues regions on els objectes de l'eden no eliminats són guardats. Els objectes guanyen 'edad' al avançar entre regions supervivents. Es troba dins de la generació jove.
3. Generació antiga: Regió on es guarden els objectes supervivents de GC.
4. Generació permanent: Metadata de la JVM per tota l'execucció del programa

Existeixen diferents GC depenen quina regió del heap s'omple:

- *Minor* garbage collection: Un cop s'ha omplert l'*eden* es donen les següents accions:
    1. Els objectes que no estan referenciats s'eliminen.
    2. Els altres es mouen a la regió de supervivents, amb edad = 1.
    3. Els objectes supervivents promocionen a generació antiga (donada una certa edad).
    4. Els objectes de la regió de supervivents augmenten en 1 la seva edad (o s'eliminen si no están referenciats).
- *Major* garbage collector: Un cop la generació antiga està plena, involucra a tots els objectes del programa.

[2]La Java virtual machine és l'entorn d'execucció de programes Java

Sobre aquesta distribucció del heap, tenim les següents versions de GC:

- Serial GC: Recol·lector senzill que funciona amb un únic thread per les recol·lecions *minor* i *major*. Utilitza el mètode d'esborrar i compactar; també ordena les regions de supervivents al principi del *heap* per afavorir l'assignació de nous objectes.
- Parallel GC: Recol·lector que utilitza múltiples threads per gestionar l'espai al Heap. Existeixen dos opcions:
  - Versió A: *Multi-threading* per *minor GC* i *single-threading* per *major* GC.
  - Versió B: En les dues GC, s'utilitza *multi-threading*.
- Concurrent Mark Sweep (CMS): El procès de GC funciona concurrentment amb l'execucció del programa, per les *major* GC. Dóna lloc a poques pauses d'execucció del programa[3].

Cal destacar una última versió de GC implementada a Java (desde el JDK7), anomenada G1. Aquesta implementació no segueix les generacions/regions del heap exposades anteriorment.

El G1 *Garbage collector* divideix el heap en regions d'igual tamany, per marcar de forma concurrent i paral·lela els objectes. Un cop marcats els objectes dins les regions, el G1 comença a alliberar espai en les regions que es consideren més lliures. Aquesta versió té molt més rendiment que el CMS.

Els casos d'ús de cada recol·lector recauen sobre el temps que s'inverteix en executar l'algoritme de GC, si la nostra aplicació busca rendiment s'haura de triar el CMS o G1.

[3]Els procesos de major i minor GC detenen tot el flux del programa, anomenat Stop the World Event al àmbit de Java

# 5 Alternatives

Per gestionar l'espai de *heap* a memòria existeixen altres alternatives, no tan genèriques i més especialitzades en aplicacions concretes. Alguns algoritmes són:

## 5.1 Assignació per *frame*

Aquesta estratègia consisteix a reservar memòria i descartar-la sencera en certs esdeveniments; s'ha de poder garantir:

- La vida dels objectes acaba abans dels esdeveniments.
- La mida dels objectes està acotada (per evitar quedar-nos sense memòria).

Aquest algorisme és típic dels videojocs (on la unitat d'esdeveniment és un frame) o les peticions a servidors web (la mida dels objectes 'petició' és constant).

## 5.2 *Pool* d'objectes

Una altra forma de reservar memòria està bassada en reutilitzar els objectes. Inicialment es reserva espai per N objectes (de mida fixa) i s'anoten com no utilitzats, alhora de reservar memòria es busca un objecte lliure. En aquesta construcció s'ha de poder garantir:

- La mida dels objectes és similar.
- S'han de crear i destruir objectes freqüentment, per amortitzar l'espai de control de la *pool* i tota la reserva de memòria inicial.
- Ha d'existir un nombre màxim d'objectes alhora.

## 5.3 Reserva directa a la pila

És possible, per alguns casos on les reserves de memòria dinàmica no són molt grans i la nostra arquitectura de sistema ens ho permet; reservar directament espai dinàmic a la pila. Aquesta estratègia afegeix moltes restriccions als nostres programes i no és gaire utilitzada.

# 6 Avantatges i inconvenients

Un cop descrites totes les tècniques per gestionar la memòria dinàmica, en la següent taula podem comparar els seus arguments a favor i en contra.

| **Solució** | **Pros** | **Contras** |
|---|---|---|
| **Garbage collector** | Facilitats pel programador. Detecció d'errors. | Pèrdua de rendiment. Aturades del programa per els GC. |
| **Gestió del programador** | Proximitat a la memòria. Ús dels apuntadors amb profunditat. | Errors de no incialització. Accesos invàlids. |
| **Assignació per Frames** | Més ràpid que els GC. | Més específic: Objectes de mida acotada, vida curta dels objectes. Cal definir un esdeveniment per esborrar objectes. |
| **Pool d'objectes** | Més ràpid que els GC. | Més específic: Objectes de mides similars, nombre màxim d'objectes. Amortització de l'espai reservat. |

# 7 Conclusió

Com s'ha pogut veure en aquest treball, l'automatització en la gestió de les regions de memòria dinàmica aporten coses bones i dolentes, depenent de l'ús de l'aplicació.

Si tenim una aplicació on els temps de resposta és important (per exemple aplicacions de temps real) és recomanable no utilitzar sistemes de *garbage collection*, ja que podem perdre interaccions amb els usuaris. De forma anàloga per aplicacions on es busca un màxim rendiment. Si aquest fet no importa, podem utilitzar sistemes automàtics.

D'altra banda, per fer prototipat ràpid de codi/aplicacions; els recollidors de memòria brossa simplifiquen el codi i el seu desenvolupament. Un cop fet un prototipat, transferir els algoritmes d'un codi amb recolectors cap a un codi on el programador és responsable del *heap* és mitjanament senzill.

Per tant, podem concloure que els algoritmes de *garbage collection* treuen temps d'execucció als programes i s'ha de tenir cura en quin context s'utilitzen. Molts cops es poden modificar paràmetres i versions dels recollidors per ajustar-ho el mès possible a les nostres aplicacions.

# Bibliografia


[1] *Alternative for Garbage Collector*. URL: https://stackoverflow.com/questions/2598089/alternative-for-garbage-collector. Forum (visitat 16.12.2020).

[2] *Creating objects (The JAVA™ tutorials)*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/objects.html. Capituls Creating objects, using objects (visitat: 13.12.2020).

[3] *Data model*. URL: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types. (visitat: 8.12.2020).

[4] *Garbage Collector interface*. URL: https://docs.python.org/3/library/gc.html. (visitat: 8.12.2020).

[5] *How to avoid, Find (and fix) memory errors in your C/C++ code*. URL: https://www.cprogramming.com/tutorial/memory_debugging_parallel_inspector.html. (visitat: 10.12.2020).

[6] *Java Garbage Collection Basics*. URL: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. (visitat: 8.12.2020).

[7] *JVM Garbage Collectors*. URL: https://www.baeldung.com/jvm-garbage-collectors. (visitat: 13.12.2020).

[8] *Memory allocation patterns used in game development*. URL: Memory%20allocation%20patterns%20used%20in%20game%20development. Forum (visitat: 17.12.2020).

[9] *Object (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#Object--. (visitat: 13.12.2020).

[10] *Object Pool*. URL: https://gameprogrammingpatterns.com/object-pool.html. (visitat: 17.12.2020).

[11] *Referència del llenguatge C i C++*. URL: https://en.cppreference.com/. (visitat: 09.12.2020).

[12] *The dangers of garbage-collected languages*. URL: https://www.lucidchart.com/techblog/2017/10/30/the-dangers-of-garbage-collected-languages/. (visitat: 8.12.2020).

[13] *The structure of the Java Virtual Machine*. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html. (visitat: 10.12.2020).

[14] *Wikipedia: Garbage collection (computer science)*. URL: https://en.wikipedia.org/wiki/Garbage_collection_(computer_science). (visitat: 8.12.2020).