COMPILADORES

RECOLECTOR DE BASURA

MEMORIA DINÁMICA

- Durante la ejecución de los programas a menudo necesitamos pedir memoria en forma dinámica
  - No es posible saber en forma estática cuánta memoria se necesita. Se decide en tiempo de ejecución.
- Se reserva un bloque grande de memoria denominado *heap*.
- Se lleva la contabilidad de cuáles porciones del heap están ocupados y cuáles libres.
- A diferencia de la pila, el tiempo de vida de los objetos en el heap no es dependiente del procedimiento actual o del marco de activación.

MANEJO DE MEMORIA EXPLÍCITA

- La programadora decide la reserva y la liberación de memoria.
  - `malloc(..)`, `realloc(..)`, `free(..)`
- Suele ser fuente de errores:
  - **Olvidar de liberar memoria**: *Memory leaks*. Error especialmente serio en programas que se ejecutan durante un largo tiempo.
  - **Liberar memoria demasiado pronto**: Quedan *dangling pointers*. El programa puede romperse, o peor, seguir funcionando corrompiendo datos.
- No es apropiado para la programación funcional.

MANEJO DE MEMORIA AUTOMÁTICO

- Los recolectores de basura manejan en forma totalmente automática la memoria.
- Mientras la memoria libre es reclamada, pueden ocasionar pausas en la ejecución.
  - Difíciles de implementar en sistemas de tiempo real.
  - Agregan cierto no-determinismo.
- No es posible saber con total certeza si un pedazo de memoria todavía va a ser referenciado. El recolector de basura aproxima en forma conservadora.
  - Se utilizan diferentes estrategias que aproximan de diferentes maneras.
  - Hay tres técnicas básicas.

1- CONTAR REFERENCIAS

- Cada objeto en el heap tiene un campo adicional con el número de referencias a ese objeto.
  - Se inicializa a 1 cuando el objeto es creado.
  - Cuando la cuenta llega a 0 la memoria es liberada (posiblemente decrementando el número de referencias de otros objetos).
- Problemas:
  - Es difícil y costoso mantener la cuenta de referencias.
  - No se lleva bien con estructuras circulares.
- Utilizado, por ejemplo, por Perl, PHP y Python.

RECOLECTORES CON RASTREO

- Las restantes dos técnicas se encuadran dentro de los recolectores con rastreo, o *tracing collectors*.
- Determinan qué objetos del heap están vivos mediante el seguimiento de punteros
- El rastreo comienza a partir de los punteros que se encuentran en el stack.
  - Estos punteros se denominan punteros raíces (root pointers).

2- MARCAR Y BARRER

- Está técnica consiste de dos pasos:
  1. Marcar: se buscan punteros en el stack (raíces) y se los sigue, marcando todos los objetos alcanzables.
  2. Barrer: se recorre la memoria, liberando los objetos no marcados.
- Ventajas:
  - No hay problema con referencias circulares,
  - No hay costo de acceso y escritura de la memoria reservada.
- Problemas:
  - Los objetos nunca se mueven, la memoria se fragmenta.
  - El costo es proporcional a la memoria disponible (barrido).
  - Pausa larga durante el barrido.
- Introducido Por McCarthy para LISP en 1960. Lo utiliza, por ejemplo, Ruby, Java, Erlang, OCaml

3- RECOLECCIÓN CON COPIA

- La memoria se divide en dos partes iguales (A y B)
- Los bloques de memoria se reservan en A.
- Se recorre el heap desde las raíces copiando los objetos a la memoria B.
- Cuando todos los objetos fueron copiados A y B intercambian roles.
- Ventajas:
  - No hay fraccionamiento de memoria
  - El costo es proporcional a la memoria alcanzable
  - No hace falta lista de bloques libres, alcanza con un puntero.
- Desventajas:
  - Se necesita el doble de memoria
- Usado por GHC, JVM, OCaml

RECOLECCIÓN GENERACIONAL

- Hipótesis generacional

"Los objetos más jóvenes tienen menor tiempo de vida"

- Separar la memoria en dos o más partes correspondientes a generaciones.
- Se recolecta la basura en las generaciones jóvenes más frecuentemente que en las más viejas.
- Cuando un objeto de generación i sobrevive k_i recolecciones se lo promueve a la generación $i + 1$
- Tiempo de recolección menor, ya que se recorre menor memoria
- En lenguajes funcionales, una generación vieja **nunca** apunta a una más nueva.
- Utilizado por cualquier GC moderno.
  - OCaml y JVM usan recolección con copia para generaciones jóvenes y "marcar y barrer" para generaciones viejas.

¿CÓMO ENCONTRAR PUNTEROS?

- Etiquetar punteros: Se usa un bit para etiquetar los punteros (eg. OCaml)
  - Pequeño costo de cálculo
  - Tipos más chicos (por ejemplo 2^{31} en lugar de 2^{32})
- Mapas de punteros. El compilador guarda información sobre dónde están los punteros en cada stack frame.
- Aprovechar el tipado estático y proveer funciones de recorrido específicas (JVM)
- Estas últimas requieren que el compilador y el recolector de basura se pongan de acuerdo.
- Perder precisión y adivinar si es un puntero o no (recolector conservador).

RECOLECTOR CONSERVADOR

- Recolector de basura Boehm-Demers-Weiser (a veces llamado simplemente Boehm)
  - “Garbage collection in an uncooperative environment” (Boehm, et al., 1988)
- No usa las direcciones bajas del espacio virtual
- No etiqueta: todo lo que es un valor alineado es un puntero
- No requiere lista de bloques libres, no hay copia de datos.
- Total independencia entre compilador y recolector de basura.
- Al ser impreciso puede haber memoria que nunca se libera.

IMPLEMENTACIÓN

- Biblioteca de C disponible en https://www.hboehm.info/gc/
- Utiliza un algoritmo de marcar y barrer.
- Uso:
  - Agregar `#include <gc.h>`
  - Reemplazar `malloc` por `GC_malloc`, `realloc` por `GC_realloc`.
  - Eliminar cualquier `free`.
  - Al compilar linkear la biblioteca con `-lgc`.

RESUMEN

- Los recolectores de basura proveen un manejo automática de memoria
- Tres técnicas básicas
  - Contar referencias
  - Marcar y Barrer
  - Recolección con copia
- Recolección generacional
- Recolector de Boehm
  - El usado en nuestro proyecto