CSc 453

Compilers and Systems Software

24 : Garbage Collection

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Dynamic Memory Management

- The run-time system linked in with the generated code should contain routines for allocation/deallocation of dynamic memory.

Pascal, C, C++, Modula-2
: **Explicit deallocation** of dynamic memory only. I.e. the programmer is required to keep track of all allocated memory and when it's safe to free it.

Eiffel
: **Implicit deallocation** only. Dynamic memory which is no longer used is recycled by the **garbage collector**.

Ada
: Implicit **or** explicit deallocation (implementation defined).

Modula-3
: Implicit **and** explicit deallocation (programmer's choice).

Memory Management

Memory Management

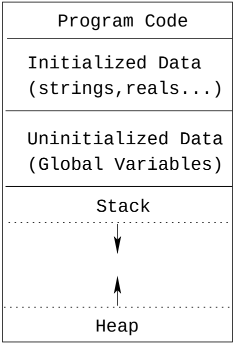
- In a language such as C or Pascal, there are three ways to allocate memory:
  1. Static allocation. Global variables are allocated at compile time, by reserving
  2. Stack allocation. The stack is used to store activation records, which holds procedure call chains and local variables.
  3. Dynamic allocation. The user can create new memory at will, by calling a **new** or (in unix) **malloc** procedure.
- The compiler and run-time system divide the available address space (memory) into three sections, one for each type of allocation:

Memory Management...

- 1. The static section is generated by the compiler and cannot be extended at run-time. Called the `uninitialized data section` in unix's `a.out`.
  2. The stack. The stack grows and shrinks during execution, according to the depth of the call chain. Infinite recursion often leads to stack overflow. Large parameters can also result in the program running out of stack space.
  3. The heap. When the program makes a request for more dynamic memory (by calling **malloc**, for example), a suitable chunk of memory is allocated on the heap.

Memory Management...

- Static allocation – Global variables
- Stack allocation – Procedure call chains, Local variables.
- Dynamic allocation – **NEW**, `malloc`, On the heap.



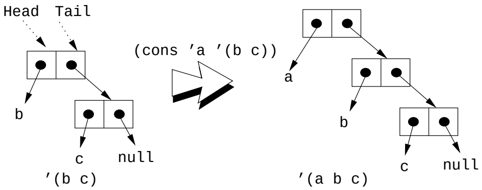
Interface to Dynamic allocation

C, C++: `char* malloc(size)` and `free(char*)` are standard library routines.

Pascal: new(*pointer var*) and dispose(*pointer var*) are builtin standard procedures.

Java: new(*class name*) is a standard function.

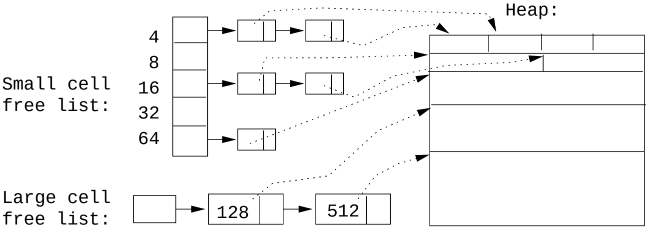
LISP: cons creates new cells:



Explicit Deallocation

Explicit Deallocation

- Pascal's `new/dispose`, Modula-2's `ALLOCATE/DEALLOCATE`, C's `malloc/free`, C++'s `new/delete`, Ada's `new/unchecked_deallocation` (some implementations).
- Problem 1: Dangling references: `p=malloc(); q=p; free(p);`.
- Problem 2: Memory leaks, Heap fragmentation.



Implicit Deallocation

Implicit Deallocation

- LISP, Prolog – Equal-sized cells; No changes to old cells.
- Eiffel, Modula-3 – Different-sized cells; Frequent changes to old cells.
- When do we GC?

  Stop-and-copy Perform a GC whenever we run out of heapspace (Modula-3).

  Real-time/Incremental Perform a partial GC for each pointer assignment or `new` (Eiffel, Modula-3).

  Concurrent Run the GC in a separate process.

Implicit Deallocation. . .

- Fragmentation – Compact the heap as a part of the GC, or only when the GC fails to return a large enough block.
- Algorithms: Reference counts, Mark/ssweep, Copying, Generational.

Garbage Collection Problems

Finding the Object Graph

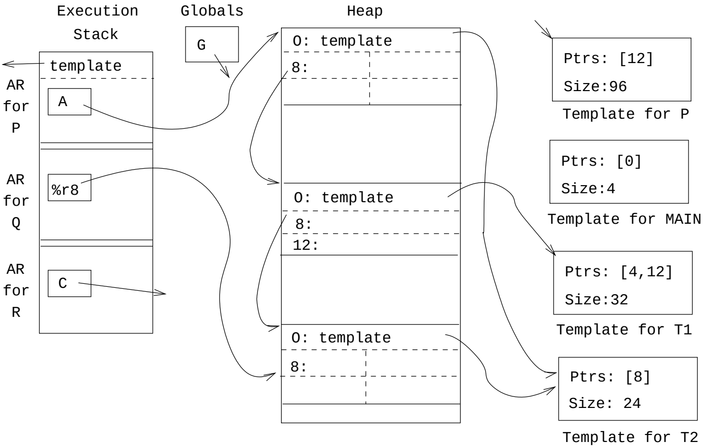
Finding the roots: The dynamic objects in a program form a **graph**. Most GC algorithms need to traverse this graph. The **roots** of the graph can be in

1. global variables
2. registers
3. local variables/formal parameters on the stack.

Hence, the compiler must communicate to the GC which registers/variables contain roots.

Finding internal pointers: Structured variables (arrays, records, objects) may contain internal pointers. These must be known to the GC so that it can traverse the graph. Hence, the compiler must communicate to the GC the **type** of each dynamic object and the internal structure of each type.

Finding the beginning of objects: What happens if the only pointer to an object points somewhere in the middle of the object? We must either be able to find the beginning of the object, or make sure the compiler does not generate such code.


Execution Stack
Globals
Heap
template
AR for P
A
AR for Q
%r8
AR for R
C
G
0: template
8:
0: template
8:
12:
0: template
8:
Ptrs: [12]
Size:96
Template for P
Ptrs: [0]
Size:4
Template for MAIN
Ptrs: [4,12]
Size:32
Template for T1
Ptrs: [8]
Size: 24
Template for T2

Pointer Maps

- The internal structure of activation records & structured variables is described by run-time templates.
- Every run-time object has an extra word that points to a *type descriptor* (or *Temaplate*), a structure describing which words in the object are pointers. This map is constructed at compile-time and stored statically in the data segment of the executable.

Pointer Maps...

- When the GC is invoked, registers may also contain valid pointers. The compiler must therefore also generate (for every point where the GC may be called) a *pointer map* that describes which registers hold live pointers at this point. For this reason, we usually only allow the GC to run at certain points, often the points where **new** is called.
- We must also provide pointer maps for every function call point. A function P may call Q which calls **new** which invokes the GC. We need to know which words in P's activation record that at this point contain live pointers.

Pointer Maps...

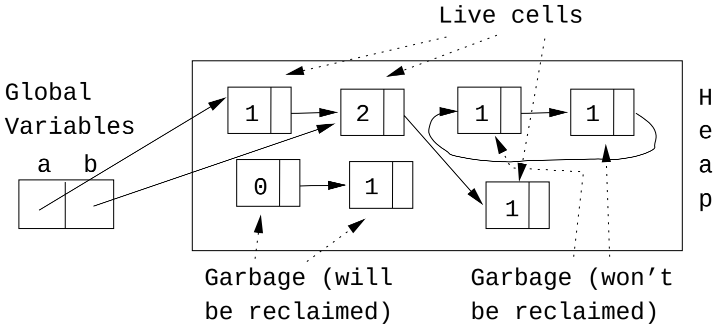
- How does the GC look up which pointer map belongs to a particular call to procedure P at a particular address a? The pointer maps are indexed by the return address of P! So, to traverse the stack of activation records, the GC looks at each frame, extracts the return address, finds the pointer map for that address, and extracts each pointer according to the map.

GC Algorithms: Reference Counts

Algorithm: Reference Counts

- An extra field is kept in each object containing a count of the number of pointers which point to the object.
- Each time a pointer is made to point to an object, that object's count has to be incremented.
- Similarly, every time a pointer no longer points to an object, that object's count has to be decremented.
- When we run out of dynamic memory we scan through the heap and put objects with a zero reference count back on the free-list.
- Maintaining the reference count is costly. Also, circular structures (circular linked lists, for example) will not be collected.

- Every object records the number of pointers pointing to it.
- When a pointer changes, the corresponding object's reference count has to be updated.
- GC: reclaim objects with a zero count. Circular structures will not be reclaimed.



NEW(p) is implemented as:

```
malloc(p); p↑.rc := 0;
```

p↑.next:=q is implemented as:

```
z := p↑.next;
if z ≠ nil then
   z↑.rc--; if z↑.rc = 0 then reclaim z↑ endif;
endif;
p↑.next := q;
q↑.rc++;
```

- This code sequence has to be inserted by the compiler for *every* pointer assignment in the program. This is very expensive.

GC Algorithms: Mark-and-Sweep

Algorithm: Mark and Sweep

- The basic idea behind **Mark-and-Sweep** is to traverse and mark all the cells that can be reached from the **root cells**.
- A **root cell** is any pointer on the stack or in global memory which points to objects on the heap.
- Once all the live cells (those which are pointed to by a global variable or some other live cells) have been marked, we scan through the heap and separate the live data from the garbage.
    - If we are dealing with equal size objects only (this is the case in LISP, for example) the we scan the heap and link all the unmarked objects onto the free list. At the same time we can unmark the live cells.

Algorithm: Mark and Sweep. . .

- - If we have cells of different sizes, just linking the freed objects together may result in heap fragmentation. Instead we need to compact the heap, by collecting live cells together in a contiguous memory area on the heap and doing the same with the garbage cells in another area.

Marking Phase:

1. Mark all objects `unmarked`.
2. Find all roots, i.e. heap pointers in stack, regs & globals.
3. Mark reachable blocks using a **depth first search** starting at the roots.
   1. DFS may run out of stack space!
   2. Use non-recursive (Deutsch-Schorr-Waite) DFS.

Scanning Phase:

same-size-cells
: Scan heap and put un-marked (non-reachable) cells back on free-list.

different-size-cells
: Compact the heap to prevent fragmentation.

Marking Phase

Marking Phase

- A straight-forward implementation of mark and sweep may run into memory problems itself! A depth-first-search makes use of a stack, and the size of the stack will be the same as the depth of the object graph.
- Remember that the stack and the heap share the same memory space, and may even grow towards eachother.
- So, if we're out of luck we might run into this situation:
  - the heap is full (otherwise we wouldn't be gc:ing!),
  - the object graph is deep,
  - we run out of stack space during the marking phase.

  We're now out of memory alltogether. Difficult to recover from!

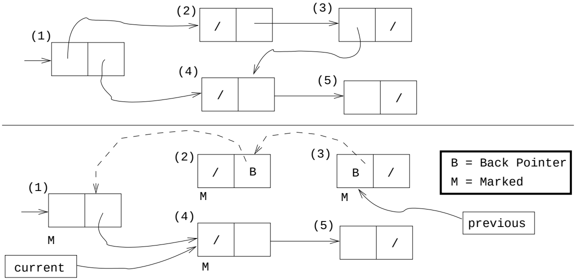
Marking Phase. . .

- Fortunately, there is a smart algorithm for marking in constant space, called the **Deutsch-Schorr-Waite algorithm**. Actually, it was developed simultaneously by Peter Deutsch and by Herbert Schorr and W. M. Waite.
- The basic idea is to store the DFS stack in the object graph itself. When a new node (object) is encountered
  1. we set the "marked"-bit to 1,
  2. the node (object) is made to point to the previous node,
  3. two global variables `current` and `previous` are updated.

  `current` points to the current cell, `previous` to the previously visited cell.

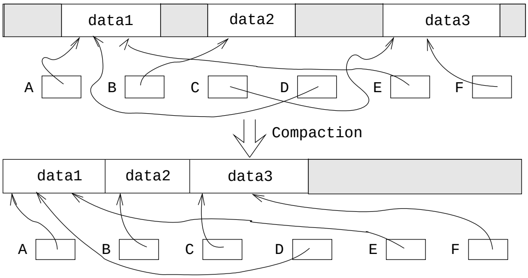
Marking: "Look Ma, No Stack!"

- Use **pointer reversal** to encode the DFS stack in the object graph itself.
- When the DFS reaches a new cell, change a pointer in the cell to point back to the DFS parent cell. When we can go no deeper, return, following the back links, restoring the links.



Compaction Phase

Sweeping: Compaction



1. Calculate the **forwarding address** of each cell.
2. Store the forwarding address of cell B in B.forw_addr.
3. If p points to cell B, replace p with B.forw_addr.
4. Move all cells to their forwarding addresses.

GC Algorithms: Copying Collection

Copying Collection

- Even if most of the heapspace is garbage, a mark and sweep algorithm will touch the entire heap. In such cases it would be better if the algorithm only touched the live objects.
- **Copying collection** is such an algorithm. The basic idea is:
  1. The heap is divided into two spaces, the **from-space** and the **to-space**.
  2. We start out by allocating objects in the **from-space**.
  3. When **from-space** is full, all live objects are copied from **from-space** to **to-space**.
  4. We then continue allocating in **to-space** until it fills up, and a new GC starts.

Copying Collection...

- An important side-effect of **copying collection** is that we get automatic compaction – after a collection **to-space** consists of the live objects in a contiguous piece of memory, followed by the free space.
- This sounds really easy, but $\cdots$:
  - We have to traverse the object graph (just like in mark and sweep), and so we need to decide the order in which this should be done, depth-first or breadth-first.
  - DFS requires a stack (but we can, of course, use pointer reversal just as with mark and sweep), and BFS a queue. We will see later that encoding a queue is very simple, and hence most implementations of copying collection make use of BFS.

Copying Collection...

- This sounds really easy, but $\cdots$
  - An object in **from-space** will generally have several objects pointing to it. So, when an object is moved from **from-space** to **to-space** we have to make sure that we change the pointers to point to the new copy.

Copying Collection...

- Mark-and-sweep touches the entire heap, even if most of it is garbage. Copying collection only touches live cells.
- Copying collection divides the heap in two parts: **from-space** and **to-space**.
- **to-space** is automatically compacted.
- How to traverse object graph: BFS or DFS?
- How to update pointers to moved objects?

Algorithm:

1. Start allocating in **from-space**.
2. When **from-space** is full, copy live objects to **to-space**.
3. Now allocate in **to-space**.

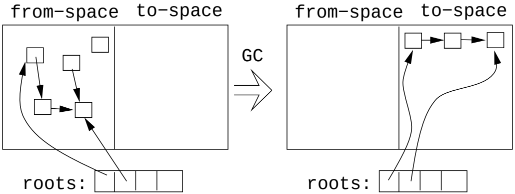
Traversing the Object Graph:

- Most implementations use BFS.
- Use the **to-space** as the queue.

Updating (Forwarding) Pointers:

- When an object is moved its new address is stored **first** in the old copy.

Example:

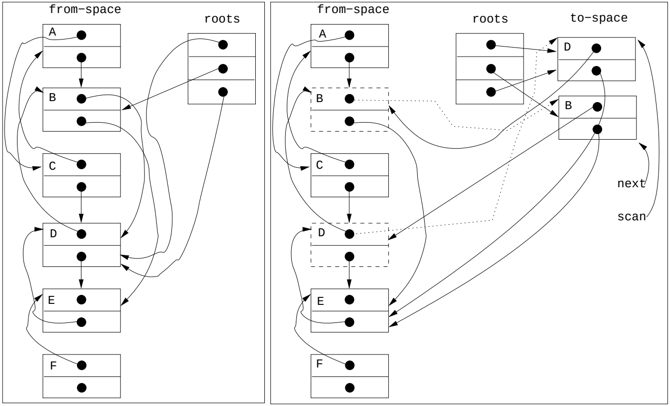


Algorithm:

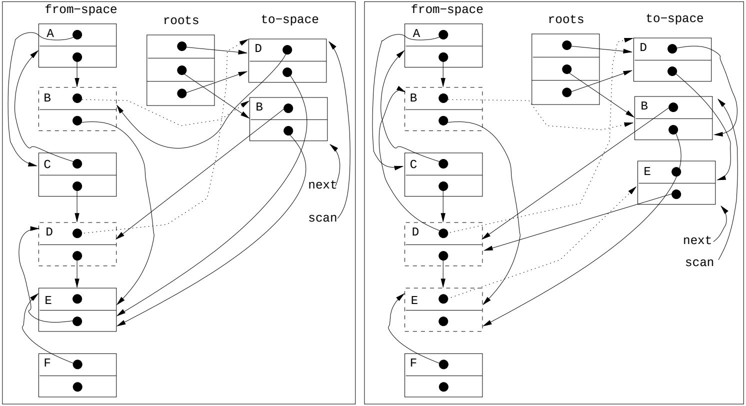
1. `scan := next := ` **ADDR**`(to-space)`
   - [`scan` $\cdots$ `next`] hold the BFS queue.
   - Objects above `scan` point into `to-space`. Objects between `scan` and `next` point into **from-space**.
2. Copy objects pointed to by the root pointers to `to-space`.
3. Update the root pointers to point to `to-space`.
4. Put each object's new address first in the original.
5. Repeat (recursively) with all the pointers in the new `to-space`.
   1. Update `scan` to point past the last processed node.
   2. Update `next` to pointe past the last copied node.

   Continue while `scan < next`.

Copying Collection Example. . . (A)



Copying Collection Example. . . (B)



GC Algorithms: Generational Collection

Generational Collection

- Works best for functional and logic languages (LISP, Prolog, ML, . . .) because
  1. they rarely modify allocated cells
  2. newly created objects only point to older objects (`(CONS A B)` creates a new two-pointer cell with pointers to old objects),
  3. new cells are shorter lived than older cells, and old objects are unlikely to die anytime soon.

Generational Collection...

- Generational Collection therefore
  1. divides the heap into **generations**, G_0 is the youngest, G_n the oldest.
  2. allocates new objects in G_0.
  3. GC's only newer generations.
- We have to keep track of back pointers (from old generations to new).

Functional Language:

$$
\begin{array}{l}
\texttt{(cons 'a '(b c))} \\
\qquad\Updownarrow \\
t_1\text{:}\quad \texttt{x} \leftarrow \texttt{new '(b c);} \\
t_2\text{:}\quad \texttt{y} \leftarrow \texttt{new 'a;} \\
t_3\text{:}\quad \texttt{return new cons(x, y)}
\end{array}
$$

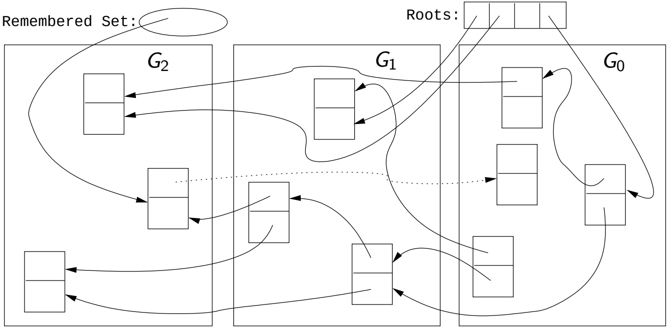
- A new object (created at time t_3) points to older objects.

Object Oriented Language:

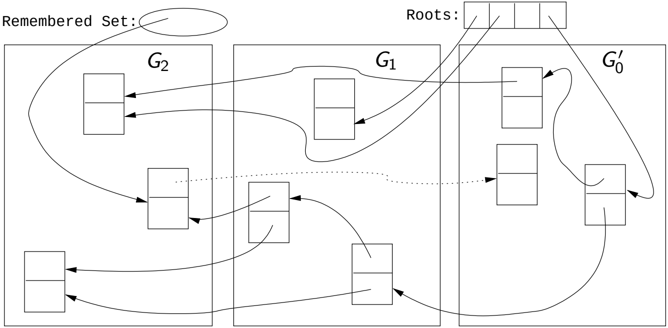
$$
\begin{array}{l}
t_1\text{:}\quad \texttt{T} \leftarrow \texttt{new Table(0);} \\
t_2\text{:}\quad \texttt{x} \leftarrow \texttt{new Integer(5);} \\
t_3\text{:}\quad \texttt{T.insert(x);}
\end{array}
$$

- A new object (created at time t_2) is *inserted into* an older object, which then points to the news object.

Generational Collection. . .



Generational Collection – After GC(G_0)



Generational Collection...

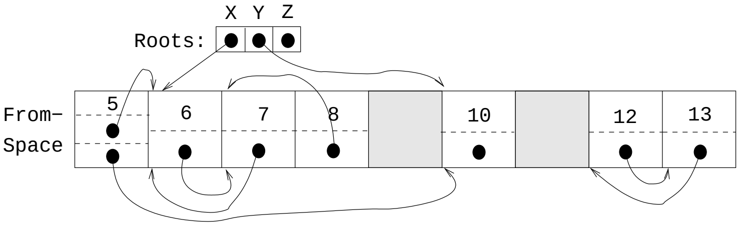
- Since old objects (in $G_n \cdots G_1$) are rarely changed (to point to new objects) they are unlikely to point into G_0.
- Apply the GC only to the youngest generation (G_0), since it is most likely to contain a lot of garbage.
- Use the stack and globals as roots.
- There might be some **back pointers**, pointing from an older generation into G_0. Maintain a special set of such pointers, and use them as roots.
- Occasionally GC older ($G_1 \cdots G_k$) generations.
- Use either mark-and-sweep or copying collection to GC G_0.

Homework

Exam Problem

1. Why is generational collection more appropriate for functional and logic languages (such as LISP and Prolog), than for object-oriented languages (such as Eiffel and Modula-3)?
2. The heap in the figure on the next slide holds 7 objects. All objects have one integer field and one or two pointer fields (black dots). The only roots are the three global variables `X`, `Y`, and `Z`. Free space is shaded. Show the state of `To-Space` after a copying garbage collection has been performed on `From-Space`. Note that several answers are possible, depending on the visit strategy (Depth-First or Breadth-First Search) you chose.

Exam Problem I. . .



Exam Problem...

1. Name five garbage collection algorithms!
2. Describe the **Deutsch-Schorr-Waite algorithm**! When is it used? Why is it used? How does it work?
3. What are the differences between **stop-and-copy**, **incremental** and **concurrent** garbage collection? When would we prefer one over the other?

Summary

Readings and References

- Read the Tiger book:
  Garbage Collection pp. 257–282
- Topics in advanced language implementation, Chapter 4, Andrew Appel, Garbage Collection. Chapter 5, David L. Detlefs, Concurrent Garbage Collection for C++. ISBN 0-262-12151-4.
- Aho, Hopcroft, Ullman. Data Structures and Algorithms, Chapter 12, Memory Management.
- Nandakumar Sankaran, A Bibliography on Garbage Collection and Related Topics, ACM SIGPLAN Notices, Volume 29, No. 9, Sep 1994.
- J. Cohen. Garbage Collection of Linked Data Structures, Computing Surveys, Vol. 13, No. 3, pp. 677–678.