

CSc 520 — Principles of Programming Languages

6 : Memory Management — Heap Allocation

Christian Collberg
Department of Computer Science
University of Arizona
collberg+520@gmail.com

Copyright © 2008 Christian Collberg

January 30, 2008

1 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).

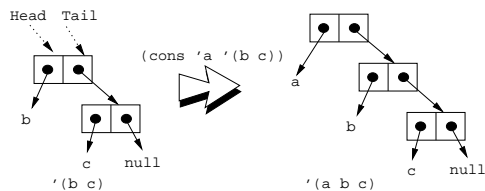
2 Interface to Dynamic allocation

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

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

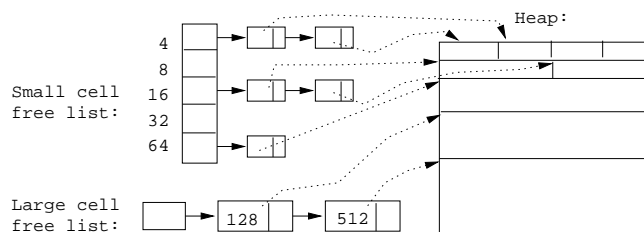
Java: `new(class name)` is a standard function.

LISP: `cons` creates new cells:



3 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.



4 Memory Leaks

```

DEFINITION MODULE Complex;
  TYPE T;
  PROCEDURE Create (Re, Im : REAL) : T;
  PROCEDURE Add (A, B : T) : T;
END Complex.

```

```

IMPLEMENTATION MODULE Complex;
  TYPE T = POINTER TO RECORD
    Re, Im : REAL; END;
  PROCEDURE Create (Re, Im : REAL) : T;
  BEGIN
    NEW(x); x↑.Re := Re; x↑.Im := Im;
    RETURN x; END Create;
  PROCEDURE Add (A, B : T) : T;
  BEGIN
    NEW(x); x↑.Re := ...; x↑.Im := ...;
    RETURN x; END Add;
END Complex;

```

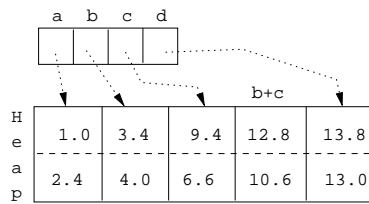
5 Memory Leaks...

```

MODULE Use;
  IMPORT Complex;
  VAR a,b,c,d : Complex.T;
BEGIN
  a := Complex.Create(1.0, 2.4);
  b := Complex.Create(3.4, 4.0);
  c := Complex.Create(9.4, 6.6);
  d := Complex.Add(a,Complex.Add(b,c));
END Use.

```

- `Complex.Add(b, c)` creates a new object which can never be reclaimed.

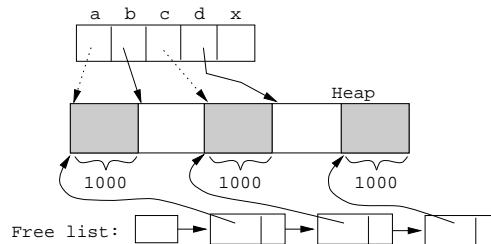


6 Fragmentation

```

VAR a, b, c, d : POINTER TO
    ARRAY [1..1000] OF BYTE;
VAR x : POINTER TO
    ARRAY [1..2000] OF BYTE;
BEGIN
    NEW(a); NEW(b); NEW(c); NEW(d);
    DISPOSE(a); DISPOSE(c); NEW(x);

```



- Without compaction the last allocation will fail, even though enough memory is available.

7 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 heap space (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.

8 Implicit Deallocation...

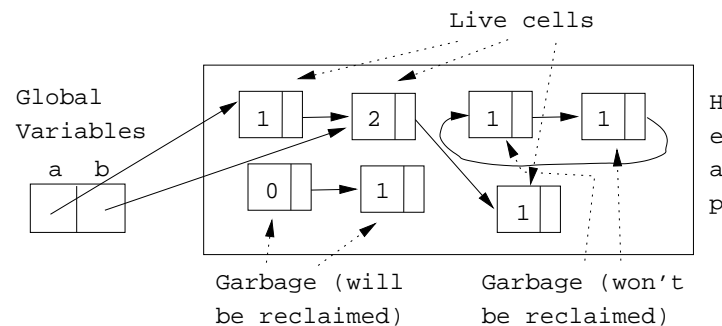
- Fragmentation – Compact the heap as a part of the GC, or only when the GC fails to return a large enough block.
- Algorithms: Reference counts, Mark/ssweep, Copying, Generational.

9 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.

10 Algorithm: Reference Counts...

- 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.



11 Algorithm: Reference Counts...

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.

12 Readings and References

- **Read Scott, pp. 395–401.**
- Apple’s Tiger book, 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.

13 Readings and References...

- 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.