Safe Manual Memory Management

David Gay[+], Rob Ennals[+] and Eric Brewer[+*]

[+]Intel Research, Berkeley

[*]University of California, Berkeley

A Buggy Program

```
struct list {
    struct list *next;
    int data;
};

void free_list(struct list *x) {
    struct list *p = x;
    do {
        free(p);
        p = p->next;
    } while (p != x);
}
```

Or is it?

```
struct list {
    struct list *next;
    int data;
};

void free_list(struct list *x) {
    struct list *p = x;
    do {
        delayed_free(p);
        p = p->next;
    } while (p != x);
    really_free_now();
}
```

Why Delay Frees? (beyond better dreams)

The two programs have an important difference:

With delayed frees, it is easier to ensure there are no live external references to the objects at deallocation time

We can use this difference as the basis for a practical, effective memory management checker - HeapSafe

HeapSafe

HeapSafe is a tool that checks the manual memory manag

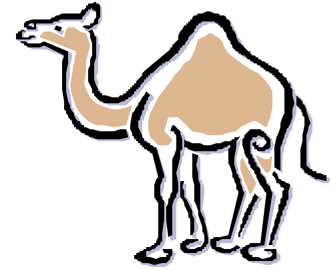
- of *single-threaded* C programs – multi-threaded support
- that use HeapSafe's delayed-free-based API
- using reference-counting to enforce runtime safety
- at reasonable cost in performance and with relatively few code changes
  - support always-on checks, rather than just debugging aids

HeapSafe:

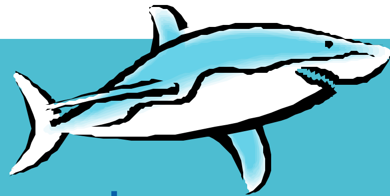
- does not enforce type safety (array bounds, etc)
  - but can be combined with a tool that does
- is not suitable for malicious code
  - practical considerations force some unsoundness

Why C?

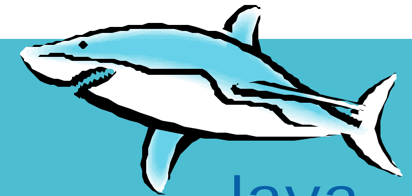
Python



OCaml



Java



Java

Why C? (aka Why Not GC?)

There is a lot of malloc/free-based C code

- programmers are often resistant to making large-scale changes

C is a systems language

- some systems need the extra control and predictability offered by manual memory management

Garbage collectors have significant memory overheads

- especially for C, for which not all GC techniques are applicable

Technical Overview

Safety condition:

a free is safe if no references remain at actual deallocation point.

To make free safes, code can

- explicitly null-out references
- or, use delayed frees

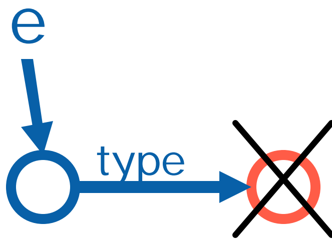
To check that no references remain

- maintain reference counts
- check reference counts at deallocation points

No References – Null-Out

```
void free_exp(exp *e)
{
    free(e->type);
    free(e);
}
```

Free fails: e->type referenced

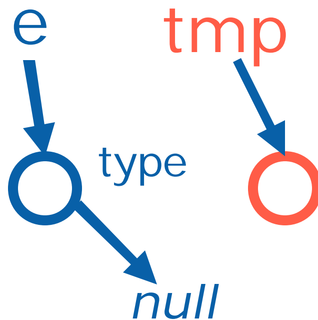


No References – Null-Out

```
void free_exp(exp *e)
{
    type *tmp = e->type;
    e->type = NULL;
    free(tmp);
    free(e);
}
```

Free fails: e->type referenced

- fix: null-out e->type first

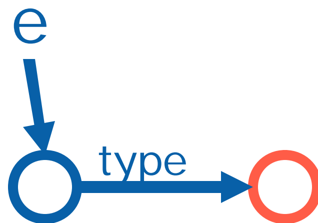


No References – Null-Out

```
void free_exp(exp *e)
{
    ZFREE(e->type);
    free(e);
}
```

Free fails: e->type referenced

- fix: null-out e->type first
- ZFREE simplifies nulling



No References – Delayed Frees

```
struct list {
    struct list *next;
    int data;
};

void free_list(struct list *x) {
    struct list *p = x;
    delayed_free_start();
    do {
        free(p);
        p = p->next;
    } while (p != x);
    delayed_free_end();
}
```

Delayed free scopes delay all frees within the scope

Scopes simplify nesting and code reuse

Deallocation points are:

- outermost delayed_free_end
- frees outside scopes

Maintaining Reference Counts

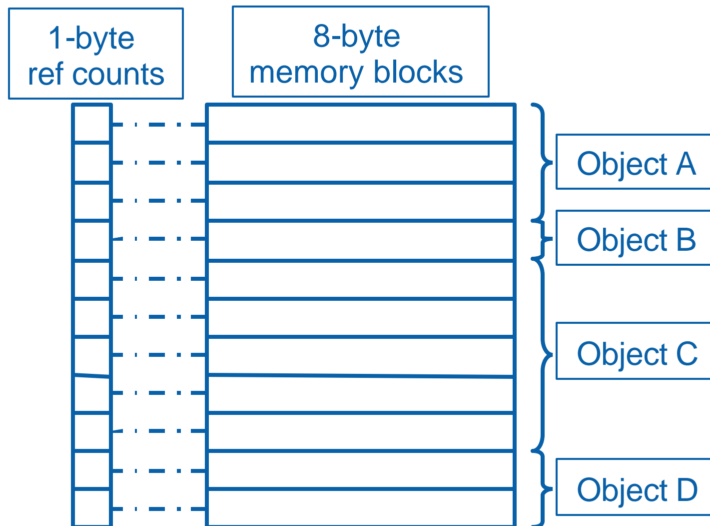
Reference counting in C is difficult, e.g.,

- internal pointers
- character-at-a-time copying (e.g., memcpy)
- pre-compiled binary code

Observations:

- reference counts only needed at relatively-rare deallocation points
- incorrect reference counts likely to be small
  - and when not small, likely to be an effectively random value

Maintaining Reference Counts



Reference count storage:

- 8-bit RC per 8-byte block
  - 12.5% space overhead
- object RC is sum of block RCs
  - false negative if RC ≡ 0 mod 256

Reference count update: *a = b becomes:

```
rcs[(int)*a >> 3]--;
*a = b;
rcs[(int)*a >> 3]++;
```

Unsoundness acceptable:

- checking system
- assuming non-malicious code
- many other sources of unsoundness remain

Checking Reference Counts

Deallocating objects o_1, o_2, ..., o_n is now straightforward:

1. decrement reference counts due to all objects o_i
2. sum 8-byte-chunk reference counts for each object o_i
3. for objects with non-zero reference counts, choose recovery strategy:
   - clear and leak o_i
   - clear, free and log o_i
   - abort the program
   - poison o_i, detect future accesses
   - ...

Reference Counting – Other Issues

The following technical issues are covered in the paper:

- need for runtime type information (pointer location)
  - in free, memcpy, generic code
- writes to local pointer variables
  - frequent, have special support to improve performance
- C's setjmp, longjmp functions
  - longjmp pops stack frames, so may need to adjust reference counts
- interoperation with pre-compiled binary code
  - may need to disable reference counting for some types

Evaluation - Implementation

HeapSafe implemented for Linux and Mac OS X

- based on CIL infrastructure, compiles to C
- malloc/free based on Doug Lea's malloc library

HeapSafe comes with a debug mode that helps fix failed frees:

- tracks the source of all pointers to object
- remembers the location of the last write
- acceptable overhead (e.g., 4x slower)

Evaluation – Benchmarks

Evaluated HeapSafe by porting ½ million lines of C code

- all malloc/free-based C programs from SPEC2000, SPEC2006
  - crafty, gzip, parser, twolf, art, equake, ammp, mesa, perl, bzip2, mcf, gobmk, hmmer, h264ref, milc, lbm, sphinx3
- five allocation-intensive programs from previous memory papers
  - cfrac, grobner, tile, espresso, boxed-sim
- applied to bootable Linux kernel in separate work [HotOS 07]

Porting

- straightforward (minutes to a day or so) for all programs except perl
- on average, 0.6% of lines changed
- perl took three weeks:
  - understanding perl's complex memory management
  - tracking down, fixing bugs (6) that caused frees to fail
- more details in paper, including typical porting process

Evaluation – Space Usage

Space Overhead
100.00%
75.00%
50.00%
25.00%
0.00%
crafty
parser
art
ammp
perl
mcf
hmmer
milc
sphinx3
grobner
espresso
mean
HeapSafe
GC
GC out of memory

Evaluation – Runtime Overhead

Runtime Overhead

	HeapSafe	GC
90.00%		
80.00%		
70.00%		
60.00%		
50.00%		
40.00%		
30.00%		
20.00%		
10.00%		
0.00%		
-10.00%		

crafty
parser
art
ammp
perl
mcf
hmmer
milc
sphinx3
grobner
espresso
mean

parser: 20% for RC, perl: 70% for RC

Related Work (Explicit Memory Management)

Debugging allocators:

- purify, valgrind, dmalloc
- very high overheads – not practical for full-time use

Regions:

- Explicit regions: RC [Gay and Aiken 01], Cyclone [Morrisett et al]
  - good performance, but hard to apply to some programs
- Type-homogeneous regions (inferred) [Dhurjati et al 05]:
  - guarantee type safety, but not deallocation correctness

Earlier systems for checking deallocation

- lock-and-key [Fischer, Leblanc 80]
- tombstones [Lomet 85]

Conclusion

Delayed frees make checking manual memory management practical

HeapSafe is an implementation of delayed frees for C that:

- has low space overhead (13%)
- mostly reasonable time overhead (11%, but 84% worst case)
- shown to work on large programs with complex allocation

Future work

- merge with Deputy, a type-safety system for C
- improve reference counting, provide concurrency support
- reference counting makes “cast to linear” practical

Conclusion

Delayed frees make checking manual memory management practical

http://memory.intel-research.net