7 Data Types

7.7.2 Dangling References

Memory access errors—dangling references, memory leaks, out-of-bounds access to arrays—are among the most serious program bugs, and among the most difficult to find. Testing and debugging techniques for memory errors vary in when they are performed, how much they cost, and how conservative they are. Several commercial and open-source tools employ binary instrumentation (Section 15.2.3) to track the allocation status of every block in memory and to check every load or store to make sure it refers to an allocated block. These tools have proven to be highly effective, but they can slow a program several-fold, and may generate *false positives*—indications of error in programs that, while arguably poorly written, are technically correct. Many compilers can also be instructed to generate dynamic semantic checks for certain kinds of memory errors. Such checks must generally be fast (much less than 2× slowdown), and must never generate false positives. In this section we consider two candidate implementations of checks for dangling references.

Tombstones

EXAMPLE 7.134
Dangling reference detection with tombstones

Tombstones [Lom75, Lom85] allow a language implementation can catch all dangling references, to objects in both the stack and the heap. The idea is simple: rather than have a pointer refer to an object directly, we introduce an extra level of indirection (Figure 7.17). When an object is allocated in the heap (or when a pointer is created to an object in the stack), the language run-time system allocates a tombstone. The pointer contains the address of the tombstone; the tombstone contains the address of the object. When the object is reclaimed, the tombstone is modified to contain a value (typically zero) that cannot be a valid address. To avoid special cases in the generated code, tombstones are also created for pointers to static objects. ■

For heap objects, it is easy to invalidate a tombstone when the program calls the deallocation operation. For stack objects, the language implementation must

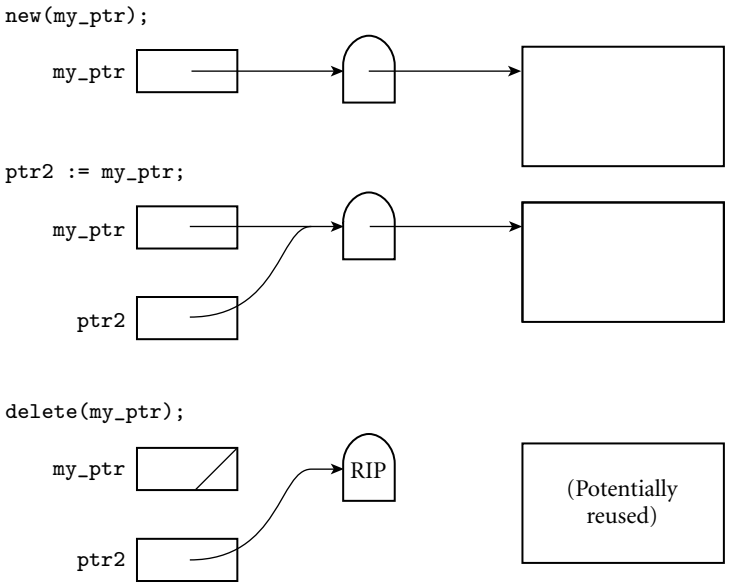

Figure 7.17 **Tombstones.** A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

be able to find all tombstones associated with objects in the current stack frame when returning from a subroutine. One possible solution is to link all stack-object tombstones together in a list, sorted by the address of the stack frame in which the object lies. When a pointer is created to a local object, the tombstone can simply be added to the beginning of the list. When a pointer is created to a parameter, the run-time system must scan down the list and insert in the middle, to keep it sorted. When a subroutine returns, the epilogue portion of the calling sequence invalidates the tombstones at the head of the list, and removes them from the list.

Tombstones may be allocated from the heap itself or, more commonly, from a separate pool. The latter option avoids fragmentation problems, and makes allocation relatively fast, since the first tombstone on the free list is always the right size.

Tombstones can be expensive, both in time and in space. The time overhead includes (1) creation of tombstones when allocating heap objects or using a "pointer to" operator, (2) checking for validity on every access, and (3) double-indirection. Fortunately, checking for validity can be made essentially free on most machines by arranging for the address in an "invalid" tombstone to lie outside the program's address space. Any attempt to use such an address will result in a hardware interrupt, which the operating system can reflect up into the language run-time system. We can also use our invalid address, in the pointer itself, to represent the constant `nil`. If the compiler arranges to set every pointer to `nil` at elaboration time, then the hardware will catch any use of an uninitialized pointer. (This technique works without tombstones, as well.)

The space overhead for tombstones can be significant. The simplest approach is never to reclaim them. Since a tombstone is usually significantly smaller than the object to which it refers, a program will waste less space by leaving a tombstone around forever than it would waste by never reclaiming the associated object. Even so, any long-running program that continually creates and reclaims objects will eventually run out of space for tombstones. A potential solution, which we will consider in Section 7.7.3, is to augment every tombstone with a *reference count*, and reclaim tombstones themselves when the reference count goes to zero.

Tombstones have a valuable side effect. Because of double-indirection, it is easy to change the location of an object in the heap. The run-time system need not locate every pointer that refers to the object; all that is required is to change the address in the tombstone. The principal reason to change heap locations is for *storage compaction*, in which all dynamically allocated blocks are "scooted together" at one end of the heap in order to eliminate external fragmentation. Tombstones are not widely used in language implementations, but the Macintosh operating system (versions 9 and below) used them internally, for references to system objects such as file and window descriptors.

Locks and Keys

Locks and *keys* [FL80] are an alternative to tombstones. Their disadvantages are that they work only for objects in the heap, and they provide only probabilistic protection from dangling pointers. Their advantage is that they avoid the need to keep tombstones around forever (or to figure out when to reclaim them). Again the idea is simple: Every pointer is a tuple consisting of an address and a key. Every object in the heap begins with a lock. A pointer to an object in the heap is valid only if the key in the pointer matches the lock in the object (Figure 7.18). When the run-time system allocates a new heap object, it generates a new key value. These can be as simple as serial numbers, but should avoid "common" values such as zero and one. When an object is reclaimed, its lock is changed to some arbitrary value (e.g., zero) so that the keys in any remaining pointers will not match. If the block is subsequently reused for another purpose, we expect it to be very unlikely that the location that used to contain the lock will be restored to its former value by coincidence.

EXAMPLE 7.135

Dangling reference detection with locks and keys

Like tombstones, locks and keys incur significant overhead. They add an extra word of storage to every pointer and to every block in the heap. They increase the cost of copying one pointer into another. Most significantly, they incur the cost of comparing locks and keys on every access (or every provably nonredundant access). It is unclear whether the lock and key check is cheaper or more expensive than the tombstone check. A tombstone check may result in two cache misses (one for the tombstone and one for the object); a lock and key check is unlikely to cause more than one. On the other hand, the lock and key check requires a significantly longer instruction sequence on most machines.

To minimize time and space overhead, most compilers do not by default generate code to check for dangling references. Most Pascal compilers allow the programmer to request dynamic checks, which are usually implemented with locks and keys. In most implementations of C, even optional checks are unavailable.

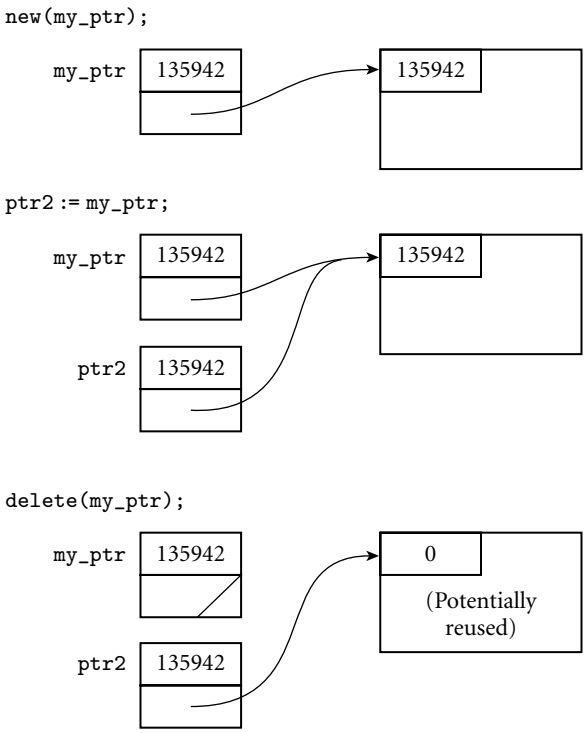

Figure 7.18 **Locks and keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

CHECK YOUR UNDERSTANDING

74. What are *tombstones*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
75. Explain how tombstones can be used to support *compaction*.
76. What are *locks* and *keys*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
77. Explain why the protection afforded by locks and keys is only probabilistic.
78. Discuss the comparative advantages of tombstones and locks and keys as a means of catching dangling references.