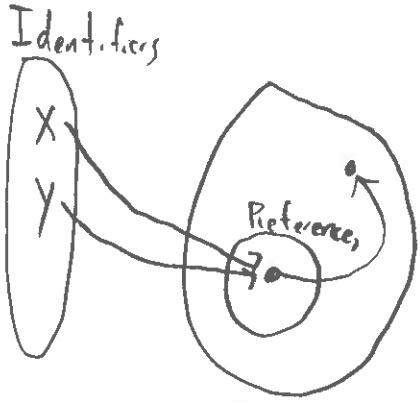
Types 2

Pointers

Aliasing

Aliasing occurs when a variable can be accessed via two or more identifiers.

E.g.



Pointers and references obviously introduce aliasing.

Other sources:

- pass by reference

Aliasing makes programs harder to read about.

Wild pointers (dangling pointers)

Pointers/references to deallocated memory.

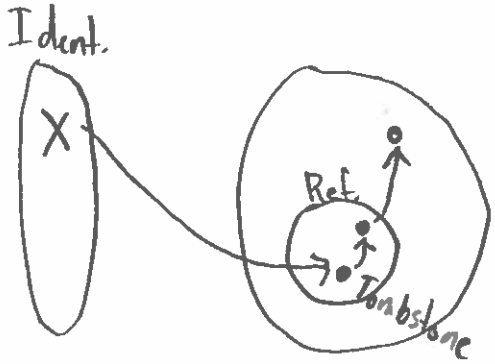
Ways to avoid:

- Disable/disallow manual memory allocation
- Ignore it (programmer's responsibility)
- Tombstones / lock-and-key.

Tombstones (deprecated)

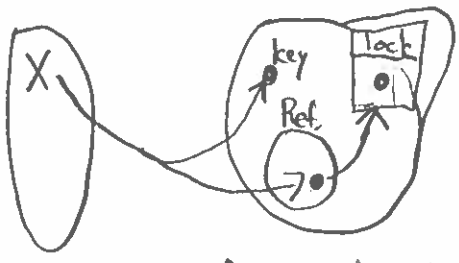
Instead of identifiers being mapped to a reference to memory, map them to a tombstone which then is mapped to a reference to memory.

When deallocation occurs, remove the reference leaving the tombstone.



Language designers decided: not worth the cost

Lock and key



Instead of just being mapped to a reference, identifiers are mapped to a reference/key pair.

When dereferencing, compare the key value to a lock value attached to the memory.

Arrays

Concrete instance of a sequence type.
- Compared to the abstract notion of sequence
- Other sequences: (linked) lists,

Contiguous collection of memory cells.

5 kinds of arrays

Based on when memory allocation is done and array length is set.

- Static arrays
  - memory allocation is static (load memory)
  - array length is statically set.
- Fixed-stack dynamic
  - allocation is dynamic, on the stack
  - array length is statically bound.
    - e.g. `int xs[10];`
- Stack dynamic
  - allocation is dynamic, on the stack
  - array length is dynamic, but remains constant after allocation.
    - e.g. `int xs[n];`

- Fixed heap-dynamic
  - allocation is dynamic, on the heap
  - length is dynamically bound, but remains constant after declaration.
- Heap-dynamic
  - ...
  - length is dynamically bound, and can change after allocation.
  - costly!

Array Lists

An ArrayList is a heap-dynamic array.
In order to mitigate the cost of reallocating memory, it is usually done as so:
when the array needs to grow longer than the currently allocated memory, double the allocated memory.

- Never using more than twice the required memory.
- Reallocation becomes infrequent as the array grows.

Abstract datatypes

Data types packaged with operations on that type.

- To be abstract, should practice information hiding.
  - Hiding or obscuring implementation details.
  - Or simply restricting the programmers ability to use knowledge of implementation details.

Eg. stacks are an abstract datatype;
can be implemented as any sequence type together with push and pop operations.

Consider: you've implemented stacks using arrays.
To access the top of the stack,
you provide a reference to the "top" of the stack.

Examples of abstract datatype support:

- Classes
- Modules, package.
- Namespaces

Conditions to be an abstract datatype:

- representation of elements of the type is hidden from users of the type
- declaration of the type and its operations are contained in a syntactic unit.

Algebraic datatypes

An algebraic / inductive datatype is a possibly recursive sum of product types.

E.g. `datatype List A = [] | A :: List A`

`datatype Maybe A = Just A | nothing`

`datatype Either A B = Left A | Right B`

`datatype Product A B = Pair A B`

Type coercions

How "reasonable" are implicit typecasts?
(or, for that matter, explicit type casts).

Depends upon the types involved.

int → float

seems more "reasonable" than

float → int

Because, (at least hopefully) int ⊂ float.

If for two types A and B, we have A ⊂ B, then we call a cast from A to B a widening conversion. A cast from B to A is a narrowing conversio