Describing the Store

- Variable declarations typically implicitly allocate storage.

- In most languages, there are other ways to allocate storage too, such as explicit new operations or implicit boxing operations.

- Simplistic store model: mutable map from locations to values.

- Better models require distinguishing different classes of storage based on the lifetime of the data.
Storage Lifetimes

- Typical computations use far more memory locations in total than they use at any one point.

- So most language implementations support re-use of memory locations that are no longer needed.

- The lifetime of every object should cover all moments when the object is being used.

- Otherwise, we get a memory safety bug.
Storage Classes: Static Data

- Lifetime = **Entire Execution**
- Typically used for **global** variables and constants
  - If language has no recursion, can also be used for function-local variables
- **Fixed** address known before program executes
- **No** runtime allocation/deallocation costs
Storage Classes: Stack Data

- **Nested** Lifetimes (last allocated is first deallocated)

- Typically used for function-internal variables (and internal control data for function calls)
  - Works because function call lifetimes also nest

- Allocation/deallocation are very cheap (just adjust the stack pointer)

- Produces good locality for caches, virtual memory
Storage Classes: Heap Data

- **Arbitrary** Lifetimes

- Typically used for *explicitly allocated* objects

  - Some languages implicitly heap-allocate other data structures, e.g. bignums, closures, etc.

- Allocation/deallocation are relatively **expensive**

  - Run-time library must decide where to allocate

  - Deallocation can be done manually (risking memory bugs) or by a **garbage collector**
Scope, Lifetime, Memory Safety

- **Lifetime** and **scope** are closely connected.

- For a language to be **memory safe**, it suffices to make sure that in-scope identifiers never point (directly or indirectly) to deallocated objects.

- For **stack-allocated local** variables, this happens naturally:
  - Stack locations are deallocated only when function returns and its local variables go out of scope forever.

- For **heap** data, easiest to enforce safety using a garbage collector (GC):
  - GC typically works by recursively **tracing** all objects reachable from names that are currently in scope (or that might come back into scope later).
  - Only **unreachable** objects are deallocated, making their locations available for future re-allocation.
Explicit Deallocation

- Many older languages (notably C/C++) support **explicit** deallocation of heap objects
- Somewhat more efficient than GC
- But makes language unsafe: “**dangling pointer**” bug occurs if we deallocate an object that is still in use [unchecked runtime error]
- Converse problem: “**space leak**” bug occurs if we don’t deallocate an unneeded object.
  - Not a safety problem, but may unnecessarily make program run slower or crash with “out of memory” error
- Rust language supports safe explicit deallocation.
Pragmatics of Large Values

- Real machines are very efficient at handling word-size chunks of data (e.g. 16-64 bits depending on hardware). Things that fit easily in a word:
  - Numbers, characters, booleans, enumerations, class tags, etc.
  - Memory addresses (locations)
- Words are very easy to move, load, store, supply to operations, etc.
- But how can we manipulate larger chunks of data, such as records or arrays, which may occupy many words?
Boxing

Two basic ways to represent large values

- The *unboxed* representation holds the actual bits of the value, using as many machine words as necessary.

  ![Unboxed representation](image)

  ~textbook: “value” model

- The *boxed* representation allocates separate storage (the “box”) for the actual bits, and then represents the value by the location of that storage.

  ![Boxed representation](image)

  ~textbook: “reference” model

- Boxes are usually, but not necessarily, stored in the heap.

- Boxing may be performed implicitly or explicitly.
Boxed vs. Unboxed

Example: an array of 100 (machine) integers

- Unboxed implementation: values occupy 100 consecutive words
- Boxed representation: values occupy 1 word pointer pointing to 100 consecutive words contents

Choice of representation can make a big difference to semantics on operations on the data

- What does assignment mean?
- How does parameter passing work?
- What do equality comparisons mean?
Unboxed Assignment Semantics

Early languages often used unboxed records and arrays. For example, in Pascal and related languages:

```pascal
TYPE Employee = RECORD
  name : ARRAY (1..80) OF CHAR;
  age : INTEGER;
END;
```

specifies an unboxed representation, in which value of type `Employee` will occupy 84 bytes (assuming 1 byte characters, 4 byte integers).

The semantics of assignment is to copy the entire representation. Hence:

```pascal
VAR e1,e2 : Employee;
e1.age := 91;
e2 := e1;
e1.age := 19;
WRITE(e1.age, e2.age);
```

prints 19,91.
### Step-by-step

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>fred</td>
</tr>
<tr>
<td>e2</td>
<td>alice</td>
</tr>
</tbody>
</table>

**e1.age := 91**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>fred</td>
</tr>
<tr>
<td>e2</td>
<td>alice</td>
</tr>
</tbody>
</table>

**e2 := e1**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>fred</td>
</tr>
<tr>
<td>e2</td>
<td>fred</td>
</tr>
</tbody>
</table>

**e1.age := 19**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>fred</td>
</tr>
<tr>
<td>e2</td>
<td>fred</td>
</tr>
</tbody>
</table>
Unboxed representation issues

- This assignment semantics seems simple and appealing, but it has problems:
  - Assignment of a large value is expensive, since lots of words may need to be copied.
  - Especially hard to generate efficient code if size of large value is not known statically.
Boxed Assignment Semantics

Most modern languages (e.g. Java, Python, Haskell) box all values (e.g. objects, records, constructions) that are larger than one word.

These languages naturally use reference semantics for assignment: just the pointer is copied, creating an alias.

case class emp(var name:String, var Age:Int)
val e1 = emp("fred",91)
val e2 = e1
e1.age = 19
println(e1.age + " " + e2.age)

prints 19,19
Step-by-step

1. $e_1$ → fred

2. $e_2 = e_1$

3. $e_1$ → fred

4. $e_2$

5. $e_1$.age = 19

6. $e_1$ → fred

7. $e_2$
Explicit Pointers

Languages that use unboxed semantics may also have explicit pointer types to support reference-style operations.

```c++
struct Emp {
    char name[80];
    int age;
};
Emp *e1 = new Emp();
e1->age = 91;
Emp *e2 = e1;
Emp e3 = *e1;
e1->age = 19;
cout << e1->age << " " << e2->age << " " << e3.age << "\n";
```

In C/C++, `struct` and `class` instances are fundamentally unboxed, but programmers usually box them explicitly (using `new` or `malloc`) and manipulate them via pointers.

prints 19, 19, 91
Varieties of Equality

Languages typically provide some form of built-in equality testing on values. When are two (large) values equal?

Under **structural** equality, values are equal when their contents are equal, bit for bit. (Only sane definition for unboxed values.)

Under **reference** equality, values are equal when their locations are identical.

Reference equality ⇒ structural equality, but not vice-versa

Reference equality may be cheaper to check than structural equality
Multiple kinds of equality

- Some languages provide both structural and reference equality, under different names.

- They may also provide a standard way for programmers to define equality for a given type in an ad-hoc way.

E.g. in Scala:

- The `eq` operator gives reference equality.
- The `==` operator invokes a user-defined `equals` method.
- For case classes `equals` is pre-defined to be structural equality.
Pairs

- To study the essence of heap data structures, we can focus on a single new kind of value, the pair

- Like a record with two fields, each containing another value

- Written using “infix dot” notation

- We can build larger records of a fixed size just by nesting pairs

\[(1 \cdot ((2 \cdot 3) \cdot 4))\] corresponds to

1

2 3

4
We can also build all kinds of interesting arbitrary-sized recursive structures using pairs.

For example, to represent (singly-linked) lists we can use a pair for each node in the list. First field contains an element; second field points to the next link, or is 0 to indicate end-of-list.

Example: 1, 2, 3  (1. (2. (3. 0)))

Note that for programs to detect end-of-list, we need a test that distinguishes integers from pairs.