Composite types
Ch 8 Scott

The line between “built-in and composite types is thin
Is a string built in?
    Not (quite) in C
What defines a composite type?

Record / structs
    Go - struct
    Rust struct
    Java class

    Why have records?
    Implications of reference model vs value model on records

    Is Go anonymous include equivalent to inheritance in Java??
    What is stored in a go struct?? Overhead??
        see size_go/structsize.go
        see size_rust/scr/main.rs
    What is stored in a java class?? Overhead?? How do you even know what the overhead of a java class instance is?
        short answer — you do not .. but This was on stackOverflow

In a modern 64-bit JDK, an object has a 12-byte header, padded to a multiple of 8 bytes, so the minimum object size is 16 bytes. For 32-bit JVMs, the overhead is 8 bytes, padded to a multiple of 4 bytes. *(From Dmitry Spikhalskiy's answer, Jayen's answer, and JavaWorld.)*

Typically, references are 4 bytes on 32bit platforms or on 64bit platforms up to `-Xmx32G`; and 8 bytes above 32Gb (`-Xmx32G`). *(See compressed object references.)*

As a result, a 64-bit JVM would typically require 30-50% more heap space. *(Should I use a 32- or a 64-bit JVM?, 2012, JDK 1.7)*

Boxed types, arrays, and strings

Boxed wrappers have overhead compared to primitive types (from JavaWorld):

- **`Integer`**: The 16-byte result is a little worse than I expected because an `int` value can fit into just 4 extra bytes. Using an `Integer` costs me a 300 percent memory overhead compared to when I can store the value as a primitive type

- **`Long`**: 16 bytes also: Clearly, actual object size on the heap is subject to low-level memory alignment done by a particular JVM implementation for a particular CPU type. It looks like a `Long` is 8 bytes of Object overhead, plus 8 bytes more for the actual long value. In contrast, `Integer` had an unused 4-byte hole, most likely because the JVM I use forces object alignment on an 8-byte word boundary.

Other containers are costly too:

- **Multidimensional arrays**: it offers another surprise.
  Developers commonly employ constructs like `int[dim1][dim2]` in numerical and scientific computing.
  In an `int[dim1][dim2]` array instance, every nested `int[dim2]` array is an `Object` in its own right. Each adds the usual 16-byte array overhead. When I don't need a triangular or ragged array, that represents pure overhead. The impact grows when array dimensions greatly differ.
  For example, a `int[128][2]` instance takes 3,600 bytes. Compared to the 1,040 bytes an `int[256]` instance uses (which has the same capacity), 3,600 bytes represent a 246 percent overhead. In the extreme case of `byte[256][1]`, the overhead factor is almost 19! Compare that to the C/C++ situation in which the same syntax does not add any storage overhead.

copy and Equality
    a==b
    what is difference in Go and Rust and Java?
        again value-model vs reference model language
        see **equal_go/equal.go**

in particular, for go show the addresses of objects in equal_go

Arrays

usually homogenous type

Why homogenous????

value-model language it is kind of required

Go array vs Slice what is stored where

Exactly What is stored in an array in Java

Java since everything inherits for Object can make non-homo array

easy in reference model language

easy with subtype polymorphism

Note that similar game is harder in value model Go

usually contiguous in memory

Go/Rust — arrays MUST be sized at compile time!! (Why?)

arrays contain the objects, literally. So each spot in otherwise "empty" array actually contains the sting with zero value(s).

Go/Rust — slices contain REFERENCES!!! Why? So what?

consider difference between

a := b for array and slice in Go

for array, everything is new! Copying can be expensive

for slice, the address of the slice is new (value model)

but all the content is the SAME (copy the references)

WHY?

Heap allocation vs stack allocation!!!

Row-Major & Column major ordering

assumes array contained in contiguous block of memory

Looking at pointer addresses in Go you can see this.

Suppose A is 7x10 array

R-M

A[2,4] followed by A[2,5] … a[2,6],a[3,0]

C-M

a[2,4], a[3,4] … a[9,4],a[0,5]

Why do I care?

Max performance says always access memory locations near each other

so nested for loop for R-M

```
for i 0..6
    for j 0..9
        a[i][j]
```

For C-M

```
for j 0..9
    for i 0..6
        a[i][j]
```

Easy to build multi-d array in RM so almost all languages use Column-major

see size_go/rowmajor.go

see rowmajor_rust/src/main.rs

Does java use row-major or column major??? Probably neither but since you cannot really see where thigns are stored, you cannot tell. See rowmajor_rust for what Java likely does

Composite equality checks
Go == on structs compares the stuff inside — a deep check. (again, kind of natural in value model)
Go defines == over array and does a deep check!!!
no == over slices!!! Why? (slices could contain themselves, Why is this a problem?)

Associative arrays (maps), sparse arrays, …
are these really arrays? Or are they something else that just uses the same syntax?

========== stop roughly here on 11/14 ==========

Strings:
are they a primitive type in the language
C, Rust — definitely not
Java, Go— might as well be.
J,G — String is a fixed entity. A length change (append) makes new string
Java StringBuffer, StringBuilder
Go: “A string is an immutable sequence of bytes”
Why are strings immutable????
String Pool
a place to store string literals
String pool — I imagine as a hashtable<String, String>
java “intern”
see string_intern/Interner.java
does go have string interning?
yes for strings known at compile time
no otherwise (no privided intern method)
see string_intern/intern.go
In big apps string pool can save lots of space
for instance, a collection of books by Scott and Gibon has 2.6M words … but only 70000 unique
Security
anti hacking. Mutable strings could let hackers attack. For instance, person passes a string — we validate — in background they change ….
Thread Safety
immutable strings are thread safe
Note that all of these arguments in favor of immutable strings can be generalized to immutable everything!

Recursive types
E.g. Linked lists
How to Handle in Value-model langs like Go.
Answer Pointers!!!
see **pointer_go —** already discussed so this code is review
see **tree_go** — lots of points to make

Linked list in Rust
and the null safe problem
see ll_rust/src/main.rs

new operator in Go / Java allocates from heap.
stack allocation auto reclaimed when frame complete (closures aside), but heap is forever!

Garbage collection
Reference Counting
when the number of references goes to zero, reclaim
problem — circular structures
problem, how to count
fragmentation of memory
Mark-and-sweep
1. mark everything as useless
2. start with all non-heap pointers and recursively follow. Mark everything touches as good
3.Go through heap and destroy everything not marked as good
Stop and Copy
split memory in half
Rather than mark and sweep, in step 2, copy from current to new. Then delete anything not copied. Next time, switch current and new

Lists, etc
difference between list and array?
pointer following?
typically not indexed (why not??)
Go: no list type?
as a package, but NOT a language primitive
Homogeneous vs heterogenous
Opinion: lists are associated with functional programming because they are one with LISP.

Counter argument. A: Lists can be built recursively by appending to the front. In so doing you can add items to list without changing the list as it was previously seen. Lists built in such a way are therefore perfect fit for functional programming.
B: Linked lists are amenable to immutability — indeed immutability makes sharing of linked list parts a practical thing
For beginning of an implementation

Go: tree_go (a tree rather than a linked list

Subsections of arrays/lists
go slice[start:end] returns that part of slice between start and end
Java: neither arrays nor ArrayList have subsections built in.