Mojo Archive Format

GOALS

- Fast
- Simple
- Static bindings
- Structured data and arrays
- Type information is not encoded
- Inter-process, not inter-device

DETAILS

The Mojo Archive Format describes the serialization of structured data (structs and arrays), starting with a root struct or array. The overall size of the serialized archive is not encoded in the archive.

Primitives:

- bool: packed down to a bit field (least significant bit first)
- {u}int{8,16,32,64}: little-endian
- float, double: little-endian IEEE-754 format.

Structs:

- Structs are laid out in memory, so that they may be dereferenced directly as C structs.
- Structs are prefixed with a header that 2 the size (uint32) in bytes of the struct and the number (uint32) of encoded fields. This second number serves to version the struct. (See the VERSIONING section below.) The size of the struct may include some padding bytes.
- If a struct has a nested struct member, then that member is expressed as a pointer.
- If a struct has a nested array member, then that member is expressed as a pointer.

Arrays:

- Arrays are prefixed with the size (uint32) in bytes of the array and the number (uint32) of elements in the array. The second number is important since the number of bytes consumed by the array may include some padding at the end, or because bools may be packed tightly as single bits.
- Strings are a specialization of an array of bytes that is known to be UTF-8 encoded. There is no extra null termination byte for strings. Embedded nulls are valid in strings, and by leaving off a null termination byte, we help dissuade people from addressing these strings with a simple `const char*` variable. (A smarter string class like

base::StringPiece is to be encouraged at the C++ bindings layer.)

Alignment:

- Members of structs should be aligned to multiples of their size within a struct or array.
- Structs and arrays should be 8-byte aligned.
- Furthermore the size of structs as specified in the size field of the struct header must be a multiple of 8 bytes so structs sometimes include padding at the end. Note that this requirement does not hold for arrays and so if padding bytes are needed after an array then the count of these is not included in the size field of the array header.

Pointers:

- Pointers are encoded as byte offsets from the location where the pointer is stored to the location of the data being pointed to. The address of what is pointed to can be computed via `reinterpret_cast<char*>(&pointer) + *pointer`.
- Pointers are 64-bit values (so they can be "fixed-up" to hold real memory addresses after a message has been copied off the wire).
- Pointers can be `null`. A `null` pointer is encoded by an offset of `0`.
- Offsets must not be negative, which implies that pointers cannot form loops.
- Pointers should only point to the beginning of structs and arrays.
- For any array or struct, there shouldn't be multiple pointers pointing to it.

Handles:

- Mojo Handles are encoded as 32-bit integers. They are offsets into a vector of handles associated with the archive, which is not part of the archive. The space allotted for a Handle offset is large enough to hold a Handle value (so they can be "fixed-up" to hold real Handle values after a message has been copied off the wire). Invalid handles are encoded as `-1` (`max uint32`).
- If handle `X` is prior to handle `Y` in the depth-first traversal order (defined in the LAYOUT section), and both of them are valid, then the value of `X` should be smaller than that of `Y`. It implies that all valid handles in an archive should have different values.

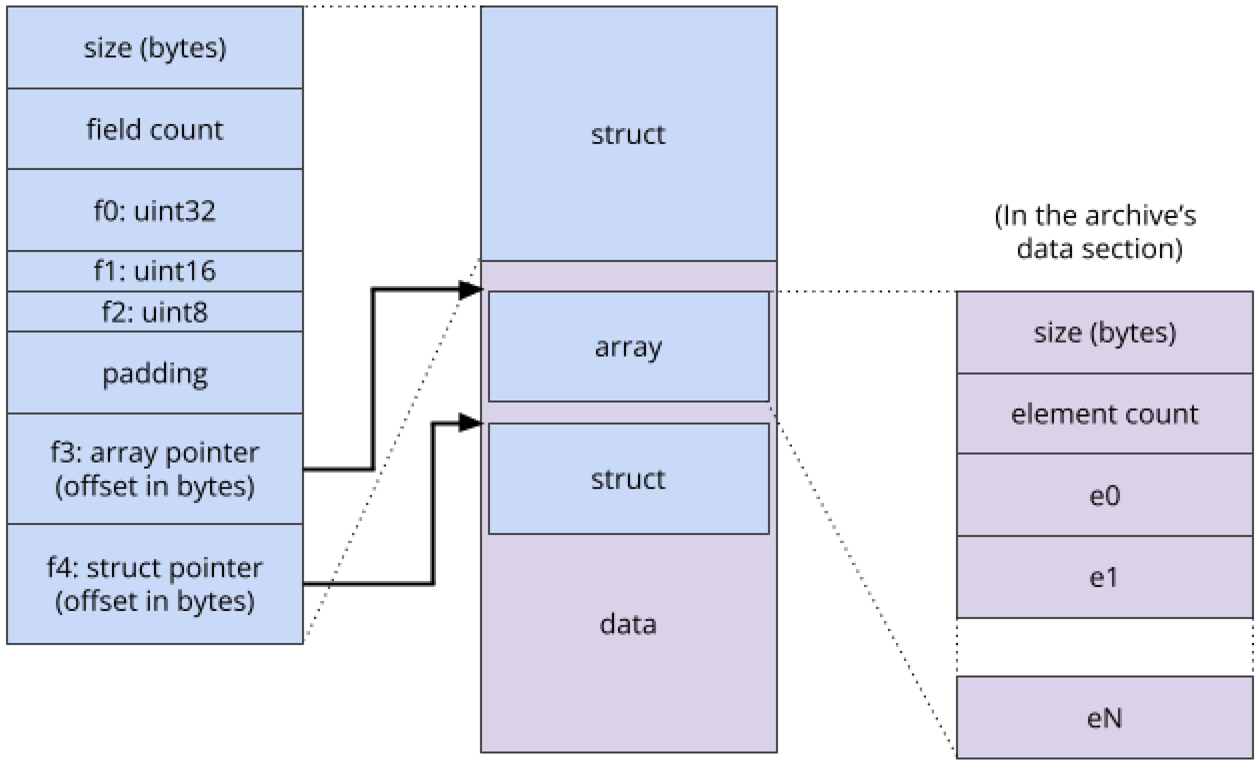
LAYOUT

The depth-first traversal of the object tree associated with an archive is determined by running the following algorithm with the root object:

```
traverse(object) {
  visit_order = [object]
  if (object is struct) {
    children = object.fields_in_ordinal_order
  } else {  // object is array
    children = object.elements
  }
  for (child in children) {
    if (child is pointer) {
      if (child is not null)
        visit_order.append(traverse(dereference(child)))
    } else {
      visit_order.append(child)
    }
  }
  return visit_order
}
```

The layout order of structs and arrays in an archive should be the same as the order in which they are visited by the depth-first traversal.

Structs and arrays are not allowed to overlap each other.



VERSIONING

This system is designed to allow sharing archives between two programs that are using slightly different versions of the same structs. The effective version of a struct is given by the number of fields encoded in the struct, and all fields up to that number must be encoded in a pre-determined order. (See the FIELD PACKING section below.)

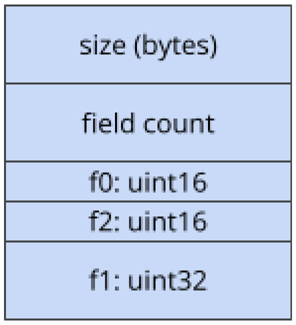
Each field is assigned a numeric name, referred to as its ordinal value. Ordinal values are stable and are 0-based, assigned in increasing order without gaps.

If a program supports a struct Foo encoded with N fields, then when it reads a Foo with N+1 fields, it must ignore the additional field. If the same program reads a Foo encoded with only N-1 fields, then it should assume a default value for the missing field. The default value may be specified in a description of the struct. Otherwise, a default value of zero'd out memory would be assumed.

FIELD PACKING

The fields of a struct are packed in ordinal order at the first place they fit, respecting general alignment/padding requirements.

For example, given a structure with three fields: f0 of type uint16, f1 of type uint32 and f2 of type uint16, the expected layout of the struct would be:


size (bytes)
field count
f0: uint16
f2: uint16
f1: uint32