Axiomatic Pointer Memory

Daniel Nikpayuk

March 4, 2018

This article is licensed under
Creative Commons Attribution-NonCommercial 4.0 International.

Constructive Narrative

If you look at the history of C and C++, the *construct of pointer memory* was developed and designed in an organic, reactive, and evolutionary way. Although this may have been acceptable at the time, higher demands for more complex software fueled by a technology driven economy in a globally connected world has made it insufficient by today's standards. As such, the builtin narrative design of memory as a *type system* within C and C++ is ad-hoc and haphazard. The goal of this essay [1] is to offer an alternate type system which provides this missing clean, axiomatic, narrative design.

There's an additional useful way to look at this: If we took the semantic construct of pointer memory, broke it down into its signifying components, then used those to generate an axiomatic type system (the smallest) containing it, what would that look like? [2]

As a starting point in this direction, I should first introduce my motivating critique of the current type system:

1. The integer type system within C is interleaved with the general pointer system. The integer type system is a specialization, and should be built on top of lower level pointer constructs.

2. There are no mitigating narratives within C++ to build memory forms from previous memory forms. If such a hierarchy of memory had existed, the STL library could have been more effectively implemented by shifting the weight to these low level memory patterns—allowing us to reuse much more low level memory to build higher level data structures.

The idea here then is to forget everything we know about pointer memory, and start over.

A natural first step is to break down our everyday pointer memory. To that end, pointer memory first of all consists of a *memory location*. It also has a *unique numerical address*, and an *internal state*—for which its content is also a block of numerals. Although such numeral blocks may be interpreted in many ways, when we restrict ourselves to pointers, we narrow our interpretation to say these internal states refer back to the numerical addresses of other locations. [3] To summarize this then, the core concepts and symbols of interest to us are: **name, location, internal state**. In terms of the relationships between these signs: **Internal states are mutable, and are identified as names**.

[1] I should mention that I intend to hold the level of mathematical rigour usually presented in an article, but I have otherwise classified this text as an essay. I have done so for the reason that the content, although rigorous, is a non-standard narrative in the literature of programming and/or computing science.

[2] An advantage to taking such an approach, is when it came time to implement, subroutine optimizations would already correspond to direct translations of processor circuit subroutines. Hardware algorithms are several orders magnitude faster, and so represent ideal optimizations.

[3] In C/C++, numerical addresses are referred to as pointer types, while their internal states can be accessed by the unary dereference ($*$) operator.

Addresses

Names show up both as "names" as well as instances of internal states. Let's start here:

Address: Our primitive name set consists of all finite length binary strings [4]

$$\{0, 1, 00, 01, 10, 11, 000, 001, 010, \ldots\}$$

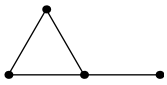
for which we will call *addresses*.

The underlying idea in shaping a constructive narrative which will allow for a clean design of pointer memory is to stratify these addresses into levels of complexity, each one building on the previous. Then, for each level of complexity, we create an interface model for which we can navigate and mutate the composite memory from the simpler variants for which it is composed.

Graph Theory

We will use graph theory as the language of choice to express our interface models, as graphs not only bind address names together, they also embed navigational content within. This is relevant to C and C++ as *iterators* are the medium of exchange when navigating memory from a given *starting* location. Furthermore, if we can navigate memory constructed from simpler memory, we can also mutate it accordingly.

As a quick review, the definition of a *graph* $G = (V, E)$ is a pair of sets, where V is called the *vertex set*, its elements being *vertices*, and E is called the *edge set*, its elements being *edges*. The relationship between the two is that the edge set consists of pairs of vertices $E \subseteq V \times V$. Corresponding to each graph G is a visual representation of *dots* (vertices) and *lines* (edges):



Beyond this, our use of graph theory will be left at an intuitive level.

Bits

We start with bits as our axiomatic level of our address set. Before we define bit memory though, we need a couple of prerequisite definitions:

Interval of access: An *interval of access* of a given memory location is a sequence of individual access operations—either to read or write.

Stable State Memory: A memory location is called *stable* if in the interval of its access, there are no mutations to its inner state.

Now we can introduce our bit memory:

Bit Memory: *Bit memory* consists of any stable memory location which holds the addresses $0, 1$.

Notice how we make no mention of the names of these memory locations? They can be any variety of name. This also means there's no need to figure out how to navigate such names just yet.

As for mutations, they are actually orthogonal to the complexity mitigating axiomatic narrative we're developing here. The reason being is that a mutation can be thought of as a function from an input internal state to an output internal state at the same location. In this case, the exploration and categorization of the mutation operators is independent of our exploration of pointer memory itself. Furthermore, as it suffers from its own categorical complexity, the theory of mutation operators will be left out of the discussion here. With that said, I will still mention the most common mutation operators.

In the case of bits, the simplest mutation operator is the basic unary *toggle*, which also goes by the name *negation* ($\sim$) if we're looking at it from a logic perspective. Otherwise, the most natural binary operators are also from logic: *inclusive-or* ($|$), *exclusive-or* ($\hat{}$), as well as *and* ($\&$).

The final word about these logic operators is they in fact imbue an unnecessary interpretation as to the *type* of a bit. For the highest entropy of design, a bit is a bit, and we should only offer it operators that act on bits. If we wish to apply other mutation operators, we are adding additional type information to our bit. There is nothing wrong with this in principle, so long as we modularize this type system away from our memory constructs. This is to say, equipping our memory constructs with types (and their specific mutation operators) is a modular and secondary consideration.

[4] For the sake of simplicity, if the context is clear, I will ignore the normal convention of string quotes "" within this essay.

Bytes

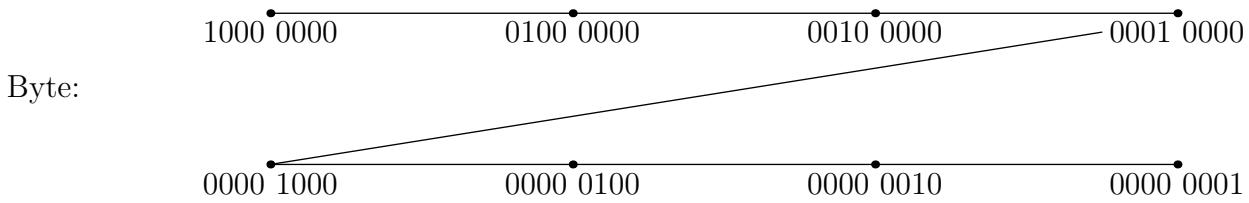
Technically, the next level of memory we should construct after bit memory is *pointer memory*, which is constructed from bit memory, and holds the same number of bits as a computer processor's primary registers. So for example a 64-bit processor has binary address names that are 64 bits long. In practice though, we don't often communicate memory sizes in bits, we communicate them in bytes—which happen to be 8 bits long—and so our processor is an 8-byte processor.

A byte is the unit for which we talk about media memory in the tech industry. As far as I was taught, there's no actual reason a byte has to be 8 bits. It was formalized that way early on, and so that's the standard. Bytes are worthy of our attention though, because when it comes to manipulating memory with respect to computer processors, bits aren't individually processed with respect to our operator commands, instead bytes and multiples of bytes are. Furthermore, when it comes to actual computer architecture, things are more complicated still: Due to processor commands that we have access to (assembly language), we can manipulate individual bytes within pointer memory. What's more, we are also given bit operators, and although we are still technically acting componentwise on full bytes, our bit operators are sufficient to simulate being able to manipulate one bit at a time if we so desire.

The implication of this architectural design—from a narrative perspective—is that we need to include another level of memory between bit memory and pointer memory if we are to maintain our constructive narrative. As such, we need to now model short blocks of binary memory which are 8-bits long.

Byte Memory: *Byte memory* consists of any stable memory location which holds address names that are of length 8 bits long.

Unlike with bit memory, we do need to include a navigational/mutational model for our byte memory. Here is my design: [5]



Our graph is called "Byte". If we wish to have more than one such graph (more than one such byte), we need to pair it with an address name (<address name>, Byte). Otherwise, within our byte, we have eight constant address names, which are linearly connected. Why have I chosen these exact names? How do we navigate them?

We're trying to model byte memory, which consists of 8 bits. Our byte has a name, but we're not concerned with it, as we're only trying to navigate the bit locations for which it is composed. As such, my claim is that an efficient way to do this is to label each bit memory location using the specially chosen byte names which happen to have only one bit turned on:

$$\begin{array}{ll} \{ & \mathbf{1}000\ 0000, \\ & 0\mathbf{1}00\ 0000, \\ & 00\mathbf{1}0\ 0000, \\ & 000\mathbf{1}\ 0000, \\ & 0000\ \mathbf{1}000, \\ & 0000\ 0\mathbf{1}00, \\ & 0000\ 00\mathbf{1}0, \\ & 0000\ 000\mathbf{1} \quad \} \end{array}$$

Why? We need 8 distinct names for each distinct bit location. That much is given. The value in using these names over any others is that we can linearly navigate from one to any of the others using *bit-shift* operators. What's more, we save on interface memory (overhead?) as any bit operator that mutates the bit location of interest is required to make use of the name of the location itself (the single bit turned on in the byte name):

$$\begin{aligned} \text{on}(e_k) &= *e_k \mid e_k \\ \text{off}(e_k) &= *e_k \ \& \sim e_k \end{aligned}$$

where $*e_k$ is the internal state at location e_k (k being the position of the *on* bit), ($\mid$) is the componentwise *inclusive-or* bit operator, (&) is the componentwise *and* bit operator, and ($\sim$) is the componentwise *negation* bit operator well known in C/C++.

[5] Keep in mind, as far as our narrative is concerned, there is often no unique way—nor single best way—to implement a graph construction of a memory name. We only require that whatever whatever graphical pattern we do choose, it should be minimalist in representation, minimalist in navigation, and minimalist in mutation operators.

In terms of other mutation operators, as with bit operators, they suffer from a complexity which is orthogonal to our pointer memory constructive narrative. The most common monoidal operators are of course arithmetic addition, subtraction, multiplication, division. Again, as with bit operators, these specific mutation operators imbue typological interpretations to our byte memory which is best left as an extensional modularization. Regardless, we can describe the algorithm for any given mutation which acts on a byte (and returns a byte) by specifying it in terms of navigating and mutating its internal bits—which is something we can now do.

Words

The strategy given in the previous for *bytes* further scales to arbitrary length *words*, which are addresses whose lengths are powers of two, such as *nibbles* ($2^2 = 4$). I mention words here but choose not to formalize them as they really are just a straightforward scaling of the byte model.

Word Memory: Any stable memory location which holds names whose lengths are of a fixed size, which are also powers of 2, but are no larger than the register size of a computer's processor.

In particular, if the fixed size is $2^2 = 4$, it is known as *nibble memory*; if the fixed size is $2^3 = 8$, it is known as *byte memory*, and for all other sizes 2^n it is known as 2^n*-bit word memory*.

Pointers

Pointer memory itself is straightforward, but this level of complexity is where things begin to splinter as pointer memory allows for cycles and other non-linear navigational paths.

Pointer Memory: Any stable memory location which holds names whose lengths are of a fixed size, which are also byte size powers of 2, and of the same length of bits as the registers of a computer's processor.

The common architecture at the time of writing this essay is 64-bit or 8-byte processors. This means 8-byte registers, or 8-byte pointer memory. We will work with this as our representative example, but just like with words, our design strategy will be such that we can generalize pointer size to arbitrary scales.

As for our navigational model? With pointer addresses, since they consist of 8 bytes we can take a similar approach to our model for navigating byte memory: [6] For each of our 8 bytes, we can assign a byte size name with only one of its bits turned on. Once there, we can mutate the whole byte, or change our scope of complexity, and change single bits at a time.

In this case, we have two levels of interface abstraction. It is a basic theorem of *interface theory* that derivative interfaces are transitive, which is to say: An indirect interface (of an interface (of a given context)), can also be translated into a direct interface of that same context.

In our situation, we can take the general word model and scale to size 64-bits, thus navigating or mutating the bits directly. In practice this would be done to optimize algorithms by inlining, though a compiler should not penalize a user for using either interface approach (direct or indirect), and should optimize either way.

Higher Memory

In C/C++, built on top of pointers are arrays, which are consecutive sequences of pointers.

Segment Memory: Any stable memory location which holds names whose lengths are of a fixed size, which are also multiples of pointer size, but less than the total number of possible pointers given their size.

We can't use the same navigational template model as we did for byte memory, because our segment length may be greater than the underlying register size (an array can have a length greater than 64). This is why we have the additional restriction that the segment length must be less than the total number of pointers (2^{64}). Why? Because we are using distinct pointer names as our model to navigate the component (pointer) memory within segments. In this case, we take our cue from C and navigate segments using pointer arithmetic. As usual, the available mutations are independent of this analysis, though at this level, the common ones again are arithmetic in nature. At the same time, at this level, it also becomes practical to assign other types to our pointer addresses, and so in practice there are also many other mutation operators naturally available to us.

[6] More generally, they consist of powers of 2 in terms of bytes, but as pointer size is never larger than register size, our template model still holds.

Pointer Types

Up until now, our internal states have remained intentionally *untyped*. At this point though—at levels which make use of pointer addresses—it becomes relevant to specify a single structural type to increase the availability of our possible memory constructs:

Pointer Type: A *pointer type* is

1. an internal state which equals the zero pointer (the address of all zeroes of the given pointer size). This pointer is called the *null pointer*.
2. an internal state which refers to another pointer memory location.

Keep in mind some internal states will still remain untyped for our purposes, but by allowing for memory names such as this, we have added new ways to navigate namespaces. As such, this definition allows us then to introduce two other forms of memory at the pointer memory level.

Hook Memory: *Hook memory* consists of

1. the null pointer.
2. two pointer addresses, one of which dereferences to another hook memory location.

The value of hook memory is in the construction of *lists*, or singly linked lists in the STL of C++. You can also interpret hook memory as a substructure of segment memory, in which case you can use the same navigational model, the only difference being you have to dereference the appropriate address to iterate to the next node of hook memory in the list.

Link Memory: *Link memory* consists of

1. the null pointer.
2. three pointer addresses, two of which dereference to other link memory locations.

The value of link memory is in the construction of *chains*, or doubly linked lists in the STL of C++. You can also interpret link memory as a substructure of segment memory, in which case you can use the same navigational model, the only difference being you have to dereference the appropriate addresses to iterate forward (or backward) to the next (or previous) node of link memory in the chain.

Concluding Remarks

To summarize the narrative presented here:

1. We have a set of address names which are finite length binary strings.
2. We partition this set of addresses into bits, bytes (words), pointers, segments, hooks, and links.
3. For each class, we model a navigational interface, as well as an initial mutative interface. Navigational interfaces are transitive, but regardless of choice, the implementation should be inlined (optimized) as a direct interface.
4. When implementing, proper dispatches should be taken for any mutation which has a direct hardware equivalent. Such mutations can be optimized to call their respective corresponding subroutine directly.
5. This constructive narrative is a non-exhaustive specification, and can be extended within the higher memory classes, as well as to include additional classes above them. [7]

Thanks!

[7] Although the stratification ends here, we could keep extending these name levels motivated by higher level memory constructs corresponding to secondary and tertiary storage, or distributed systems for example, as well as the whole internet if we really wanted to go overboard.