SC22/WG11/N247

5/1/91

From: Ed Greengrass
To: CLIDT-ers
Subject: Updated Comments on Objects and Pointers

This message updates my comments on Objects and Pointers (WG11/N232, X3T2/91-072) in the light of the discussion of X3T2 Subgroup 2 in Gaithersburg (as I understand it).

(1) In N232, I used the terms "object" and "instance" more or less interchangeably. In Gaithersburg, there was some sentiment for favoring "instance" because of the conceptual baggage that object-oriented technology associates with the term "object". It doesn't really matter but, if anything, I would favor "object" precisely because I find that baggage helpful rather than misleading. However, "instance" captures the crucial point: that there can be multiple occurrences (instances) of a given value VT1 of a given type T1 and that a Pointer value "points" to (identifies uniquely) an instance rather than a value. Five is a unique value of type Integer. However, two distinct pointers can point to two distinct instances of the value five. In what follows, feel free to substitute the word "instance" for the word "object".

(2) In N232, I defined an object as an association of a unique identifier (UID) and a value. In Gaithersburg, Hamilton carried this one reasonable step further by pointing out that a collection of such objects is a Table by the CLIDT definition of Table. Hence, the generator Object of Type T which I defined in N232 can be redefined as: Table of (T, UID). The UID becomes the Table key. The characterizing operations of Object become Table operations, e.g., Create Object becomes an Insert operation on Table.

(3) In N232, I carefully described and distinguished various CLIDT generators as collections of objects: Arrays are collections in which the individual objects are identified by ordinal value; the identifier is called an Index. A Set is a collection in which the identifier of the object is its value itself, i.e., access is "associative". A Table (I said in N232) was a collection in which the identifier of an object was an application-specific key, e.g., name identifies person, addresss identifies house (or device!), employee number identifies person, purchase order number identifies transaction, etc. By insisting on the application specific nature of the key, I excluded objects identified by UID. I viewed UID's as system generated in the sense that the UID served no function except to distinguish and identify an object within some system scope. However, system generated keys are still keys and hence collections of objects identified by UID's are still Tables. The CLIDT insert operation requires a key value as one of its operands but says nothing about how these key values are generated.

(4) In Gaithersburg, it was pointed out that the scope of the UID space defines how and where a given collection of objects can be referenced. For example, in the CLIPCM case, one can define a global object table, i.e., a UID space that is common to caller (client) and called procedure (server). In that case, client and server can both access directly the "same" collection of objects. A client can pass an object to a server by passing its UID, a value of type Pointer. (Of course, there may be multiple physical copies, locking for concurrency control, etc, but those are physical implementation details.)

On the other hand, client and server may have independent UID spaces. The latter was the situation I envisioned in N232 when i gave the example of a client passing a graph structure (in the graph-theoretic sense) to a procedure as an operand. The graph structure consists of objects, e.g., records, linked together by pointers. The value of each pointer is a UID, i.e., a Pointer points to an object by identifying it uniquely. In order to pass such an operand from client to server, it is necessary to map UID values from the UID space of the client to the UID space of the server (and perhaps vice versa). The exact mapping doesn't matter since UID values are arbitrary. What does matter is that the mapping must be consistent and complete, i.e., each client UID must map into a server UID and all occurrences of a given client UID value (e.g., UIDC1) must map into the same server value (e.g., UIDS1). In that way, the graph structure is preserved in going from client to server in the sense that client and server can traverse the structure in the same way even though the actual UID values may be entirely different.

(5) In Gaithersburg, Yellin raised the question of how one would specify a linked List of Integers in terms of the above concepts. He pointed out one approach which mirrors traditional programming practice. (In fairness, Yellin was not recommending this approach, merely using it as a basis for discussing the Table view of objects.) One could define a table of "integer linked list element" objects. The ugly term in quotes is mine, not Yellin's, but it expresses what Yellin was proposing. The table would be defined as "Table of (integer linked list element, UID)". An integer linked list element, in turn, would be defined as "Record of (intval:Integer, nextval:Pointer)". Each row of thie table would be a linked list element. The UID in nextval for a given row would identify (point to) another row of the table, i.e., the row containing the logically next element in the given linked list. And so on. Notice that this table as defined would contain ALL integer linked list elements defined within the scope of the given UID space. Thus, at any one time, there might be many linked lists of integers in the table, as many as were currently defined in the given scope. Indeed, nothing in the above definition would preclude two or more linked lists merging, i.e., two or more list elements with nextvals pointing to the same element (nextvals containing the same UID value).

There is an interesting point about the above definitions. In the Table definition, I called the type of the key UID. In the Record defintion, I called the type of field nextval Pointer. Since the value space of Pointer is UID's, what's the difference? The difference is in the characterizing operations. The nextval field is a pointer because it makes sense to apply the dereference operation to it to get the next element in the list. However, in the Table definition, the UID functions as a key, identifying the object in which it occurs, not pointing to any other object. Hence, dereference would make no sense and the type is NOT pointer even though the value space may be the same.

One objection to the above conceptual arrangement is that the linked lists are effectively buried in the table. If one wants to manipulate individual linked lists, perhaps linking them together or unlinking them, then one wants a higher level object definition, i.e., a Table of integer linked lists. Each row of THIS table is an entire integer linked list, not a linked list element. The definition is something like, "Table of (intlist:X, UID)" where the type X is --- what? Not CLIDT List because List in CLIDT has no explicit concept of a link. Why would one WANT explicit links at the conceptual level? Aren't links purely an implementation consideration? No. That issue is discussed below. Here, let's just accept the idea of an explicit conceptual link. What is X? X might be Pointer to Intlist element, e.g., each value of X might be a pointer to the first element of a linked list in the table of "integer linked list elements" defined above. In C terminolgy, we would have "intlist *X".

************************ A MAJOR WARNING ***********************

The preceding discussion may have conveyed an extremely misleading impression which I now want to correct. At one time, it was argued by some that Pointer was not a CLIDT type, not a conceptual type, but only an implementation device. In X3T2/90-038, I explained in detail why Pointer IS a conceptual type that belongs in CLIDT and not a mere implementation device. I won't repeat that argument here. However, it strikes me that the discussion in this paper may tend to reinforce the implementation view of Pointer. After all, haven't we discussed linked lists? Isn't that exactly the kind of thing that people have in mind when they think of implementations using pointers? And tables with system generated UID's. Doesn't that sound like the kind of system table that might sit in an operating system? Don't objects sound a lot like plain old programming language variables? Where is the conceptual (as opposed to implementation) aspect of all this?

Let me take linked lists as an example of what I have in mind. Is a linked list a concept or an implementation strategy? Answer: It is both. That programmers use linked lists in actual implementations is too obvious to discuss further. But what is a CONCEPTUAL linked list? It is not hard to think of examples. Consider the concept "eldest child", a concept that used to have a lot of significance with regard to the law of estate inheritance. That concept, that relationship, defines a conceptual linked list down through the generations of a family. My father Robert was the eldest child in his family. I (Edward) was the oldest of his children. My daughter Mara is the oldest child in my family. We have a conceptual linked list: ((Robert Edward), (Edward, Mara), (Mara, Null)). Here the links are associative: human names. However these names are not unique. Hence, we may replace them by system generated UID's; in the U.S., We might replace them by Government-assigned Social Security Numbers (SSN's) but the distinction between SSN's and UID's is not important for the present discussion. In each case, we choose the UID of the eldest child. The link is defined by a concept, a relationship. This conceptual linked list might be implemented quite differently, e.g., by a sequential linear list in which the list elements did not contain links at all.

Conceptually, how does a linked list differ from a List as defined in CLIDT? The answer is that the ordering of the list is explicit. For instance, in the above example, Edward follows Robert not because Edward was inserted directly after Robert (as would be the case with a CLIDT List) but because Robert has an explicit link whose value is Edward. (To express the fact that the links were chosen according to an eldest child rule would require additional characterizing operations in a wider conceptual domain, outside of the scope of CLIDT.) Replacing names by UID's preserves this characteristic and avoids the problem of duplicate names.

Suppose that a client calls a procedure according to the CLIPCM model and passes that procedure a linked list. Which kind of linked list are we talking about now? In Gaithersburg, Barkmeyer suggested two levels to CLIPCM: (1) the procedure call model and (2) the actual protocol. In the model, the linked list is conceptual in the sense of being defined in terms of CLIDT types. At the protocol level, an abstract syntax and transfer syntax for the actual exchange of this linked list must be specified; now, we are getting down to implementation considerations, e.g., how is the linked list to be encoded for exchange?