A FORTH Implementation of the Heap Data Structure*

W. B. Dress
Instrumentation and Controls Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

The use of the heap for memory management provides the FORTH programmer with a versatile tool. Its use speeds program development at the conceptual level by allowing the program designer a means to consider dynamic arrays, garbage collection, and overlays; and at the implementation stage by providing a framework for easy manipulation of data structures.

The need for a heap memory manager arose during a project to rewrite OPS5, a popular expert-systems language, to respond to real-time events. Once the syntax and underlying algorithm of OPS5 were incorporated into a FORTH environment, the language could be extended towards the goal of a multitasking, real-time expert-system tool. It was at this point that the need for garbage collection became evident. A subject of numerous research papers, book chapters, and vendor hype, garbage collection is a means of marking list nodes (linked data elements) as unneeded and reclaiming memory space when it is almost exhausted. A feature of this process is that the typical application software ignores reclamation, leaving it to a garbage-collection cycle which can last a very long time indeed. Strictly speaking, garbage collection could be fatal in a real-time system.

With the Macintosh heap memory manager [1,2] as a model for the basics of memory allocation and deallocation, it became a matter of a few hours to write code implementing a simple, but adequate, heap data structure in Multi-FORTH [3]. The fundamental idea of the heap is to reserve a large pool of memory which is partitioned into two major portions: the handles and the pieces. The pieces are blocks of memory of arbitrary size which are reserved for use by a call to the heap manager. The pointers to these pieces are contained in "handles" which reside at the top of the heap. Figure 1 shows the basic internal structure as implemented.

---

*Research performed at Oak Ridge National Laboratory and sponsored by the U. S. Department of Energy under Contract No. DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc. An expanded version of this paper, including the source code, has been submitted for publication in *The Journal of FORTH Application and Research*.

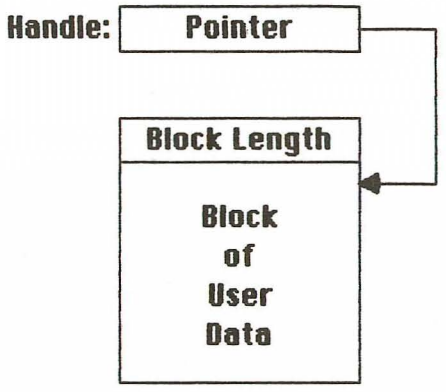

Figure 1. Pointer and Handle Relationships within the Memory Heap.

When a block of memory is required, the address of the next available memory location is written to the next available handle cell, the memory-location pointer is advanced by the size of the piece requested, and the handle (address of the pointer to the memory actually allocated) is returned on the stack. The user application is responsible for keeping track of the handle until it is no longer needed. A call should then be made to the heap manager to release the handle thus deallocating the referenced memory block. At this point, the collection of allocated blocks is compacted to recover the unused space. Such holes are moved to the top of the allocated portion for later use. Figure 2 illustrates this process.

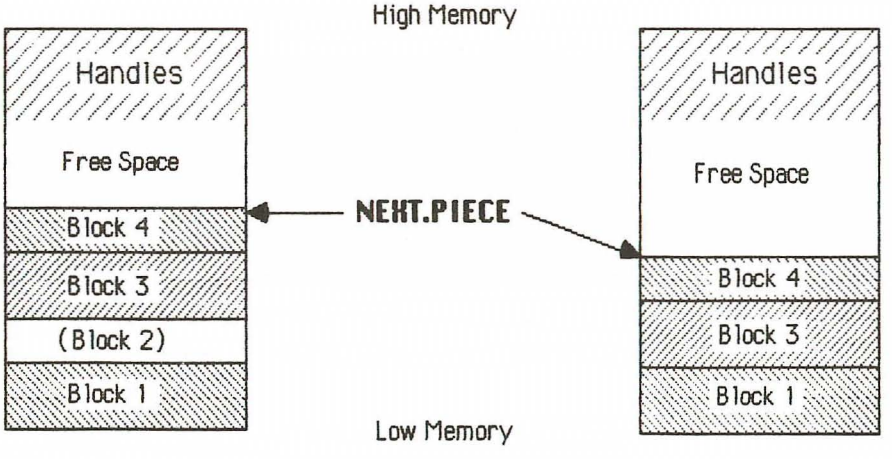

Figure 2. Removal of Block 2, Showing the Heap Before and After Compaction.

Since the region of the heap containing the handles is never moved once the heap is created, the handles can be thought of as absolute memory locations containing pointers to blocks of relocatable memory. The virtue of such activity is the optimal use of memory even when the size and temporal characteristics of data blocks are apriori unknown.

The use of the heap to solve (more properly, avoid) the garbage collection problem involved running the system in a synchronous mode. Thus each time a block of memory was needed, it was reserved by a simple call to the heap manager. Alternately, when the space was no longer needed, it was returned to the heap via the handle, when the memory was reclaimed. Removing the top block involved only a pointer update and a handle reset.

The use for data overlays is straightforward unless the data contain references to other nonresident data which must then be located and brought into the heap. Assume the compilation to have occurred earlier within the normal FORTH dictionary structure and that the compiled code has been copied to mass storage and then forgotten. To execute the stored code, simply allocate heap space for the stored code, read it into the heap and execute the sequence " @ PFA CFA EXECUTE " after the handle to the code has been placed on the stack. Note that this direct execution of the code assumes that all PFAs in the heap refer to code or data currently resident in main memory, and at the same locations as at compilation time.

The problem (insurmountable in some languages) of arrays whose size is unknown until run time is one that most every FORTH programmer meets sooner or later. The usual solution is to allot enough memory to cover every conceivable case, which is wishful thinking for an application to be turned over to an end-user. The use of the heap is quite simple in this case; the size, determined at run-time, is used to request heap space which is then used for the array. If the allotted size becomes insufficient as determined by the usual bounds tests, the piece can easily be resized. When the array is no longer needed, the space is returned to the heap.

Most any type of data can be represented in the form of a linked list. The convenience of allowing such lists to be made and destroyed at run time is a solution to the problems of sorting and data-base searches. Since efficiency requires as many keys as possible to be present in memory and the data base is almost certainly larger than available memory, a rapid means of allocation and deallocation is necessary. This is particularly true when lists of varying sizes are arranged without the benefit of suitable keys.

The simple implementation presented here is useful only for single user systems (who releases a structure?) and where absolute memory references are only made outside the heap. A natural extension of the memory heap is to add features allowing absolute references to in-heap data and resolving conflicts between different users. An approach to extending the usefulness of the heap would be to allow an extra byte in each memory block, or in the handle itself for a tag field containing bits for marking (as in standard garbage collection), for information as to the absolute nature of the data contained in the referenced block, as well as a flag to indicate the presence of a semaphore field in the block (for resolving mutliuser requests). A few verbs could be written to make appropriate use of the tag field according to the needs of the application. For example, one would add verbs to lock a block in memory (assuring validity of absolute memory references) and a means to mark a block as purgeable (can be removed on the next heap compaction) and unpurgeable. A means of reallocating a block via an additional level of indirection is also possible.

The uses of the heap data structure, even in its simplest form, are varied and versatile. Ranging in complexity from the simple dynamic array to the FORTH program overlay in multiuser systems, the heap memory manager has a place in the toolbox of every FORTH programmer. Once familiar with the heap concept, the programmer can defer thorny issues from the design stage of a problem, where details do not belong, to the implementation stage where they are necessary.

The ease of bookkeeping necessary for many volatile list structures makes the heap a natural manager for LISP-like extensions to an application. The idea of keeping a list of handles in the heap (a handle to a block of handles) provides a way to manage second order data structures in the same convenient manner as primary ones.

The heap as presented here has been used for some time in our FORTH-based version of the expert-systems language, OPS5. This has resulted in faster execution times for OPS5 programs and overcome problems in extending the language to the real-time domain.

---

[1] Apple Computer, Inc., *Inside Macintosh*, Cupertino, California, 1984.

[2] Creative Solutions, Inc., *MacFORTH Level Two User's Manual*, Rockville, Maryland, 1985.

[3] Creative Solutions, Inc., *Multi-FORTH Version 2.00 User's Manual*, Rockville, Maryland, 1984.