# C. DYNAMIC MEMORY MANAGEMENT

String processing requires memory to be allocated for string storage. New strings may be input or created, old strings discarded, and strings in general may expand or contract during this processing. This requires some means of allocating storage in segments of variable size, and "recycling" unused space for new data.

The Forth language has only two structures for allocation and de-allocation: the dictionary, and the stack. Both of these are Last-In First-Out allocators, releasing memory only in the reverse order of its original allocation.

Since this is an unacceptable restriction, patternForth requires a dynamic memory manager.

## 1. Design

There are four design criteria for a memory allocator.

> a) Efficiency of representation. Data should be stored in a form which can be efficiently processed by the end application.
>
> b) Speed of allocation. The time involved to obtain a new segment of memory would preferably be finite and deterministic; ideally, short and invariant.
>
> c) Speed of "recycling". Likewise, the time make a segment of memory available for re-use, would preferably be finite and deterministic; ideally, short and invariant.
>
> d) Utilization of memory. The amount of memory which is made available for data should be maximized. This implies:
>
> > * memory should be "parceled out" in such a manner as to minimize "wasted space";
> >
> > * the allocator should use as little storage as possible for its own internal data;
> >
> > * the amount of memory which is made available through recycling should be maximized. Ideally, storage which is recycled should be completely re-usable. This is particularly significant since the allocate-recycle- allocate cycle will repeat endlessly.
>
> A memory allocator should solve, or at least address, the problem of "fragmentation." This is the division of memory so as to leave small, unusable segments.

All memory managers represent a tradeoff among these four criteria. The Patternforth memory manager design weighted them as follows:

> a) Efficiency of representation. We decided that each string would be stored in a contiguous block of memory. The machine-level string operations which are available (on the 8086) are much more suited to incrementing pointers than traversing linked lists.
>
> b) Speed of allocation. The time required to allocate a segment of memory could be variable, but bounded. We desired an "upper limit" on the search time for a suitable segment.
>
> c) Speed of "recycling". The time involved to recycle a segment of memory could be variable, but bounded. In particular, we wished to avoid the often-seen problem of halting for several seconds while memory is consolidated.
>
> We valued recycling over allocation. We strongly desired a recycling scheme whose time is short and fixed.
>
> d) Utilization of memory. This was the least emphasized of the four criteria. Since we anticipated the use of long strings, we favored schemes having a "fixed" overhead (X bytes per string), as opposed to a "variable" overhead (X bytes per character).

## 2. The Boundary Tag Representation

The method of memory management we are using has been described by Knuth [KNU73] as the "boundary tag method," but it can be (and was) deduced from first principles as follows:

Assume as given that memory is to be allocated from a large area, in contiguous blocks of varying size, and that no form of compaction or rearrangement of the allocated segments will be used.

The process of allocation is illustrated in Figure C-1a. To reserve a block of 'n' bytes of memory, a free space of size 'n' or larger must be located. If it is larger, then the allocation process will divide it into an allocated space, and a new, smaller free space.

After the free space is subdivided in this manner several times, and some of the allocated regions are "released" (designated free), a picture similar to Figure C-1b will form.

Note that the free segments designated (1) and (2) are distinct segments, although adjacent. (Perhaps each was just released from a prior use.) This is a sub-optimal arrangement. Even though there is a large contiguous chunk of free space, the memory manager perceives it as two smaller segments and so may falsely conclude that it has insufficient free space to satisfy a large request.

For optimal use of the memory, adjacent free segments must be combined. For maximum availability, they must be combined as soon as possible.

"As soon as possible" is when this situation is first created. The task of identifying and merging adjacent free segments should be done when a segment is released.

> * Identification: it is easy to locate the end of the preceding segment, and the beginning of the following segment; these are the memory cells adjacent to the segment being released. The simplest identification method would be to have the neighbors' "used/free" flags kept in these locations. This means that each segment needs "used/ free" flags at both ends. (That is, at the boundaries -- hence, "boundary tags.")
>
> * Coalescence: assume for the moment that the necessary bookkeeping information is kept at the beginning of the segment. To merge two or three adjacent blocks, it is necessary to change all of their bookkeeping information. It is straightforward to find the beginning of the following segment. To find the beginning of the preceding segment, either a pointer to the beginning of the segment, or an offset (i.e. the segment length), should be stored at the end of the segment. There are advantages to storing a length value rather than a pointer. Since this is part of the bookkeeping information, it is present at both ends of the segment, and can be made part of the boundary tag.
>
> (A similar conclusion can be reached if the bookeeping information is stored at the end of the block.)

Thus, at both ends of every segment, an "allocated or free" status bit and a segment length are stored. Figure C-2 illustrates this "boundary tag" system.

In the current patternForth implementation, the dynamically allocated region is a 64K byte region separate from the 64K Forth region. The largest string which can be allocated in this "string space" is 64K bytes, and two bytes suffices to hold the length of any string.

The boundary tag is two bytes, with the high 15 bits used for length information, and the low bit as the "allocated" flag --0 meaning "free," and 1 meaning "in use." The tags at both ends of a segment are identical. The smallest possible segment is two bytes, consisting only of a single tag, which appears to be at both "ends."

It is impossible to represent a segment of one byte. To ensure that no combination of allocations and releases leave a one-byte fragment, all allocations are restricted to multiples of two bytes. This is why the least significant bit of the length can be used to hold a flag.

The value actually stored in the boundary tags is "length - 2." This causes the two-byte empty cell to contain zero.

The boundary tag method, consistently applied, ensures that there will never be two adjacent free segments. This guarantees the largest available free space short of compacting the string space.

## 3. Allocating Memory

To satisfy a storage request of length 'n', the allocator must examine all of the free segments in the "pool," or at least as many as needed to find a suitable candidate. Two search criteria are commonly used:

> * "first fit," in which the first free segment encountered, which is larger than n, is used; and
>
> * "best fit," in which the segment which is closest in size to n, equal or larger, is used.

For simplicity and speed of allocation, we have chosen the "first fit" approach. This may lead to more fragmentation but this problem can be alleviated somewhat by adding a "roving pointer" to the allocator (to be discussed shortly). This, and the abundance of memory, should minimize the problem of fragmentation.

**a) the algorithm**

A segment of memory is allocated as follows:

1. get the starting search address in the string space

2. is this segment free, and is its length greater than or equal to the needed length?

2a. YES: divide this segment into two parts: one of the needed length, and the remainder. If the free segment is exactly the needed length, there will be no remainder.

2b. Set the boundary tags in the allocated part, and in the remainder (if any). Exit with success.

3. (segment not free or too small) Add the length of this free segment to the address pointer. Wrap around to the beginning of memory if necessary. This wraparound is automatic on the 8086, since we are using 16-bit addressing in a 64K region.

4. have we returned to our starting search position?

4a. YES. All of memory has been searched, with no success. Exit with failure.

4b. NO. Repeat with step 2.

This logic is implemented in the Forth word ALLOC. The calling sequence is

```
n ALLOC
```

where 'n' is the length of the data for which memory is to be allocated. The overhead of the boundary tags is added internally by ALLOC. ALLOC returns the address, in string space, of the allocated segment.

**b) the roving pointer**

The algorithm just given does not specify where to begin the search. Since the allocated and free segments form, via their lengths, a circular list, the search can start at any segment. (But it must start at the beginning of the segment.)

One strategy is to always start at the base of the string space. Knuth proposes an better choice: a "roving" pointer which, ideally, always points to the largest free region remaining. (With a first fit strategy, this would ensure that allocations succeed or fail immediately.) Since it is impractical to search for the largest free region, Knuth proposes a compromise: set this roving pointer to the segment immediately following the last allocation performed. In almost every case, this will be the free "remainder" chunk left over from the allocation process. Often this remainder will be sufficient for the next allocation. (On the first allocation pass through the string space, this "remainder" is always the largest free region left.)

This technique is used in patternForth.

## 4. Memory Release

The strategy for release has already been described: when a segment of memory is released, immediately attempt to combine it with the adjacent segments to form a larger free segment.

**a) the algorithm**

A segment is released as follows:

1. clear its "allocated" flags.

2. is the preceding segment free?

2a. YES. combine the segments. Update the two tags at the new "ends" with the combined length.

3. is the following segment free?

3a. YES. combine the segments. Update the two tags at the new "ends" with the combined length.

This algorithm works correctly if either, or both, of the adjacent segments are free. (The result of step 2a is a "valid" which will be correctly handled by step 3.)

This logic is implemented in the Forth word RELEASE. It runs in "almost fixed" time; there are no loops, and only two conditional jumps. The calling sequence is

```
adr RELEASE
```

where 'adr' is the address of the segment, as returned by ALLOC.

### b) the roving pointer

A potential hazard exists if the roving pointer was pointing either to the segment being released if the preceding segment is free, or to the following segment if that segment is free. Once the coalescence has taken place, the roving pointer will be pointing, not to the start of a valid segment, but somewhere inside a merged segment. The search logic requires the starting point to be the beginning of a valid segment. If a search is initiated inside a segment, the allocator will certainly crash.

One solution would be to check for the two cases listed above, and when they occur, to reset the roving pointer to the beginning of the combined free segment. To eliminate this testing overhead, however, this implementation uses the simpler (but perfectly valid) approach of always resetting the roving pointer to the beginning of the combined free segment, whenever any segment is released.

## 5. Future work

### a) free list

The allocation algorithm as described above, and as currently implemented, treats all of the string space as a circular list of intermixed free and allocated segments. A substantial portion of its time is "wasted" as it tests those segments which are already in use -- since the boundary tag method guarantees that at least half of the segments which are examined will be allocated!

This time could be saved by confining the allocation search to just the free segments. This can be done by keeping the free segments in a separate linked list [KNU73,ROD85]. Since these links are only required in the unused segments, there is no space penalty. (Actually, there is a small penalty: "granularity." The system must ensure that any possible free segment is large enough for a link and two end tags. The easiest way to do this is to constrain all allocations to multiples of six bytes, rather than two.)

Of course, there is some complication in the allocation code, since it must unlink an element from the list (possibly leaving a "remainder" linked); and in the release code, since it may need to combine free segments.

### b) integrity checks

Another problem with the current implementation is the fragility of the links (tags). Should the application program -- which has complete access to the string space -- damage the links, or should the memory manager run amok (perhaps due to the roving pointer being corrupted), the circular list of memory segments will quickly be destroyed.

Fortunately, there is a modest safeguard built into the boundary tag method. Each segment has two copies of the tag, which must match. And, given one tag, the other can be located easily. So, an "integrity check" which compares the two tags would need only a few cycles of CPU time.

This integrity check could be performed at allocation time, release time, or both.

### c) paragraph allocation

A possible variation for the 8086 architecture is a "paragraph-allocated" memory space. This could extend the memory allocator for a space of up to 1M byte -- the full 8086 addressing range -- while remaining within the constraints of a 16-bit Forth model.

The 8086 architecture divides the 1 MB address space into "paragraphs" of 16 bytes each. There are 64K paragraphs, so each can be uniquely identified by a 16-bit "paragraph address." The 8086 processor addressing model defines four 64K byte "segments," each starting on a paragraph boundary identified by a 16-bit "segment register."

By increasing the "granularity" of the memory allocator, so that all allocation is in multiples of 16 bytes, all memory segments can be identified by paragraph address, and the full 1 MB space can be dynamically allocated.

There may be improvements at the machine code level, as well. Basing all data structures on a paragraph boundary allows (indeed, requires) the use of the segment registers as "base registers." The additional addressing flexibility gained by having additional base registers may simplify or speed up the code.

**d) EMS memory**

An IBM PC variation, for even larger memory spaces, would use EMS memory. The Lotus- Intel-Microsoft Expanded Memory Specification allows up to 8 MB of memory to be added to an IBM PC [DUN86]. This memory is accessed through a bank-switched 64K region.

Undoubtedly, an address of larger than 16 bits will be necessary to do justice to EMS memory. (A granularity of 128 bytes seems excessive.) There are other issues to be addressed:

* whether the memory manager should use the "official" EMS manager, for the sake of portability, or direct hardware access, for the sake of speed;

* how to ensure that, in the bank switched environment, strings are always available in a contiguous form (i.e., not "broken" across two banks); and

* allocation and bank switching strategies for optimum speed.

**e) memory compaction**

It may happen that the fragmentation of the memory space which still occurs will not leave sufficiently large free segments for the operation of a particular program. This would indicate a need for memory compaction.

Memory compaction involves moving all of the allocated segments to one "end" of the string space, all adjacent, so that all of the free segments are combined into a single large free block extending to the other end of memory. Since this involves moving data, it will invalidate pointers to the data. For this reason, most compaction schemes require a "master pointer" to each item in the allocated region, so that only one pointer need be adjusted when any item is moved. Although patternForth does not presently use compaction, provision has been made for master pointers. (See next section.)

Most memory compaction is performed "on demand," i.e., when the program runs out of space. This is likely to be a disadvantage in a real-time application, where an unexpected "pause" for a few seconds, while memory is reshuffled, will disrupt the process. An "incremental" memory compactor would be worthy of study.