

F

The Heap Manager

This appendix describes heap management. It contains the following sections:

§F.1: *Overview*

§F.2: *The Heap Chain*

§F.3: *The Free Chain*

§F.4: *Heap Integrity Checking*

Note: This appendix applies to you only if you are using the MetaWare-supplied (ANSI) library. If you compile and link with option **-Hsyslib**, this appendix does not apply.

F.1 Overview

The *heap* is a region of memory used for data structures whose size is not known at compile time and whose size might change during the course of the program. These are called *dynamically allocated* data structures.

A program can request (*allocate*) memory from the heap or return (*free*) previously allocated space to the heap at any time during its execution. These are the basic processes for manipulating the heap, provided to C programs through the C library functions `malloc()` and `free()`.

The data structures that reside in the heap come into existence, grow, shrink, and disappear in an unpredictable fashion. There are two important requirements for this:

- The operating system must be able to reclaim memory that is no longer being used by the program.
- The mechanism for organizing heap storage must be very flexible. For example, a stack organization cannot be used, because it is not necessarily the case that the last data structure allocated is the first freed.

The heap organization is flexible. Each entry in the heap contains several bytes of information about the heap space it represents. Each item in the heap is either allocated or free, and the free items are doubly linked in a chain.

Note: Because the specifics of the heap-management software are subject to change, the information presented here is provided for debugging purposes only.

F.2 The Heap Chain

The heap information itself is stored in the heap. Each heap entry consists of three units of information: the size of the current data entry (four bytes), the size of the previous data entry (four bytes), and the user data being stored in the entry.

Table F.1 shows items of information contained in each heap entry.

Table F.1 *Items of Information in Each Heap Entry*

Item	Bytes	Contents
Size	4	Size of data entry
Prev_Size	4	Size of data entry before current item in heap
User_data	variable	Size of user data stored in the entry

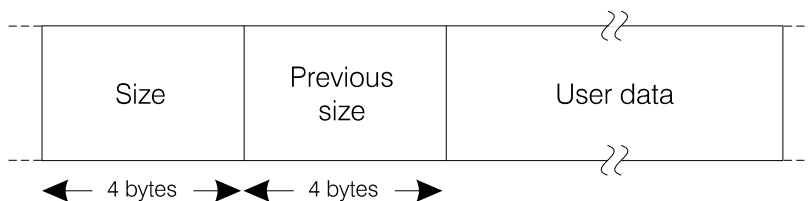


Figure F.1 *Heap Item*

Suppose we have the following heap activity:

```
p1 = malloc (10);
p2 = malloc (10000);
p3 = malloc (512);
p4 = malloc (2048);
```

Then the heap will have four entries on it:

```
    10 bytes (allocated)
10,000 bytes (allocated)
    512 bytes (allocated)
 2,048 bytes (allocated)
```

Assume that after all the calls to `malloc()`:

```
p1 = 0x6e74
p2 = 0x6f20
p3 = 0x9638
p4 = 0x9840
```

Then the heap looks like this:

Addr	Size	Prev_Size	Data
0x6e6c:	14	24	User data starting at 0x6e74
0x6f18:	2718	4c	User data starting at 0x6f20
0x9630:	208	2718	User data starting at 0x9638
0x9838:	808	208	User data starting at 0x9840

`Addr` is not part of the heap node; it is the address at which the entry is found. The entry takes eight bytes. Eight bytes from the beginning of each list entry is the address pointed to by the allocation pointer.

Figure F.2 shows what the heap looks like after these four calls to `malloc()`. The first call to `malloc()` finds a chunk of memory in the free chain large enough to hold 20 bytes. This accounts for the four allocated blocks not being contiguous. The first allocated block has size 20, not 10 bytes plus 8 bytes overhead, because blocks allocated by `malloc()` are four-byte aligned. Therefore `Size` modulo 4 is always 0 (zero). The `Prev_Size` values of 36 and 76 shown in the `malloc()` of `p1` and `p2` represent previously allocated pieces of memory.

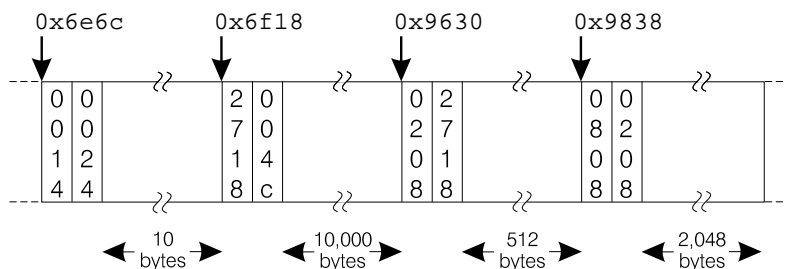


Figure F.2 Heap Chain After Four Calls to `malloc()`

F.3 The Free Chain

The chain of free items can be searched to find a free item big enough to hold a requested amount of memory. However, it would be nice not to have to consider items that are not close to the requested size when searching for free space.

To avoid the expense of chasing the free-chain links that are associated with allocated items, the heap manager maintains separate lists that contain only free items that fit in a predefined size range. When an item is freed, the space that used to contain user data becomes available to the heap manager. At the beginning of that space are the free-list pointers. The *free list* is a doubly linked list of free items. The heap manager has a static array, the *free-chain pointer array*, that holds an array of pointers to the first item in each free list. When an item is freed, it is placed at the front of the appropriate list and the free-chain pointer is modified to point to it.

```
P = malloc(some_amount);
```

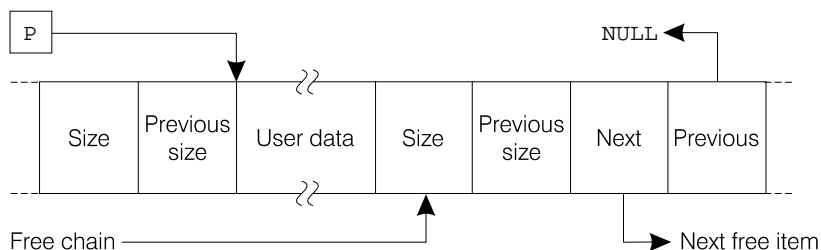


Figure F.3 Heap Chain Before Call to `free()`

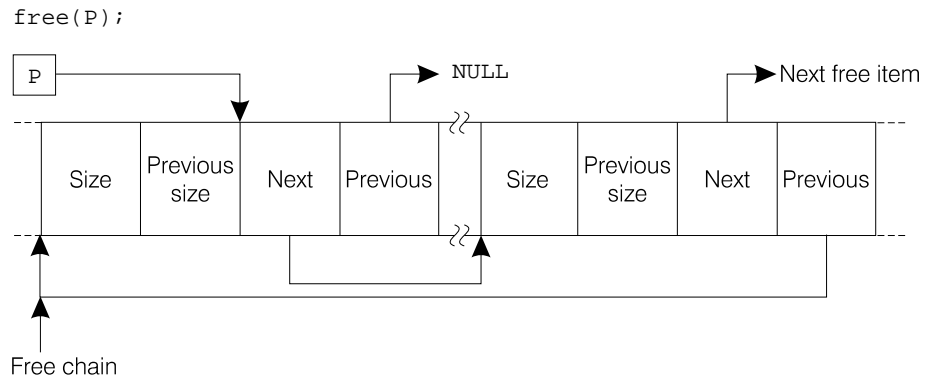


Figure F.4 *Heap Chain After Call to `free()`*

A free-chain link consists of two pointers. These two pointers are located in the memory that is being added to the free chain, immediately following the heap-chain data. One of these pointers points to the next free block of memory; the other points to the prior free block of memory.

When space is requested from the heap, the heap manager allocates the first free item found that is big enough to hold the request; if it is larger than the request, it is split into two items, the first allocated, the second free. When an item is freed, if the item on either side is free, the items are coalesced into one free-chain item.

There is no minimum number of bytes that the heap manager allocates to a user. An item on the heap can be as small as one byte in addition to the heap-chain overhead. Therefore, there can be items that are too small to contain free-chain information, because the free chain is written on the space that previously held user data. When an item that is too small to hold the free-chain pointers is freed, it is marked as free but not placed in the free chain. Though free, the space cannot be reused until one of the items next to it is freed, and the items are coalesced, which guarantees room for the free-chain pointers in the resulting item.

Note:

On some systems, the memory allocated can be aligned to two-byte or four-byte boundaries. This gives the appearance of a minimum heap-item size, but this is not the case.

F.4 Heap Integrity Checking

The heap manager provides integrity checking. To control the level of heap integrity checking, you set environment variable `MALLOC_LEVEL` on the command line. (If your compiler runs on an extended DOS host, you can also set it in your `autoexec.bat` file.)

These are the levels of heap integrity checking:

- 0 No integrity checking.
- 1 Double links are used, which permits better coalescing of free space. No integrity checking. This is the default.
- 2 Items are checked as they are allocated and freed within the heap.
- 3 Complete heap integrity checking is performed at every call to `malloc()` and `free()`.
- 4 Complete heap integrity checking is performed at every call to `malloc()` and `free()`. In addition, if heap corruption is detected, a post-mortem heap dump is printed to standard output.

For example:

```
set MALLOC_LEVEL=2
```

First fit versus best fit By default the heap manager uses a first-fit allocation strategy. However, it is also capable of using a best-fit allocation strategy, which results in more efficient use of available memory at the cost of run-time performance. To configure the heap manager to use the best-fit strategy, append the letter `b` or `B` to the numeric value you assign to `MALLOC_LEVEL`:

```
set MALLOC_LEVEL=2b
```