# The Heap

In most systems the Heap is a large, static chunk of memory that can be allocated dynamically.

The heap does not coordinate with function calls the way the stack does. Functions can allocate memory from the heap that persists after the function returns.

We can allocate a 1MB heap with

```
.comm Heap, 1000000
```

There are 3 issues with heap management:

- How to allocate chunks of memory from the heap.
- How to manage fragmentation of the heap.
- How to deallocate memory from the heap that was previously assigned and no longer needed.

Note that all of these must be handled at runtime.

# I. Allocation

How would you manage allocation? The heap is a finite resource. While you can't prevent a program from requesting more memory than is available, you want to manage the resources so that you will be able to satisfy as many requests as possible.

# Good allocation strategies

A. are quick
B. don't overly fragment the heap, which would make subsequent allocations difficult
C. try to allocate related data in nearby locations, to minimize page faults

All strategies maintain a HeapTable that lists all allocated chunks of the heap. Each chunk has a header that says

a) the size of the current chunk
b) the amount of the chunk currently in use
c) a pointer to to the start of the object contained in the heap

Almost all strategies allocate chunks of data that are actually larger than requested by the program to allow for dynamic allocation. For example, if a program requests 13 bytes the system might actually allocate a 512-byte or 1K block. A second request for a small amount of the heap might actually come from this block that is already allocated. This helps to reduce the fragmentation of the heap.

Here are 3 popular allocation stragegies:

- Best Fit: The heap is searched for a chunk of memory that most closely matches the requested size.
- First Fit: The heap is searched for the first chunk of memory at least as large as the requested size.
- Next Fit: Looks through the heap for the first chunk of memory that is large enough, starting at the location of the last alloation.

How do these do with our 3 criteria?

# II. Defragmentation

Fragmentation is a problem because a highly fragmented heap may not allow a large allocation even if the total amount of memory is sufficient.

Naturally, everyone tries to allocate in such a way as to minimize fragmentation, but it will inevitably occur.

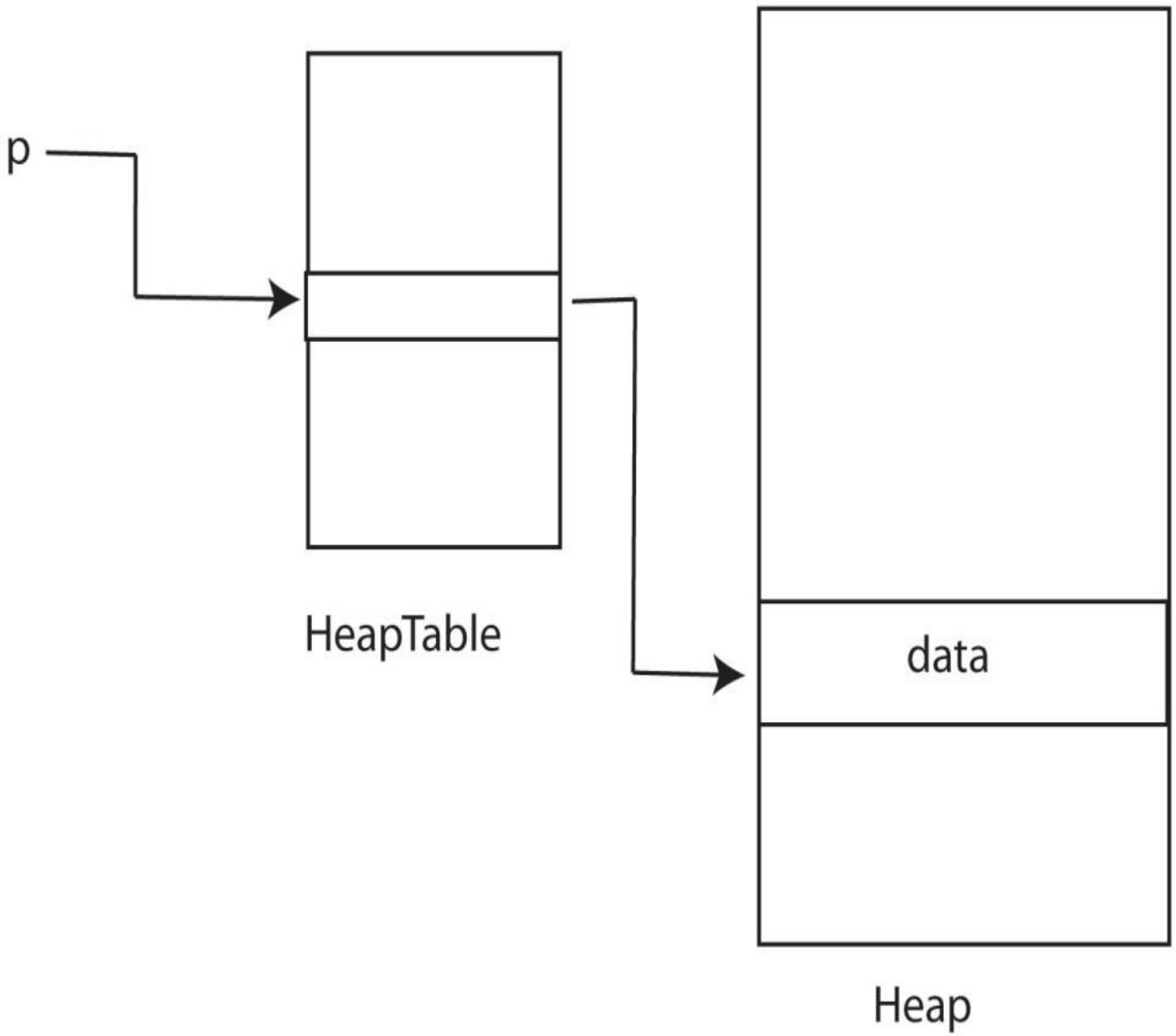
Once the heap is fragmented, the only possible way to defragment it involves moving data that is in blocks that have already been allocated.

Consider a C program that has a line of code

```
p = (int *)malloc(23*sizeof(int));
```

At some point in the future the defragmenter will want to change the address p refers to. How can it do that?

It is not possible to find all of the memory locations that hold the value of p. Instead, we make the value of p be an address in the HeapTable. That address will stay fixed throughout the life of p. However, we can use the HeapTable as an additional level of indirection -- the table entry will be an actual address in the heap where the data p points to will be stored.


p
HeapTable
data
Heap

This allows us to move the block that the HeapTable points to for p.

When a trigger is reached an algorithm runs through all of the allocated blocks, moving them together to leave large free blocks of the heap. Each time a block is move its address in the HeapTable is updated.

Here are some possible triggers for defragmenting the heap:

- The largest free block is less than some value
- The average free block is less than some value
- The allocation algorithm needs to check X locations to find a suitable location.