Wanted: dynamic allocation of memory.

because sometimes the program does not know at compile time how much memory space it will need.

Solution: design the O.S. to provide contiguous chunks of memory similar to `sbrk + mmap`

- these are expensive (time), so we only want to do them occasionally for large pieces of memory.
- too hard for programmer to use. We want cleaner, easier interface.
To allocate:

```c
void *malloc(size_t size);
```

address of first byte within the contiguous allocation

To deallocate:

```c
void free(void *ptr);
```

address of first byte within the allocation

Example:

```c
nodeptr = malloc(sizeof(struct node));
free(nodeptr);
```
C expects the programmer to do memory management.

Other languages have the system do garbage collection
(examples: Java, C#)
Implementation possibilities for `malloc()` and `free()`

allocate fixed size chunks

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Only needs a bit map of free/allocated chunks.
- Allocation is easy: find free chunk within bit map, mark it as allocated, return ptr to the chunk.
- Deallocation is easy: mark chunk as free in bit map
Variable size allocator needed. Assume we have a large quantity of free space. Maintain a free list.

Allocation walks the free list to find a large enough chunk.

Efficiency depends on:

1. Order of the list
2. Size of the list

by size? by addr? by time of allocation?
Which free portion to choose in allocation?

**Best Fit** search free list to find piece that is larger than needed, but by the smallest amount.

Example: `malloc (45)`

Free list has

```
20 40 44 50 52 100
100000
```

After allocation, free list has

```
5: 20 40 44 52 100
100000
```
Best Fit considerations:

+ less "wasted" space overall

- can be time consuming to find the best fit. Unless free list is ordered by size, must search entire list.

- less wasted space can mean lots of small pieces. Can make free list bigger. Perhaps those small pieces cannot be used (as they are too small).
Worst Fit

Find the largest free portion and allocate part of it.

Same example: `malloc(45)`

20 40 44 50 52 100 100000

After:

20 40 44 50 52 100 99955

Like best fit, it is time consuming if the entire free list must be searched for the largest portion.
First Fit

The first block in the free list that is big enough is used.

Same example: malloc(45)

Assume initial free list:
44 20 50 100 100000 40 52

After:
44 20 5: 100 100000 40 52

+ Faster search, and list does not need to be ordered by size.
- Can lead to pollution, where beginning of list has large number of small items.
Next Fit

A variation on First Fit, where an extra pointer to the position within the list last allocated.

+ less pollution.
Strategies for `free()`:

Return to free list.

Assume we free that 45 byte chunk this list

```
20 100 100000 44
```

- add at end

```
45
```

- add at beginning

```
20 44 45
```

Keep ordered list
An issue: Each `free()` contributes to a larger number of items in the free list, and leaves the space in broken pieces.

Assume heap memory:

<table>
<thead>
<tr>
<th>20</th>
<th>44</th>
<th>45</th>
<th>100</th>
<th>100,000</th>
</tr>
</thead>
</table>

↑ to be freed.

Result:

<table>
<thead>
<tr>
<th>20</th>
<th>44</th>
<th>45</th>
<th>100</th>
<th>100,000</th>
</tr>
</thead>
</table>

Now, assume `malloc(150)`. It fails!

We need to coalesce free chunks.
Coalescing is not difficult if the free list is ordered by address.

Check if previous is next to newly freed chunk.
Check if next is ...
Next issue:

`free(ptr)` does not provide size info.

If it did, `free(ptr, size)` then a program bug could really mess up the heap.

So, the library must maintain the size info itself.

Possibility: Keep both free list and allocated list.

- wasteful, since one implies the other.
Related issue:
(both issues are solved with the same implementation)

How does the library build a free list (a dynamically created linked list)?

Alternative wording: How do we implement \texttt{malloc(freeNode, size)} inside the \texttt{malloc()} function?
Solution: We have plenty of memory space already... within the free chunks.

So embed the needed info within the free chunks.

Example: `malloc (16)`

Find free space of 24 bytes.
Place value 16 in size field
Place magic# in 2\textsuperscript{nd} int
Return addr of 3\textsuperscript{rd} int.
With `free(ptr)`

```
   4  4
   +---+
   |   |
   v   v
[ptr]
```

Check if

```
*(ptr - 4) == magic number
```

This is a very fast check to see if the program may have overwritten the header.

If header OK,

```
*(ptr - 8) gives size of chunk, so add it back into free list
```

and **COALESCE**.
To embed the free list into the free blocks, recall earlier diagram:

Where does the memory for free list nodes come from?
head of free list

assume program does
free(ptr);

pointers to allocated space
An idea of part of the `free()` code:

```c
if *(ptr-4) == magic #
    ptr-8 is addr1 of start of free space
    *(ptr-8)+8 is size1 of free space
    addr1 + size1 is addr2 of start of next chunk
```

To coalesce, where freed chunk is adjacent + before another free chunk:

Traverse free list
    if a next == addr2
    *(addr1+8) or *(ptr-4) = *(addr2+4)
    *(addr1) =
    next =
```