Be sure that a pointer passed to `realloc` came from a previous call of `malloc`, `calloc`, or `realloc`. If it didn't, calling `realloc` causes undefined behavior.

The C standard spells out a number of rules concerning the behavior of `realloc`:

- When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
- If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
- If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
- If `realloc` is called with 0 as its second argument, it frees the memory block.

The C standard stops short of specifying exactly how `realloc` works. Still, we expect it to be reasonably efficient. When asked to reduce the size of a memory block, `realloc` should shrink the block "in place," without moving the data stored in the block. By the same token, `realloc` should always attempt to expand a memory block without moving it. If it's unable to enlarge the block (because the bytes following the block are already in use for some other purpose), `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.

Once `realloc` has returned, be sure to update all pointers to the memory block, since it's possible that `realloc` has moved the block elsewhere.

17.4 Deallocating Storage

`malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap***. Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.

To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space. Consider the following example:

```
p = malloc(...);
q = malloc(...);
p = q;
```