```c
/*                                                                      tab:8
 *
 * mem220.c - a simple memory management package for ECE220
 *
 * "Copyright (c) 2003-2018 by Steven S. Lumetta."
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice and the following
 * two paragraphs appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE AUTHOR OR THE UNIVERSITY OF ILLINOIS BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
 * DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION,
 * EVEN IF THE AUTHOR AND/OR THE UNIVERSITY OF ILLINOIS HAS BEEN ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE AUTHOR AND THE UNIVERSITY OF ILLINOIS SPECIFICALLY DISCLAIM ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE
 * PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND NEITHER THE AUTHOR NOR
 * THE UNIVERSITY OF ILLINOIS HAS ANY OBLIGATION TO PROVIDE MAINTENANCE,
 * SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Author:          Steve Lumetta
 * Version:         3
 * Creation Date:   4 December 2003
 * Filename:        mem220.c
 * History:
 *      SL      1       4 December 2003
 *              First written.
 *      SL      2       28 March 2018
 *              Updated slightly for new numbering and standards.
 *      SL      3       2 April 2018
 *              Added braces and flipped operands to follow newer coding style.
 */


#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "mem220.h"


/*
   This memory manager allocates blocks in sizes of powers of two,
   allowing reasonably efficient reuse of freed blocks.  As with almost
   all memory managers, management information is held in a header
   preceding the region allocated to the caller.  For this implementation,
   we need only the block size in the header, which allows free to
   place the block into the correct bin.  We actually store the index
   of the bin in our array of bins, which is equivalent to the log_2 of
   the block size.
*/
```

```c
/*
   The memory block header structure, stored at the front of each block
   of memory.  It contains the size of the block and a pointer allowing
   us to chain free blocks together into a list.
*/
typedef struct mem_block_t mem_block_t;
struct mem_block_t {
    size_t       size;
    mem_block_t* next;
};


/* static functions (not visible outside of this file) */

/*
   mem220_init

   Initializes the memory management package.  Called before any blocks
   are allocated.
*/
static void mem220_init ();

/*
   log2_ceil

   Calculates the logarithm base 2 of a number, rounded up to the
   nearest integer.  Useful in determining what size block to allocate
   for a given memory request, as allocations are always made in powers
   of two.
*/
static int32_t log2_ceil (size_t value);


/* file scoped variables */

static uint8_t*     free_bytes;                        /* unallocated memory   */
static size_t       n_free_bytes;                      /* unallocated bytes    */
static mem_block_t* mem_bin[MEM220_MAX_ALLOC_LOG+1]; /* free block lists     */
static int32_t      init_done = 0;                     /* package initialized? */


/*
    mem220_allocate -- allocate a new block of n_bytes
    INPUTS -- minimum number of bytes in block available to caller
    OUTPUTS -- none
    RETURN VALUE -- pointer to new block (past header), or
                    NULL if no more memory available
*/

void*
mem220_allocate (size_t n_bytes)
{
    size_t       block_size; /* minimum size of allocated block           */
    int32_t      bin;        /* bin that holds blocks of appropriate size */
    mem_block_t* new_block;  /* the new block                             */

    /* On first call, initialize static data for the memory manager. */
    if (!init_done) {
        mem220_init ();
    }
    /* Add room for a header to find the necessary size. */
    block_size = n_bytes + sizeof (*new_block);

    /* Unsigned, so no need to check for requests < 0. */
    if (0 == n_bytes || MEM220_MAX_ALLOC < block_size) {
        return NULL;
    }

    /* Find the bin number. */
    bin = log2_ceil (block_size);

    /* Do we have a block sitting around? */
    if (NULL != mem_bin[bin]) {

        /*
           If so, remove the first one from the bin
           (a linked list of blocks).
        */
        new_block = mem_bin[bin];
        mem_bin[bin] = new_block->next;

    } else {
        /* No spare block, so try to allocate a new one. */

        /* Find the total block size. */
        n_bytes = (1UL << bin);

        /* Not enough space left in heap?  Return failure. */
        if (n_bytes > n_free_bytes) {
            return NULL;
        }

        /* Allocate the block from the front of the heap. */
        new_block = (mem_block_t*)free_bytes;
        free_bytes += n_bytes;
        n_free_bytes -= n_bytes;

        /* Mark the block's size in the header area. */
        new_block->size = n_bytes;
    }

    /* Return a pointer to the part AFTER the header. */
    return (new_block + 1);
}


/*
    mem220_allocate_and_zero -- allocate a new block of n_bytes and fill
                                it with zeroes
    INPUTS -- minimum number of bytes in block available to caller
    OUTPUTS -- none
    RETURN VALUE -- pointer to new block (past header), or
                    NULL if no more memory available
*/

void*
mem220_allocate_and_zero (size_t n_bytes)
{
    void* new_block;
```

```
    /* First allocate a block.  If the attempt fails, so does this
       function. */
    new_block = mem220_allocate (n_bytes);
    if (NULL == new_block) {
        return NULL;
    }

    /* Set the bytes to zero.  Note that the pointer returned to us
       points past the memory header, so we only zero the data to be
       used by the caller, not the memory management information. */
    memset (new_block, 0, n_bytes);

    /* Return the new block. */
    return new_block;
}


/*
    mem220_reallocate -- change the size of a block of memory, allocating
                         a new block if necessary
    INPUTS -- ptr_to_ptr, a pointer to the pointer to the old block
              n_bytes, the minimum number of bytes in reallocated block
                       available to caller
    OUTPUTS -- *ptr_to_ptr, a pointer to the new block (on success only)
    RETURN VALUE -- 0 for success, in which case *ptr_to_ptr may have changed
                    -1 for failure, in which case *ptr_to_ptr does not change
    SIDE EFFECTS -- if a new block is necessary, and is created successfully,
                    data from the old block are copied into it, and the
                    old block is freed
*/

int32_t
mem220_reallocate (void** ptr_to_ptr, size_t n_bytes)
{
    mem_block_t* old_block;  /* pointer to old block of data */
    mem_block_t* new_block;  /* pointer to reallocated block */

    /*
       Calling with ptr_to_ptr equal to NULL should lead to an
       assertion (deliberate crash), but we'll just return failure.
    */
    if (NULL == ptr_to_ptr) {
        return -1;
    }

    /* If the pointer is valid, read the old block pointer. */
    old_block = *ptr_to_ptr;

    /*
       If the new size (including the header) still fits in the
       current block, nothing need be done to succeed.  Note the
       method used to access the header, which sits before the pointer
       returned by the earlier allocation call.
    */
    if (NULL != old_block &&
        n_bytes + sizeof (*old_block) <= old_block[-1].size) {
        return 0;
    }
    /*
       Try to create a new block.  Return failure if necessary
       (without changing the old block pointer).
    */
    new_block = mem220_allocate (n_bytes);
    if (NULL == new_block) {
        return -1;
    }

    /*
       New block exists, so write it over the old pointer; we still
       have a copy in old_block for the rest of this function.
    */
    *ptr_to_ptr = new_block;

    /*
       The data block must have grown, so copy all old bytes if an old
       block existed, then free the old block.  Note that the header
       bytes are not included, since old_block points past them, and
       the new block has its own header.
    */
    if (NULL != old_block) {
        memcpy (new_block, old_block,
                old_block[-1].size - sizeof (*old_block));
        mem220_free (old_block);
    }

    /* All done.  Return success. */
    return 0;
}


/*
    mem220_free -- free a block of memory
    INPUTS -- a pointer to the old block
    OUTPUTS -- none
    RETURN VALUE -- none
    SIDE EFFECTS -- the block now belongs to the memory management package,
                    which may reuse it later
*/

void
mem220_free (void* ptr)
{
    mem_block_t* mem_block = ptr;  /* memory block pointer     */
    int32_t      bin;              /* bin number for old block */

    /* Check for free of NULL pointer.  Again, should probably have
       assertion rather than simple return. */
    if (NULL == ptr) {
        return;
    }

    /* Put the block into the appropriate bin. */
    bin = log2_ceil (mem_block[-1].size);
    mem_block[-1].next = mem_bin[bin];
    mem_bin[bin] = &mem_block[-1];
}
```

```
/*
    mem220_init -- initialize memory management data
    INPUTS -- none
    OUTPUTS -- none
    RETURN VALUE -- none
    SIDE EFFECTS -- initializes static data and sets up pointers to
                    unallocated region of memory (a simulated heap)
*/

static void
mem220_init ()
{
    /* All bins are empty (set pointers to NULL). */
    memset (mem_bin, 0, sizeof (mem_bin));

    /* Allocate a "heap" for us to manage. */
    n_free_bytes = 16 * MEM220_MAX_ALLOC;
    free_bytes = malloc (n_free_bytes);
    if (NULL == free_bytes) {
        perror ("initialize (malloc) mem220 package");
        exit (3);
    }

    /* Init has run; make a note of it. */
    init_done = 1;
}


/*
    log2_ceil -- calculate the logarithm base 2 of the value passed, rounded
                 up to the nearest integer
    INPUTS -- an unsigned value on which to operate
    OUTPUTS -- none
    RETURN VALUE -- ceil (log_2 (value)), as an integer, or
                    -1 if value == 0
*/
static int32_t
log2_ceil (size_t value)
{
    int32_t ret_val;

    /*
       If value is a power of 2, we start counting at -1, otherwise,
       we start counting at 0 (to round up).
    */
    if (0 == (value & (value - 1))) {
        ret_val = -1;
    } else {
        ret_val = 0;
    }

    /*
       Shift the value to the right until it disappears.  Counting with
       a loop in this manner is not the fastest possible method, but it
       is the simplest.
    */
    while (0 < value) {
        ret_val++;
        value >>= 1;
    }

    return ret_val;
}
```