alloc.c

```
#include "alloc.h"

#include <stdio.h>
#include <stdint.h>
#include <string.h> // memset
#include <stdlib.h> // atexit, [cm]alloc, free
#include <assert.h> // assert

#define MEMTEST

const unsigned char PADDING = 255;

static int registeredExit = 0;
static size_t allocCount = 0;
static size_t allocAmount = 0;
static size_t freeCount = 0;

typedef struct reserved_t
{
  unsigned char* base; // the base address of this reserved memory
  unsigned char* data; // where the data actually is
  unsigned char* dend; // where the data ends
  unsigned char* end; // the last piece of data in this bit of reserved memory
  size_t num; // how many elements is this memory for
  // n.b.: each individual element won't be padded,
  // so corruption could occur here
  size_t size; // what's the size of each element
  struct reserved_t* next;

  const char* file;
  size_t line;
} Reserved;

Reserved* root = NULL;

Reserved* last()
```

```
{
  Reserved* node = root;
  while (1)
  {
    if (!node->next)
      break;
    node = node->next;
  }
  return node;
}

void* mt_malloc_(const size_t sz,
                 const char* file, const size_t line)
{
#ifdef MEMTEST
  unsigned char* p = malloc(sz*2);
  if (p)
  {
    Reserved* r = (Reserved*)malloc(sizeof(Reserved));
    assert(r);
    r->base = p;
    r->data = (sz >> 1) + p;
    r->dend = r->data + sz;
    r->end = p + sz * 2;
    r->num = 1;
    r->size = sz;
    r->next = NULL;
    r->file = file;
    r->line = line;

    // set all the bytes except those in our data to be 0
    // this preserves the junk values we get
    // but allow us to test for under and overflows later
    memset(r->base, PADDING, r->data - r->base);
    memset(r->dend, PADDING, r->end - r->dend);

    r->next = root;
    root = r;

    if (!registeredExit)
      registeredExit = !atexit(mt_check);
    ++allocCount;
    allocAmount += sz;

    return r->data;
  }
  return NULL;
#else
  return malloc(sz);
#endif
}

void* mt_calloc_(const size_t n, const size_t sz,
                 const char* file, const size_t line)
{
#ifdef MEMTEST
  unsigned char* p = calloc(n, sz*2); // use calloc, as we want all zeroes
  if (p)
  {
    Reserved* r = (Reserved*)malloc(sizeof(Reserved));
    assert(r);
    r->base = p;
    r->data = ((intptr_t)(n*sz) >> 1) + p;
    r->dend = r->data + r->size * r->num;
    r->end = p + n*sz*2;
    r->num = n;
    r->size = sz;
    r->next = NULL;
    r->file = file;
    r->line = line;

    r->next = root;
    root = r;

    if (!registeredExit)
      registeredExit = !atexit(mt_check);
    ++allocCount;
    allocAmount += sz * n;

    return r->data;
  }
  return NULL;
#else
  return calloc(n, sz);
#endif
}

void underwrite(const Reserved* const node)
{
  size_t badBytes = 0;
  for (unsigned char* i = node->base; i < node->data; ++i)
  {
    if (*((unsigned char*)i) != PADDING)
      ++badBytes;
  }
  if (!badBytes)
    return;
  fprintf(stderr, "Underwrite detected:\n    "
          "From: %s:%zu, base: %p, data: %p, size: %zu, num: %zu\n",
          node->file, node->line,
          node->base, node->data, node->size, node->num);
  for (unsigned char* i = node->base; i < node->data; ++i)
  {
    if (*((unsigned char*)i) != PADDING)
    {
      fprintf(stderr, "\tByte %zu has value %x\n",
              (size_t)(i - node->data), *((unsigned char*)i));
    }
  }
}

void overwrite(const Reserved* const node)
{
  size_t badBytes = 0;
  for (unsigned char* i = node->dend; i < node->end; ++i)
  {
    if (*((unsigned char*)i) != PADDING)
      ++badBytes;
  }
  if (!badBytes)
    return;
  fprintf(stderr,
          "Overwrite detected:\n    "
          "From: %s:%zu, base: %p, data: %p, size: %zu, num: %zu\n",
          node->file, node->line,
          node->base, node->data, node->size, node->num);
  for (unsigned char* i = node->dend; i < node->end; ++i)
  {
    if (*((unsigned char*)i) != PADDING)
    {
      fprintf(stderr, "\tByte %zu has value %x\n",
              (size_t)(i - node->data), *((unsigned char*)i));
    }
  }
}

void mt_free(void* p)
{
```

```
#ifdef MEMTEST
  if (!p)
    return;

  Reserved* prev;
  Reserved* node = root;
  if (root && root->data == p)
  {
    prev = NULL;
  }
  else
  {
    while (node && node->data != p)
    {
      prev = node;
      node = node->next;
    }
  }

  // if it wasn't allocated with one of the mt_ functions, just free it normally
  if (!node)
  {
    free(p);
    return;
  }

  // check for any under or overwrites
  underwrite(node);
  overwrite(node);

  // free our base, we're done with the data
  free(node->base);
  if (prev)
    prev->next = node->next; // relink our linked list
  else
    root = node->next;
  ++freeCount;
  free(node); // cull the node
#else
  free(p);
#endif
}

void mt_check(void)
{
#ifndef MEMTEST
```

```
  return;
#endif

  if (allocCount)
    fprintf(stderr, "Made %zu allocations totalling %zu bytes\n",
           allocCount, allocAmount);
  if (freeCount)
    fprintf(stderr, "Made %zu frees\n", freeCount);

  size_t leaks = 0;
  size_t bytes = 0;
  while (root)
  {
    size_t l = root->num * root->size;
    bytes += l;
    ++leaks;
   fprintf(stderr, "Leaked from %s:%zu, %zu bytes of memory at %p (%p)\n",
            root->file, root->line, l, root->base, root->data);
    mt_free(root->data);
  }
  if (leaks)
    fprintf(stderr, "Found a total of %zu leaks, leaking %zu bytes\n",
            leaks, bytes);
}
```