stack.h (Page 1 of 1)

```
 1: /*--------------------------------------------------------------------*/
 2: /* stack.h (Version 3)                                                */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #ifndef STACK_INCLUDED
 7: #define STACK_INCLUDED
 8:
 9: /* A Stack_T object is a last-in-first-out collection of doubles. */
10:
11: typedef struct Stack *Stack_T;
12:
13: /*--------------------------------------------------------------------*/
14:
15: /* Return a new Stack_T object, or NULL if insufficient memory is
16:    available. */
17:
18: Stack_T Stack_new(void);
19:
20: /*--------------------------------------------------------------------*/
21:
22: /* Free oStack. */
23:
24: void Stack_free(Stack_T oStack);
25:
26: /*--------------------------------------------------------------------*/
27:
28: /* Push dItem onto oStack.  Return 1 (TRUE) if successful, or 0
29:    (FALSE) if insufficient memory is available. */
30:
31: int Stack_push(Stack_T oStack, double dItem);
32:
33: /*--------------------------------------------------------------------*/
34:
35: /* Pop and return the top item of oStack. */
36:
37: double Stack_pop(Stack_T oStack);
38:
39: /*--------------------------------------------------------------------*/
40:
41: /* Return 1 (TRUE) if oStack is empty, or 0 (FALSE) otherwise. */
42:
43: int Stack_isEmpty(Stack_T oStack);
44:
45: #endif
```

stack.c (Page 1 of 2)

```
 1: /*--------------------------------------------------------------------*/
 2: /* stack.c (Version 3)                                                */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include "stack.h"
 7: #include <stdlib.h>
 8: #include <assert.h>
 9:
10: /*--------------------------------------------------------------------*/
11:
12: /* A Stack object consists of an array of items, and related data. */
13: struct Stack
14: {
15:    /* The array in which items are stored. */
16:    double *pdArray;
17:
18:    /* The index one beyond the top element. */
19:    size_t uTop;
20:
21:    /* The number of elements in the array. */
22:    size_t uPhysLength;
23: };
24:
25: /*--------------------------------------------------------------------*/
26:
27: /* Double the physical length of oStack.  Return 1 (TRUE) if
28:    successful, or 0 (FALSE) if insufficient memory is available. */
29:
30: static int Stack_grow(Stack_T oStack)
31: {
32:    const size_t GROWTH_FACTOR = 2;
33:
34:    size_t uNewPhysLength;
35:    double *pdNewArray;
36:
37:    assert(oStack != NULL);
38:
39:    uNewPhysLength = GROWTH_FACTOR * oStack->uPhysLength;
40:    pdNewArray = (double*)
41:       realloc(oStack->pdArray, sizeof(double) * uNewPhysLength);
42:    if (pdNewArray == NULL)
43:       return 0;
44:    oStack->uPhysLength = uNewPhysLength;
45:    oStack->pdArray = pdNewArray;
46:
47:    return 1;
48: }
49:
50: /*--------------------------------------------------------------------*/
51:
52: Stack_T Stack_new(void)
53: {
54:    const size_t INITIAL_PHYS_LENGTH = 2;
55:
56:    Stack_T oStack;
57:
58:    oStack = (Stack_T)malloc(sizeof(struct Stack));
59:    if (oStack == NULL)
60:       return NULL;
61:
62:    oStack->pdArray =
63:       (double*)calloc(INITIAL_PHYS_LENGTH, sizeof(double));
```

stack.c (Page 2 of 2)

```
64:    if (oStack->pdArray == NULL)
65:    {
66:       free(oStack);
67:       return NULL;
68:    }
69:
70:    oStack->uTop = 0;
71:    oStack->uPhysLength = INITIAL_PHYS_LENGTH;
72:    return oStack;
73: }
74:
75: /*--------------------------------------------------------------------*/
76:
77: void Stack_free(Stack_T oStack)
78: {
79:    assert(oStack != NULL);
80:    free(oStack->pdArray);
81:    free(oStack);
82: }
83:
84: /*--------------------------------------------------------------------*/
85:
86: int Stack_push(Stack_T oStack, double dItem)
87: {
88:    assert(oStack != NULL);
89:    if (oStack->uTop == oStack->uPhysLength)
90:       if (! Stack_grow(oStack))
91:          return 0;
92:    (oStack->pdArray)[oStack->uTop] = dItem;
93:    (oStack->uTop)++;
94:    return 1;
95: }
96:
97: /*--------------------------------------------------------------------*/
98:
99: double Stack_pop(Stack_T oStack)
100: {
101:    assert(oStack != NULL);
102:    assert(oStack->uTop > 0);
103:    (oStack->uTop)--;
104:    return (oStack->pdArray)[oStack->uTop];
105: }
106:
107: /*--------------------------------------------------------------------*/
108:
109: int Stack_isEmpty(Stack_T oStack)
110: {
111:    assert(oStack != NULL);
112:    return oStack->uTop == 0;
113: }
```

teststack.c (Page 1 of 2)

```
 1: /*--------------------------------------------------------------------*/
 2: /* teststack.c (Version 3)                                            */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include "stack.h"
 7: #include <stdio.h>
 8: #include <stdlib.h>
 9:
10: /*--------------------------------------------------------------------*/
11:
12: /* Write an error message to stderr indicating that not enough memory
13:    is available.  Then exit with status EXIT_FAILURE. */
14:
15: static void handleMemoryError(void)
16: {
17:    fprintf(stderr, "Insufficient memory\n");
18:    exit(EXIT_FAILURE);
19: }
20:
21: /*--------------------------------------------------------------------*/
22:
23: /* Test the Stack ADT.  Return 0, or EXIT_FAILURE if not enough memory
24:    is available. */
25:
26: int main(void)
27: {
28:    Stack_T oStack1;
29:    Stack_T oStack2;
30:    int iSuccessful;
31:
32:    /* Use a Stack object referenced by oStack1. */
33:
34:    oStack1 = Stack_new();
35:    if (oStack1 == NULL) handleMemoryError();
36:
37:    iSuccessful = Stack_push(oStack1, 1.1);
38:    if (! iSuccessful) handleMemoryError();
39:
40:    iSuccessful = Stack_push(oStack1, 2.2);
41:    if (! iSuccessful) handleMemoryError();
42:
43:    iSuccessful = Stack_push(oStack1, 3.3);
44:    if (! iSuccessful) handleMemoryError();
45:
46:    while (! Stack_isEmpty(oStack1))
47:       printf("%g\n", Stack_pop(oStack1));
48:
49:    Stack_free(oStack1);
50:
51:    /* Use a Stack object referenced by oStack2. */
52:
53:    oStack2 = Stack_new();
54:    if (oStack2 == NULL) handleMemoryError();
55:
56:    iSuccessful = Stack_push(oStack2, 4.4);
57:    if (! iSuccessful) handleMemoryError();
58:
59:    iSuccessful = Stack_push(oStack2, 5.5);
60:    if (! iSuccessful) handleMemoryError();
61:
62:    iSuccessful = Stack_push(oStack2, 6.6);
63:    if (! iSuccessful) handleMemoryError();
```

teststack.c (Page 2 of 2)

```
64:
65:    while (! Stack_isEmpty(oStack2))
66:       printf("%g\n", Stack_pop(oStack2));
67:
68:    Stack_free(oStack2);
69:
70:    return 0;
71: }
72:
73: /*-------------------------------------------------------------------------*/
74:
75: /*
76:    Output:
77:    3.3
78:    2.2
79:    1.1
80:    6.6
81:    5.5
82:    4.4
83: */
```

stackao.h (Page 1 of 1)

```
 1: /*--------------------------------------------------------------------*/
 2: /* stackao.h                                                          */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #ifndef STACKAO_INCLUDED
 7: #define STACKAO_INCLUDED
 8:
 9: /* The Stack object is a last-in-first-out collection of doubles. */
10:
11: /*--------------------------------------------------------------------*/
12:
13: /* Initialize the Stack object.  Return 1 (TRUE) if successful, or
14:    0 (FALSE) if insufficient memory is available. */
15:
16: int Stack_init(void);
17:
18: /*--------------------------------------------------------------------*/
19:
20: /* Free the resources consumed by the Stack object, and uninitialize
21:    it. */
22:
23: void Stack_free(void);
24:
25: /*--------------------------------------------------------------------*/
26:
27: /* Push dItem onto the Stack object.  Return 1 (TRUE) if successful,
28:    or 0 (FALSE) if insufficient memory is available. */
29:
30: int Stack_push(double dItem);
31:
32: /*--------------------------------------------------------------------*/
33:
34: /* Pop and return the top item of the Stack object. */
35:
36: double Stack_pop(void);
37:
38: /*--------------------------------------------------------------------*/
39:
40: /* Return 1 (TRUE) if the Stack object is empty, or 0 (FALSE)
41:    otherwise. */
42:
43: int Stack_isEmpty(void);
44:
45: #endif
```

stackao.c (Page 1 of 2)

```
 1: /*-------------------------------------------------------------------------*/
 2: /* stackao.c                                                               */
 3: /* Author: Bob Dondero                                                     */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #include "stackao.h"
 7: #include <stdlib.h>
 8: #include <assert.h>
 9:
10: /*-------------------------------------------------------------------------*/
11:
12: /* In lieu of a boolean data type. */
13: enum {FALSE, TRUE};
14:
15: /*-------------------------------------------------------------------------*/
16:
17: /* The state of the Stack object. */
18:
19: /* The array in which items are stored. */
20: static double *pdArray;
21:
22: /* The index one beyond the top element. */
23: static size_t uTop;
24:
25: /* The number of elements in the array. */
26: static size_t uPhysLength;
27:
28: /* Is the Stack initialized? */
29: static int iInitialized = FALSE;
30:
31: /*-------------------------------------------------------------------------*/
32:
33: /* Increase the physical length of the Stack object.  Return 1 (TRUE) if
34:    successful, or 0 (FALSE) if insufficient memory is available. */
35:
36: static int Stack_grow(void)
37: {
38:    const size_t GROWTH_FACTOR = 2;
39:
40:    size_t uNewPhysLength;
41:    double *pdNewArray;
42:
43:    assert(iInitialized);
44:
45:    uNewPhysLength = GROWTH_FACTOR * uPhysLength;
46:    pdNewArray = (double*)
47:       realloc(pdArray, sizeof(double) * uNewPhysLength);
48:    if (pdNewArray == NULL)
49:       return 0;
50:
51:    uPhysLength = uNewPhysLength;
52:    pdArray = pdNewArray;
53:
54:    return 1;
55: }
56:
57: /*-------------------------------------------------------------------------*/
58:
59: int Stack_init(void)
60: {
61:    const size_t INITIAL_PHYS_LENGTH = 2;
62:
63:    assert(! iInitialized);
```

```
 64:
 65:    pdArray = (double*)calloc(INITIAL_PHYS_LENGTH, sizeof(double));
 66:    if (pdArray == NULL)
 67:       return 0;
 68:
 69:    uTop = 0;
 70:    uPhysLength = INITIAL_PHYS_LENGTH;
 71:    iInitialized = TRUE;
 72:    return 1;
 73: }
 74:
 75: /*--------------------------------------------------------------------*/
 76:
 77: void Stack_free(void)
 78: {
 79:    assert(iInitialized);
 80:    free(pdArray);
 81:    iInitialized = FALSE;
 82: }
 83:
 84: /*--------------------------------------------------------------------*/
 85:
 86: int Stack_push(double dItem)
 87: {
 88:    assert(iInitialized);
 89:    if (uTop == uPhysLength)
 90:       if (! Stack_grow())
 91:          return 0;
 92:
 93:    pdArray[uTop] = dItem;
 94:    uTop++;
 95:    return 1;
 96: }
 97:
 98: /*--------------------------------------------------------------------*/
 99:
100: double Stack_pop(void)
101: {
102:    assert(iInitialized);
103:    assert(uTop > 0);
104:    uTop--;
105:    return pdArray[uTop];
106: }
107:
108: /*--------------------------------------------------------------------*/
109:
110: int Stack_isEmpty(void)
111: {
112:    assert(iInitialized);
113:    return uTop == 0;
114: }
```

teststackao.c (Page 1 of 2)

```
 1: /*--------------------------------------------------------------------*/
 2: /* teststackao.c                                                      */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include "stackao.h"
 7: #include <stdio.h>
 8: #include <stdlib.h>
 9:
10: /*--------------------------------------------------------------------*/
11:
12: /* Write an error message to stderr indicating that not enough memory
13:    is available.  Then exit with status EXIT_FAILURE. */
14:
15: static void handleMemoryError(void)
16: {
17:    fprintf(stderr, "Insufficient memory\n");
18:    exit(EXIT_FAILURE);
19: }
20:
21: /*--------------------------------------------------------------------*/
22:
23: /* Test the Stack abstract object.  Return 0, or EXIT_FAILURE if not
24:    enough memory is available. */
25:
26: int main(void)
27: {
28:    int iSuccessful;
29:
30:    iSuccessful = Stack_init();
31:    if (! iSuccessful) handleMemoryError();
32:
33:    iSuccessful = Stack_push(1.1);
34:    if (! iSuccessful) handleMemoryError();
35:
36:    iSuccessful = Stack_push(2.2);
37:    if (! iSuccessful) handleMemoryError();
38:
39:    iSuccessful = Stack_push(3.3);
40:    if (! iSuccessful) handleMemoryError();
41:
42:    while (! Stack_isEmpty())
43:       printf("%g\n", Stack_pop());
44:
45:    Stack_free();
46:
47:    iSuccessful = Stack_init();
48:    if (! iSuccessful) handleMemoryError();
49:
50:    iSuccessful = Stack_push(4.4);
51:    if (! iSuccessful) handleMemoryError();
52:
53:    iSuccessful = Stack_push(5.5);
54:    if (! iSuccessful) handleMemoryError();
55:
56:    iSuccessful = Stack_push(6.6);
57:    if (! iSuccessful) handleMemoryError();
58:
59:    while (! Stack_isEmpty())
60:       printf("%g\n", Stack_pop());
61:
62:    Stack_free();
63:
```

teststackao.c (Page 2 of 2)

```
64:     return 0;
65: }
66:
67: /*-------------------------------------------------------------------------*/
68:
69: /*
70:     Output:
71:     3.3
72:     2.2
73:     1.1
74:     6.6
75:     5.5
76:     4.4
77: */
```

heapmgr.h (Page 1 of 1)

```
 1: /*--------------------------------------------------------------------*/
 2: /* heapmgr.h                                                          */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #ifndef HEAPMGR_INCLUDED
 7: #define HEAPMGR_INCLUDED
 8:
 9: #include <stddef.h>
10:
11: /*--------------------------------------------------------------------*/
12:
13: /* Allocate and return the address of a chunk of memory that is large
14:    enough to hold an object whose size is uBytes bytes. The chunk is
15:    guaranteed to be properly aligned for data of any type. Return NULL
16:    if uBytes is 0 or the request cannot be satisfied. The chunk is
17:    uninitialized. */
18:
19: void *HeapMgr_malloc(size_t uBytes);
20:
21: /*--------------------------------------------------------------------*/
22:
23: /* Free the chunk of memory pointed to by pv. pv must point to a chunk
24:    that was allocated by HeapMgr_malloc(). Do nothing if pv is NULL. */
25:
26: void HeapMgr_free(void *pv);
27:
28: #endif
```

```
 1: /*-------------------------------------------------------------------------*/
 2: /* testheapmgr.c                                                           */
 3: /* Author: Bob Dondero                                                     */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #define _GNU_SOURCE
 7:
 8: #include "heapmgr.h"
 9: #include <stdio.h>
10: #include <stdlib.h>
11: #include <time.h>
12: #include <string.h>
13: #include <assert.h>
14: #include <unistd.h>
15:
16: #ifndef S_SPLINT_S
17: #include <sys/resource.h>
18: #endif
19:
20: /* In lieu of a boolean data type. */
21: enum {FALSE, TRUE};
22:
23: /*-------------------------------------------------------------------------*/
24:
25: /* These arrays are too big for the stack section, so store
26:    them in the bss section. */
27:
28: /* The maximum allowable number of calls of HeapMgr_malloc(). */
29: enum {MAX_CALLS = 1000000};
30:
31: /* Memory chunks allocated by HeapMgr_malloc(). */
32: static char *apcChunks[MAX_CALLS];
33:
34: /* Randomly generated chunk sizes.  */
35: static int aiSizes[MAX_CALLS];
36:
37: /*-------------------------------------------------------------------------*/
38:
39: /* Function declarations. */
40:
41: /* Get command-line arguments *piTestNum, *piCount, and *piSize,
42:    from argument vector argv. argc is the number of used elements
43:    in argv. Exit if any of the arguments is invalid.  */
44: static void getArgs(int argc, char *argv[],
45:    int *piTestNum, int *piCount, int *piSize);
46:
47: /* Set the process's "CPU time" resource limit. After the CPU
48:    time limit expires, the OS will send a SIGKILL signal to the
49:    process. */
50: static void setCpuTimeLimit(void);
51:
52: /* Allocate and free iCount memory chunks, each of size iSize, in
53:    last-in-first-out order. */
54: static void testLifoFixed(int iCount, int iSize);
55:
56: /* Allocate and free iCount memory chunks, each of size iSize, in
57:    first-in-first-out order. */
58: static void testFifoFixed(int iCount, int iSize);
59:
60: /* Allocate and free iCount memory chunks, each of some random size
61:    less than iSize, in last-in-first-out order. */
62: static void testLifoRandom(int iCount, int iSize);
63:
```

```
 64: /* Allocate and free iCount memory chunks, each of some random size
 65:    less than iSize, in first-in-first-out order. */
 66: static void testFifoRandom(int iCount, int iSize);
 67:
 68: /* Allocate and free iCount memory chunks, each of size iSize, in
 69:    a random order. */
 70: static void testRandomFixed(int iCount, int iSize);
 71:
 72: /* Allocate and free iCount memory chunks, each of some random size
 73:    less than iSize, in a random order. */
 74: static void testRandomRandom(int iCount, int iSize);
 75:
 76: /* Allocate and free iCount memory chunks, each of some size less
 77:    than iSize, in the worst possible order for a HeapMgr that is
 78:    implemented using a single linked list. */
 79: static void testWorst(int iCount, int iSize);
 80:
 81: /*--------------------------------------------------------------------*/
 82:
 83: /* apcTestName is an array containing the names of the tests. */
 84:
 85: static char *apcTestName[] =
 86: {
 87:    "LifoFixed", "FifoFixed", "LifoRandom", "FifoRandom",
 88:    "RandomFixed", "RandomRandom", "Worst"
 89: };
 90:
 91: /*--------------------------------------------------------------------*/
 92:
 93:
 94: /* An object of type TestFunction is a pointer to a function that
 95:    accepts two ints and returns nothing. */
 96:
 97: typedef void (*TestFunction)(int, int);
 98:
 99: /* apfTestFunction is an array containing pointers to the test
100:    functions. Each pointer corresponds, by position, to a test name
101:    in apcTestName. */
102:
103: static TestFunction apfTestFunction[] =
104: {
105:    testLifoFixed, testFifoFixed, testLifoRandom, testFifoRandom,
106:    testRandomFixed, testRandomRandom, testWorst
107: };
108:
109: /*--------------------------------------------------------------------*/
110:
111: /* Test the HeapMgr_malloc() and HeapMgr_free() functions.
112:
113:    As always, argc is the command-line argument count.
114:
115:    argv[1] indicates which test to run:
116:       LifoFixed: LIFO with fixed size chunks,
117:       FifoFixed: FIFO with fixed size chunks,
118:       LifoRandom: LIFO with random size chunks,
119:       FifoRandom: FIFO with random size chunks,
120:       RandomFixed: random order with fixed size chunks,
121:       RandomRandom: random order with random size chunks,
122:       Worst: worst case for the doubly linked list implementation.
123:
124:    argv[2] is the number of calls of HeapMgr_malloc() and HeapMgr_free()
125:    to execute. argv[2] cannot be greater than MAX_CALLS.
126:
```

```
127:    argv[3] is the (maximum) size of each memory chunk.
128:
129:    If the NDEBUG macro is not defined, then initialize and check
130:    the contents of each memory chunk.
131:
132:    At the end of the process, write the heap memory and CPU time
133:    consumed to stdout, and return 0. */
134:
135: int main(int argc, char *argv[])
136: {
137:    int iTestNum = 0;
138:    int iCount = 0;
139:    int iSize = 0;
140:    clock_t iInitialClock;
141:    clock_t iFinalClock;
142:    char *pcInitialBreak;
143:    char *pcFinalBreak;
144:    unsigned int uiMemoryConsumed;
145:    double dTimeConsumed;
146:
147:    /* Get the command-line arguments. */
148:    getArgs(argc, argv, &iTestNum, &iCount, &iSize);
149:
150:    /* Start printing the results. */
151:    printf("%16s %12s %7d %6d ", argv[0], argv[1], iCount, iSize);
152:    fflush(stdout);
153:
154:    /* Save the initial clock and program break. */
155:    iInitialClock = clock();
156:    pcInitialBreak = sbrk(0);
157:
158:    /* Set the process's CPU time limit. */
159:    setCpuTimeLimit();
160:
161:    /* Call the specified test function. */
162:    (*(apfTestFunction[iTestNum]))(iCount, iSize);
163:
164:    /* Save the final clock and program break. */
165:    pcFinalBreak = sbrk(0);
166:    iFinalClock = clock();
167:
168:    /* Use the initial and final clocks and program breaks to compute
169:       CPU time and heap memory consumed. */
170:    uiMemoryConsumed = (unsigned int)(pcFinalBreak - pcInitialBreak);
171:    dTimeConsumed =
172:       ((double)(iFinalClock - iInitialClock)) / CLOCKS_PER_SEC;
173:
174:    /* Finish printing the results. */
175:    printf("%6.2f %10u\n", dTimeConsumed, uiMemoryConsumed);
176:    return 0;
177: }
178:
179: /*--------------------------------------------------------------------*/
180:
181: /* Get command-line arguments *piTestNum, *piCount, and *piSize,
182:    from argument vector argv. argc is the number of used elements
183:    in argv. Exit if any of the arguments is invalid.  */
184:
185: static void getArgs(int argc, char *argv[],
186:    int *piTestNum, int *piCount, int *piSize)
187: {
188:    int i;
189:    int iTestCount;
```

```
190:
191:    assert(argv != NULL);
192:    assert(piTestNum != NULL);
193:    assert(piCount != NULL);
194:    assert(piSize != NULL);
195:
196:    if (argc != 4)
197:    {
198:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
199:       exit(EXIT_FAILURE);
200:    }
201:
202:    /* Get the test number. */
203:    iTestCount = (int)(sizeof(apcTestName) / sizeof(apcTestName[0]));
204:    for (i = 0; i < iTestCount; i++)
205:       if (strcmp(argv[1], apcTestName[i]) == 0)
206:       {
207:          *piTestNum = i;
208:          break;
209:       }
210:    if (i == iTestCount)
211:    {
212:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
213:       fprintf(stderr, "Valid testnames:\n");
214:       for (i = 0; i < iTestCount; i++)
215:          fprintf(stderr, " %s", apcTestName[i]);
216:       fprintf(stderr, "\n");
217:       exit(EXIT_FAILURE);
218:    }
219:
220:    /* Get the count. */
221:    if (sscanf(argv[2], "%d", piCount) != 1)
222:    {
223:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
224:       fprintf(stderr, "Count must be numeric\n");
225:       exit(EXIT_FAILURE);
226:    }
227:    if (*piCount <= 0)
228:    {
229:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
230:       fprintf(stderr, "Count must be positive\n");
231:       exit(EXIT_FAILURE);
232:    }
233:    if (*piCount > MAX_CALLS)
234:    {
235:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
236:       fprintf(stderr, "Count cannot be greater than %d\n", MAX_CALLS);
237:       exit(EXIT_FAILURE);
238:    }
239:
240:    /* Get the size. */
241:    if (sscanf(argv[3], "%d", piSize) != 1)
242:    {
243:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
244:       fprintf(stderr, "Size must be numeric\n");
245:       exit(EXIT_FAILURE);
246:    }
247:    if (*piSize <= 0)
248:    {
249:       fprintf(stderr, "Usage: %s testname count size\n", argv[0]);
250:       fprintf(stderr, "Size must be positive\n");
251:       exit(EXIT_FAILURE);
252:    }
```

```
253: }
254:
255: /*-------------------------------------------------------------------*/
256:
257: #ifndef S_SPLINT_S
258: /* Set the process's "CPU time" resource limit. After the CPU
259:    time limit expires, the OS will send a SIGKILL signal to the
260:    process. */
261:
262: static void setCpuTimeLimit(void)
263: {
264:    enum {CPU_TIME_LIMIT_IN_SECONDS = 300};
265:    struct rlimit sRlimit;
266:    sRlimit.rlim_cur = CPU_TIME_LIMIT_IN_SECONDS;
267:    sRlimit.rlim_max = CPU_TIME_LIMIT_IN_SECONDS;
268:    setrlimit(RLIMIT_CPU, &sRlimit);
269: }
270: #endif
271:
272: /*-------------------------------------------------------------------*/
273:
274: #ifndef NDEBUG
275:
276: #define ASSURE(i) assure(i, __LINE__)
277:
278: /* If !iSuccessful, print an error message indicating that the test
279:    at line iLineNum failed. */
280:
281: static void assure(int iSuccessful, int iLineNum)
282: {
283:    if (! iSuccessful)
284:       fprintf(stderr, "Test at line %d failed.\n", iLineNum);
285: }
286:
287: #endif
288:
289: /*-------------------------------------------------------------------*/
290:
291: /* Allocate and free iCount memory chunks, each of size iSize, in
292:    last-in-first-out order. */
293:
294: static void testLifoFixed(int iCount, int iSize)
295: {
296:    int i;
297:
298:    /* Call HeapMgr_malloc() repeatedly to fill apcChunks. */
299:    for (i = 0; i < iCount; i++)
300:    {
301:       apcChunks[i] = (char*)HeapMgr_malloc((size_t)iSize);
302:       if (apcChunks[i] == NULL)
303:       {
304:          printf("Malloc returned NULL.\n");
305:          exit(0);
306:       }
307:
308:       #ifndef NDEBUG
309:       {
310:          /* Fill the newly allocated chunk with some character.
311:             The character is derived from the last digit of i.
312:             So later, given i, we can check to make sure that
313:             the contents haven't been corrupted. */
314:          int iCol;
315:          char c = (char)((i % 10) + '0');
```

```
316:          for (iCol = 0; iCol < iSize; iCol++)
317:             apcChunks[i][iCol] = c;
318:       }
319:       #endif
320:    }
321:
322:    /* Call HeapMgr_free() repeatedly to free the chunks in
323:       LIFO order. */
324:    for (i = iCount - 1; i >= 0; i--)
325:    {
326:       #ifndef NDEBUG
327:       {
328:          /* Check the chunk that is about to be freed to make sure
329:             that its contents haven't been corrupted. */
330:          int iCol;
331:          char c = (char)((i % 10) + '0');
332:          for (iCol = 0; iCol < iSize; iCol++)
333:             ASSURE(apcChunks[i][iCol] == c);
334:       }
335:       #endif
336:
337:       HeapMgr_free(apcChunks[i]);
338:    }
339: }
340:
341: /*--------------------------------------------------------------------*/
342:
343: /* Allocate and free iCount memory chunks, each of size iSize, in
344:    first-in-first-out order. */
345:
346: static void testFifoFixed(int iCount, int iSize)
347: {
348:    int i;
349:
350:    /* Call HeapMgr_malloc() repeatedly to fill apcChunks. */
351:    for (i = 0; i < iCount; i++)
352:    {
353:       apcChunks[i] = (char*)HeapMgr_malloc((size_t)iSize);
354:       if (apcChunks[i] == NULL)
355:       {
356:          printf("Malloc returned NULL.\n");
357:          exit(0);
358:       }
359:
360:       #ifndef NDEBUG
361:       {
362:          /* Fill the newly allocated chunk with some character.
363:             The character is derived from the last digit of i.
364:             So later, given i, we can check to make sure that
365:             the contents haven't been corrupted. */
366:          int iCol;
367:          char c = (char)((i % 10) + '0');
368:          for (iCol = 0; iCol < iSize; iCol++)
369:             apcChunks[i][iCol] = c;
370:       }
371:       #endif
372:    }
373:
374:    /* Call HeapMgr_free() repeatedly to free the chunks in
375:       FIFO order. */
376:    for (i = 0; i < iCount; i++)
377:    {
378:       #ifndef NDEBUG
```

```
379:       {
380:          /* Check the chunk that is about to be freed to make sure
381:             that its contents haven't been corrupted. */
382:          int iCol;
383:          char c = (char)((i % 10) + '0');
384:          for (iCol = 0; iCol < iSize; iCol++)
385:             ASSURE(apcChunks[i][iCol] == c);
386:       }
387:       #endif
388:
389:       HeapMgr_free(apcChunks[i]);
390:    }
391: }
392:
393: /*--------------------------------------------------------------------*/
394:
395: /* Allocate and free iCount memory chunks, each of some random size
396:    less than iSize, in last-in-first-out order. */
397:
398: static void testLifoRandom(int iCount, int iSize)
399: {
400:    int i;
401:
402:    /* Fill aiSizes, an array of random integers in the range 1 to
403:       iSize. */
404:    for (i = 0; i < iCount; i++)
405:       aiSizes[i] = (rand() % iSize) + 1;
406:
407:    /* Call HeapMgr_malloc() repeatedly to fill apcChunks. */
408:    for (i = 0; i < iCount; i++)
409:    {
410:       apcChunks[i] = (char*)HeapMgr_malloc((size_t)aiSizes[i]);
411:       if (apcChunks[i] == NULL)
412:       {
413:          printf("Malloc returned NULL.\n");
414:          exit(0);
415:       }
416:
417:       #ifndef NDEBUG
418:       {
419:          /* Fill the newly allocated chunk with some character.
420:             The character is derived from the last digit of i.
421:             So later, given i, we can check to make sure that
422:             the contents haven't been corrupted. */
423:          int iCol;
424:          char c = (char)((i % 10) + '0');
425:          for (iCol = 0; iCol < aiSizes[i]; iCol++)
426:             apcChunks[i][iCol] = c;
427:       }
428:       #endif
429:    }
430:
431:    /* Call HeapMgr_free() repeatedly to free the chunks in
432:       LIFO order. */
433:    for (i = iCount - 1; i >= 0; i--)
434:    {
435:       #ifndef NDEBUG
436:       {
437:          /* Check the chunk that is about to be freed to make sure
438:             that its contents haven't been corrupted. */
439:          int iCol;
440:          char c = (char)((i % 10) + '0');
441:          for (iCol = 0; iCol < aiSizes[i]; iCol++)
```

```
442:             ASSURE(apcChunks[i][iCol] == c);
443:       }
444:       #endif
445:
446:       HeapMgr_free(apcChunks[i]);
447:    }
448: }
449:
450: /*--------------------------------------------------------------------*/
451:
452: /* Allocate and free iCount memory chunks, each of some random size
453:    less than iSize, in first-in-first-out order. */
454:
455: static void testFifoRandom(int iCount, int iSize)
456: {
457:    int i;
458:
459:    /* Fill aiSizes, an array of random integers in the range 1 to
460:       iSize. */
461:    for (i = 0; i < iCount; i++)
462:       aiSizes[i] = (rand() % iSize) + 1;
463:
464:    /* Call HeapMgr_malloc() repeatedly to fill apcChunks. */
465:    for (i = 0; i < iCount; i++)
466:    {
467:       apcChunks[i] = (char*)HeapMgr_malloc((size_t)aiSizes[i]);
468:       if (apcChunks[i] == NULL)
469:       {
470:          printf("Malloc returned NULL.\n");
471:          exit(0);
472:       }
473:
474:       #ifndef NDEBUG
475:       {
476:          /* Fill the newly allocated chunk with some character.
477:             The character is derived from the last digit of i.
478:             So later, given i, we can check to make sure that
479:             the contents haven't been corrupted. */
480:          int iCol;
481:          char c = (char)((i % 10) + '0');
482:          for (iCol = 0; iCol < aiSizes[i]; iCol++)
483:             apcChunks[i][iCol] = c;
484:       }
485:       #endif
486:    }
487:
488:    /* Call HeapMgr_free() repeatedly to free the chunks in
489:       FIFO order. */
490:    for (i = 0; i < iCount; i++)
491:    {
492:       #ifndef NDEBUG
493:       {
494:          /* Check the chunk that is about to be freed to make sure
495:             that its contents haven't been corrupted. */
496:          int iCol;
497:          char c = (char)((i % 10) + '0');
498:          for (iCol = 0; iCol < aiSizes[i]; iCol++)
499:             ASSURE(apcChunks[i][iCol] == c);
500:       }
501:       #endif
502:
503:       HeapMgr_free(apcChunks[i]);
504:    }
```

```
505: }
506:
507: /*--------------------------------------------------------------------*/
508:
509: /* Allocate and free iCount memory chunks, each of size iSize, in
510:    a random order. */
511:
512: static void testRandomFixed(int iCount, int iSize)
513: {
514:    int i;
515:    int iRand;
516:    int iLogicalArraySize;
517:
518:    iLogicalArraySize = (iCount / 3) + 1;
519:
520:    i = 0;
521:
522:    /* Call HeapMgr_malloc() and HeapMgr_free() in a randomly
523:       interleaved manner. */
524:    while (i < iCount)
525:    {
526:       /* Assign some random integer to iRand. */
527:       iRand = rand() % iLogicalArraySize;
528:
529:       if (apcChunks[iRand] == NULL)
530:       {
531:          apcChunks[iRand] = (char*)HeapMgr_malloc((size_t)iSize);
532:          if (apcChunks[iRand] == NULL)
533:          {
534:             printf("Malloc returned NULL.\n");
535:             exit(0);
536:          }
537:
538:          #ifndef NDEBUG
539:          {
540:             /* Fill the newly allocated chunk with some character.
541:                The character is derived from the last digit of iRand.
542:                So later, given iRand, we can check to make sure that
543:                the contents haven't been corrupted. */
544:             int iCol;
545:             char c = (char)((iRand % 10) + '0');
546:             for (iCol = 0; iCol < iSize; iCol++)
547:                apcChunks[iRand][iCol] = c;
548:          }
549:          #endif
550:
551:          i++;
552:       }
553:
554:       /* Assign some random integer to iRand. */
555:       iRand = rand() % iLogicalArraySize;
556:
557:       /* If apcChunks[iRand] contains a chunk, free it and set
558:          apcChunks[iRand] to NULL. */
559:       if (apcChunks[iRand] != NULL)
560:       {
561:          #ifndef NDEBUG
562:          {
563:             /* Check the chunk that is about to be freed to make sure
564:                that its contents haven't been corrupted. */
565:             int iCol;
566:             char c = (char)((iRand % 10) + '0');
567:             for (iCol = 0; iCol < iSize; iCol++)
```

```
  568:                ASSURE(apcChunks[iRand][iCol] == c);
  569:             }
  570:             #endif
  571:
  572:             HeapMgr_free(apcChunks[iRand]);
  573:             apcChunks[iRand] = NULL;
  574:          }
  575:       }
  576:
  577:       /* Free the rest of the chunks. */
  578:       for (i = 0; i < iLogicalArraySize; i++)
  579:       {
  580:          if (apcChunks[i] != NULL)
  581:          {
  582:             #ifndef NDEBUG
  583:             {
  584:                /* Check the chunk that is about to be freed to make sure
  585:                   that its contents haven't been corrupted. */
  586:                int iCol;
  587:                char c = (char)((i % 10) + '0');
  588:                for (iCol = 0; iCol < iSize; iCol++)
  589:                   ASSURE(apcChunks[i][iCol] == c);
  590:             }
  591:             #endif
  592:
  593:             HeapMgr_free(apcChunks[i]);
  594:             apcChunks[i] = NULL;
  595:          }
  596:       }
  597: }
  598:
  599: /*--------------------------------------------------------------------*/
  600:
  601: /* Allocate and free iCount memory chunks, each of some random size
  602:    less than iSize, in a random order. */
  603:
  604: static void testRandomRandom(int iCount, int iSize)
  605: {
  606:    int i;
  607:    int iRand;
  608:    int iLogicalArraySize;
  609:
  610:    iLogicalArraySize = (iCount / 3) + 1;
  611:
  612:    /* Fill aiSizes, an array of random integers in the range 1
  613:       to iSize. */
  614:    for (i = 0; i < iLogicalArraySize; i++)
  615:       aiSizes[i] = (rand() % iSize) + 1;
  616:
  617:    i = 0;
  618:
  619:    /* Call HeapMgr_malloc() and HeapMgr_free() in a randomly
  620:       interleaved manner. */
  621:    while (i < iCount)
  622:    {
  623:       /* Assign some random integer to iRand. */
  624:       iRand = rand() % iLogicalArraySize;
  625:
  626:       if (apcChunks[iRand] == NULL)
  627:       {
  628:          apcChunks[iRand] = (char*)HeapMgr_malloc((size_t)aiSizes[iRand]
);
  629:          if (apcChunks[iRand] == NULL)
```

```
630:          {
631:             printf("Malloc returned NULL.\n");
632:             exit(0);
633:          }
634:
635:          #ifndef NDEBUG
636:          {
637:             /* Fill the newly allocated chunk with some character.
638:                The character is derived from the last digit of iRand.
639:                So later, given iRand, we can check to make sure that
640:                the contents haven't been corrupted. */
641:             int iCol;
642:             char c = (char)((iRand % 10) + '0');
643:             for (iCol = 0; iCol < aiSizes[iRand]; iCol++)
644:                apcChunks[iRand][iCol] = c;
645:          }
646:          #endif
647:
648:          i++;
649:       }
650:
651:       /* Assign some random integer to iRand. */
652:       iRand = rand() % iLogicalArraySize;
653:
654:       /* If apcChunks[iRand] contains a chunk, free it and set
655:          apcChunks[iRand] to NULL. */
656:       if (apcChunks[iRand] != NULL)
657:       {
658:          #ifndef NDEBUG
659:          {
660:             /* Check the chunk that is about to be freed to make sure
661:                that its contents haven't been corrupted. */
662:             int iCol;
663:             char c = (char)((iRand % 10) + '0');
664:             for (iCol = 0; iCol < aiSizes[iRand]; iCol++)
665:                ASSURE(apcChunks[iRand][iCol] == c);
666:          }
667:          #endif
668:
669:          HeapMgr_free(apcChunks[iRand]);
670:          apcChunks[iRand] = NULL;
671:       }
672:    }
673:
674:    /* Free the rest of the chunks. */
675:    for (i = 0; i < iLogicalArraySize; i++)
676:    {
677:       if (apcChunks[i] != NULL)
678:       {
679:          #ifndef NDEBUG
680:          {
681:             /* Check the chunk that is about to be freed to make sure
682:                that its contents haven't been corrupted. */
683:             int iCol;
684:             char c = (char)((i % 10) + '0');
685:             for (iCol = 0; iCol < aiSizes[i]; iCol++)
686:                ASSURE(apcChunks[i][iCol] == c);
687:          }
688:          #endif
689:
690:          HeapMgr_free(apcChunks[i]);
691:          apcChunks[i] = NULL;
692:       }
```

```
693:    }
694: }
695:
696: /*--------------------------------------------------------------------*/
697:
698: /* Allocate and free iCount memory chunks, each of some size less
699:    than iSize, in the worst possible order for a HeapMgr that is
700:    implemented using a single linked list. */
701:
702: static void testWorst(int iCount, int iSize)
703: {
704:    int i;
705:    int iChunkSize;
706:
707:    /* Make sure iCount is even. */
708:    if (iCount % 2 != 0)
709:       iCount++;
710:
711:    /* Fill the array with chunks of increasing size, each separated by
712:       a small dummy chunk. */
713:    i = 0;
714:    while (i < iCount)
715:    {
716:       iChunkSize =
717:          (int)(((double)i * ((double)iSize / (double)iCount)) + 1.0);
718:       apcChunks[i] = HeapMgr_malloc((size_t)iChunkSize);
719:       if ((i != 0) && (apcChunks[i] == NULL))
720:       {
721:          printf("Malloc returned NULL.\n");
722:          exit(0);
723:       }
724:
725:       #ifndef NDEBUG
726:       {
727:          /* Fill the newly allocated chunk with some character.
728:             The character is derived from the last digit of i.
729:             So later, given i, we can check to make sure that
730:             the contents haven't been corrupted. */
731:          int iCol;
732:          char c = (char)((i % 10) + '0');
733:          for (iCol = 0; iCol < iChunkSize; iCol++)
734:             apcChunks[i][iCol] = c;
735:       }
736:       #endif
737:       i++;
738:       apcChunks[i] = HeapMgr_malloc((size_t)1);
739:       if (apcChunks[i] == NULL)
740:       {
741:          printf("Malloc returned NULL.\n");
742:          exit(0);
743:       }
744:       i++;
745:    }
746:
747:    /* Free the non-dummy chunks in reverse order.  Thus a HeapMgr
748:       implementation that uses a single linked list will be in a
749:       worst-case state:  the list will contain chunks in increasing
750:       order by size. */
751:    i = iCount;
752:    while (i >= 2)
753:    {
754:       i--;
755:       i--;
```

```
   #ifndef NDEBUG
   {
      /* Check the chunk that is about to be freed to make sure
         that its contents haven't been corrupted. */
      int iCol;
      char c = (char)((i % 10) + '0');
      iChunkSize =
         (int)(((double)i * ((double)iSize / (double)iCount)) + 1.0);
      for (iCol = 0; iCol < iChunkSize; iCol++)
         ASSURE(apcChunks[i][iCol] == c);
   }
   #endif
   HeapMgr_free(apcChunks[i]);
}

/* Allocate chunks in decreasing order by size, thus maximizing the
   amount of list traversal required. */
i = iCount;
while (i >= 2)
{
   i--;
   i--;
   iChunkSize =
      (int)(((double)i * ((double)iSize / (double)iCount)) + 1.0);
   apcChunks[i] = HeapMgr_malloc((size_t)iChunkSize);
   if (apcChunks[i] == NULL)
   {
      printf("Malloc returned NULL.\n");
      exit(0);
   }
}

/* Free all chunks. */
for (i = 0; i < iCount; i++)
   HeapMgr_free(apcChunks[i]);
}
```

heapmgr1.c (Page 1 of 1)

```
 1: /*-------------------------------------------------------------------------*/
 2: /* heapmgr1.c                                                              */
 3: /* Author: Bob Dondero                                                     */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #define _GNU_SOURCE
 7:
 8: #include "heapmgr.h"
 9: #include <unistd.h>
10:
11: /* On our system the largest type is long double. */
12: typedef long double LargestType;
13:
14: /*-------------------------------------------------------------------------*/
15:
16: void *HeapMgr_malloc(size_t uBytes)
17: {
18:    static char *pcBrk;
19:
20:    char *pc = pcBrk;
21:    char *pcNewBrk;
22:
23:    if (uBytes == 0)
24:       return NULL;
25:
26:    /* Make sure uBytes is a multiple of the size of the largest
27:       data type. */
28:    uBytes =
29:       (((uBytes - 1) / sizeof(LargestType)) + 1) * sizeof(LargestType);
30:
31:    /* If this is the first call, then initialize. */
32:    if (pc == NULL)
33:       pc = sbrk(0);
34:
35:    /* Move the heap end. */
36:    pcNewBrk = pc + uBytes;
37:    if (pcNewBrk < pc) /* Check for overflow. */
38:       return NULL;
39:    if (brk(pcNewBrk) == -1)
40:       return NULL;
41:    pcBrk = pcNewBrk;
42:
43:    return (void*)pc;
44: }
45:
46: /*-------------------------------------------------------------------------*/
47:
48: void HeapMgr_free(void *pv)
49: {
50: }
```

heapmgr2.c (Page 1 of 1)

```
 1: /*--------------------------------------------------------------------*/
 2: /* heapmgr2.c                                                         */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #define _GNU_SOURCE
 7:
 8: #include "heapmgr.h"
 9: #include <unistd.h>
10:
11: /* On our system the largest type is long double. */
12: typedef long double LargestType;
13:
14: #define MAX(x, y) (x) < (y) ? (y) : (x)
15:
16: void *HeapMgr_malloc(size_t uBytes)
17: {
18:    enum {MIN_ALLOC = 8192};
19:
20:    static char *pcBrk;
21:    static char *pcPad;
22:
23:    char *pc;
24:    char *pcNewBrk;
25:
26:    if (uBytes == 0)
27:       return NULL;
28:
29:    /* Make sure uBytes is a multiple of the size of the largest
30:       data type. */
31:    uBytes =
32:       (((uBytes - 1) / sizeof(LargestType)) + 1) * sizeof(LargestType);
33:
34:    /* If this is the first call, then initialize. */
35:    if (pcBrk == NULL)
36:    {
37:       pcBrk = sbrk(0);
38:       pcPad = pcBrk;
39:    }
40:
41:    /* Move the heap end (and thereby increase the size of the pad)
42:       if necessary. */
43:    if (pcPad + uBytes > pcBrk)
44:    {
45:       pcNewBrk = MAX(pcPad + uBytes, pcBrk + MIN_ALLOC);
46:       if (pcNewBrk < pcBrk) /* Check for overflow. */
47:          return NULL;
48:       if (brk(pcNewBrk) == -1)
49:          return NULL;
50:       pcBrk = pcNewBrk;
51:    }
52:
53:    /* Allocate memory from the pad. */
54:    pc = pcPad;
55:    pcPad = pcPad + uBytes;
56:    return (void*)pc;
57: }
58:
59: void HeapMgr_free(void *pv)
60: {
61: }
```