

```
1 // Glenn Hewey
2 // CECS 424
3 // Lab 2 – Dynamic memory allocator (heap manager) in C
4 // Due: March 9 ,2018
5
6 #include "stdio.h"
7 #include "stdlib.h"
8
9 struct Block {
10     int block_size; // # of bytes in the data section
11     struct Block *next_block; // in C, you have to use •struct Block• as the
    type
12 };
13
14 const int OVERHEADSIZE = sizeof(struct Block);
15 const int MINSIZE = sizeof(void*);
16 struct Block *free_head;
17 // points to memory allocated with malloc inorder to deallocate at end of
    program
18 struct Block *head;
19
20 // initialize heap to specified size
21 void my_initialize_heap(int size){
22     free_head = (struct Block*) malloc(size);
23     (*free_head).block_size = size - OVERHEADSIZE;
24     (*free_head).next_block = NULL;
25 }
26
27 void remove_front() {
28
29     // temp head
30     struct Block *front = free_head;
31
32     // point free head to the next block
33     free_head = (*free_head).next_block;
34
35     // set the next pointer of the block you just removed equal to null
36     (*front).next_block = NULL;
37 }
38
39 void remove_back() {
40     struct Block *cursor = free_head;
41
42     // keeps track of the previous block when traversing the free list
43     struct Block *back = NULL;
44
45     // traverse free list till you find the tail
46     while((*cursor).next_block != NULL)
47     {
48         back = cursor;
49         cursor = (*cursor).next_block;
50     }
51
52     // back will not equal null if you traversed the free list
53     // remove the block from the free list
54     if(back != NULL){
55         (*back).next_block = NULL;
56     }
57
58 }
```

```
59
60 // Remove a block from the free list depending on if it is
61 // in the front back or in the middle of the free list
62 void my_remove(struct Block* nd){
63
64     // if current block is the head of the free list
65     if(nd == free_head){
66         remove_front();
67     }
68
69     // if current block is the tail of the free list
70     if((*nd).next_block == NULL){
71         remove_back();
72     }
73
74     // if the current block is in the middle of the free list
75     struct Block* cursor = free_head;
76     while(cursor != NULL)
77     {
78         if((*cursor).next_block == nd)
79             break;
80         cursor = (*cursor).next_block;
81     }
82
83     if(cursor != NULL)
84     {
85         struct Block* tmp = (*cursor).next_block;
86         (*cursor).next_block = (*tmp).next_block;
87         (*tmp).next_block = NULL;
88     }
89 }
90
91 void my_free(void* addr){
92
93     // set the blocks next equal to the free head
94     struct Block* block = (void*)addr-OVERHEADSIZE;
95     (*block).next_block = free_head;
96
97     // set free_head equal to the the block we just freed
98     free_head = block;
99 }
100
101 // Rounds the size requested up to a multiple of the MINSIZE = 8
102 // (sizeof(void*))
103 int RoundUpToMultiple(int size){
104     int remainder = abs(size) % MINSIZE;
105
106     if (remainder == 0)
107         return size;
108
109     return size + MINSIZE - remainder;
110 }
111
112 void *my_alloc(int size){
113
114     // Round up size
115     size = RoundUpToMultiple(size);
116
117     int found = 0;
```

```
118 // Walk free_head
119 struct Block *cur = free_head;
120 while(cur != NULL){
121     if ( size <= (*cur).block_size ){
122         found = 1;
123         break;
124     } else {
125         cur = (*cur).next_block;
126     }
127 }
128
129 // if there is a block with enough space. found will be set to 1
130 // if not return 0
131
132 if(found == 1){
133
134     // current free block is large enough to fit the size being allocated
135     // AND the excess space in the data portion is sufficient to fit
another block with over head
136     if( ((*cur).block_size - size - OVERHEADSIZE - MINSIZE) > 0 ){
137
138         // Split
139
140         //Calculate the left over size after the split
141         int leftOver = (*cur).block_size - size;
142
143         // create a block address of new block is equal to the cur addr +
overhead + size
144         struct Block* block = (void*)cur+OVERHEADSIZE+size;
145
146         // Set the next block equal to the current blocks next block
147         (*block).next_block = (*cur).next_block;
148
149         // Set the new blocks size equal to the leftover size - the
required overheadsize
150         (*block).block_size = leftOver - OVERHEADSIZE;
151         free_head = block;
152
153         // Set the size of the current block equal to the size requested
154         (*cur).block_size = size;
155
156         // Remove the next pointer does not point to anything
157         // because it is not a part of the free_list anymore
158         (*cur).next_block = NULL;
159
160     }else{
161
162         //Remove from free list
163         my_remove(cur);
164
165     }
166     // return addr to new block
167     // add 16 bytes to get the address of the data portion of the block
168     return (void*)cur+OVERHEADSIZE;
169 }else{
170     return 0;
171 }
172 }
173
174 int main()
```

```
175 {
176     void* a;
177     void* b;
178     void* c;
179     void* d;
180     void* e;
181     int n = 5;
182
183     my_initialize_heap(1000);
184
185     switch(n){
186         case 1:
187             //Test 1
188             printf("\tTest 1\n");
189             a = my_alloc(sizeof(int));
190             printf("a: %p\n", a);
191             my_free(a);
192             b = my_alloc(sizeof(int));
193             printf("b: %p\n", b); // The addresses should be the same.
194             break;
195         case 2:
196             //Test 2
197             printf("\tTest 2\n");
198             a = my_alloc(sizeof(int));
199             b = my_alloc(sizeof(int));
200             printf("a: %p\n", a);
201             printf("b: %p\n", b); // address should be 24 (0x18) bytes apart
202             break;
203         case 3:
204             //Test 3
205             printf("\tTest 3\n");
206             a = my_alloc(sizeof(int));
207             b = my_alloc(sizeof(int));
208             c = my_alloc(sizeof(int));
209             printf("a: %p\n", a);
210             printf("b: %p\n", b);
211             printf("c: %p\n", c);
212             my_free(b);
213             d = my_alloc(sizeof(double)); // should be the same as b's
214             address
215             printf("d: %p\n", d);
216             e = my_alloc(sizeof(int)); // address should be 24 (0x18) bytes
217             apart from c
218             printf("e: %p\n", e);
219             break;
220         case 4:
221             //Test 4
222             printf("\tTest 4\n");
223             a = my_alloc(sizeof(char));
224             b = my_alloc(sizeof(int));
225             printf("a: %p\n", a);
226             printf("b: %p\n", b); //Should equal b in test 2
227             break;
228         case 5:
229             //Test 5
230             printf("\tTest 5\n");
231             a = my_alloc(sizeof(int[100]));
232             b = my_alloc(sizeof(int));
233             printf("a: %p\n", a);
234             printf("b: %p\n", b);
```

```
233         my_free(a);
234         printf("b: %p\n", b); // Should not change
235         break;
236     default:
237         printf("\n");
238 }
239 free(head);
240 return 0;
241 }
242
```