Habib University
Operating Systems - CS232

Homework 03 - Report

Memory Managment

Instructor: Munzir Zafar

Sadiqah Mushtaq - 07152

Contents

1 Introduction

This assignment involves the implementation of a memory management system in C, simulating stack and heap memory with specific constraints. The program is required to process commands related to creating and deleting frames, managing variables, and handling heap memory. The report will discuss the chosen data structures and algorithms. The assignment emphasizes the practical aspects of correct command functionality, a well-structured makefile, and proper submission.

2 Makefile

```
build:
  gcc -o memorysystem memorysystem.c

run:
  ./memorysystem

clean:
  rm -rf memorysystem
```

The Makefile streamlines the build and execution processes for the C program `memorysystem.c`. It consists of three targets: `build`, `run`, and `clean`.

- **Build:** The `build` target compiles the source code using gcc, creating an executable named `memorysystem` with the command `gcc -o memorysystem memorysystem.c`.
- **Run:** The `run` target executes the compiled program via `./memorysystem`.
- **Clean:** The `clean` target removes the compiled executable using `rm -rf memorysystem`.

These targets can be utilized with simple commands: `make build` to compile, `make run` to execute, and `make clean` to remove the executable.

3 Input

The input for the program is a series of commands provided by the user in an interactive, shell-like environment. The commands are designed to simulate memory management operations in a stack and heap system. Each command has a specific syntax and performs a particular task related to memory allocation, deallocation, and snapshot display. The input assumes correct formatting, and after executing each command, the program updates the state of the heap and stack memory in an in-memory data structure.

Sample Input:

```
prompt> CF main 45
prompt> CI x 5
prompt> CD y 4.5
prompt> CC z c
prompt> CH Node 20
prompt> CD Node
prompt> SM
```

4 Structures and Typedefs

- `struct framestatus` represents the status of a stack frame, storing information such as frame number, function name, function address, frame address, and usage status.
- `struct freelist` represents a node in the free list used for heap memory, storing information about the start address, size, and a pointer to the next free region.
- `struct allocated` represents a node in the list of allocated memory on the heap, storing information about the allocated memory's name, start address, and a pointer to the next allocated region.

5 Functions

5.1 CF - Create Frame

```
9  void create_frame(char *name, int address, struct framestatus *framestatus_array,
       int *top, char *memory, int *frame_head){
10     printf("create frame function called\n");
11
12     //=================================================================
13
14     // check if the stack is full
15     if (*top == MAX_FRAMES - 1) {
16         printf("cannot create another frame, maximum number of frames have reached
       \n");
17         return;
18     }
19
20     //check if the frame with the same name already exists
21     for (int i = 0; i <= *top; i++) {
22         if (strcmp(framestatus_array[i].name, name) == 0) {
23             printf("frame already exists\n");
24             return;
25         }
26     }
27
28     // check if there is enough memory available for new frame
29     if (105 - sizeof(struct framestatus) * (*top + 1) < sizeof(struct framestatus)
       ) {
30         printf("stack overflow, not enough memory available for new frame\n");
31         return;
32     }
33
34     // create a new frame
35     struct framestatus fs;
36     fs.frameaddress = (long long int)*frame_head;
37     fs.used = '1';
38     fs.number = *top + 1;
39     fs.functionaddress = address;
40     strcpy(fs.name, name);
41     // fs.size = 0;
42     (*top)++;
43     printf("top: %d\n", *top);
44     framestatus_array[*top] = fs;
45
46     //=================================================================
```

```
47      // update the memory buffer. Remove comment to view with sprintf
48      // sprintf(&memory[(*frame_head)], "%d", fs.number);
49      // memcpy(&memory[(*frame_head)], &fs.number, sizeof(int));
50      // sprintf(&memory[(*frame_head) + 4], "%s", fs.name);
51      // memcpy(&memory[(*frame_head) + 4], fs.name, sizeof(char) * 8);
52      // sprintf(&memory[(*frame_head) + 12], "%d", fs.functionaddress);
53      // memcpy(&memory[(*frame_head) + 12], &fs.functionaddress, sizeof(int));
54      // sprintf(&memory[(*frame_head) + 16], "%d", fs.frameaddress);
55      // memcpy(&memory[(*frame_head) + 16], &fs.frameaddress, sizeof(int));
56      // sprintf(&memory[(*frame_head) + 17], "%c", fs.used);
57      // memcpy(&memory[(*frame_head) + 17], &fs.used, sizeof(char));
58
59      memcpy(&memory[394], framestatus_array, sizeof(framestatus_array));
60  }
```

This function creates a new frame on the stack. It checks for errors such as stack overflow, existing frame names, and maximum frame limit. If no errors are encountered, it creates a new frame in the framestatus array and updates the memory accordingly.

5.2 DF - Delete Frame

```
61  // delete the current frame from the stack
62  void delete_frame(struct framestatus *framestatus_array, char *memory, int *
        frame_head, int *top){
63      printf("delete frame function called\n");
64      if(top <0){
65          printf("stack is empty\n");
66          return;
67      }
68      printf("Head before deletion: %d\n", *frame_head);
69      *frame_head = framestatus_array[*top].frameaddress;
70      framestatus_array[*top].used = '0';
71      framestatus_array[*top].number = -1;
72      framestatus_array[*top].functionaddress = -1;
73      framestatus_array[*top].frameaddress = -1;
74      strcpy(framestatus_array[*top].name, "0");
75      printf("Head after deletion: %d\n", *frame_head);
76
77      (*top)--;
78      memcpy(&memory[394], framestatus_array, sizeof(framestatus_array));
79
80  }
```

This function deletes the top frame from the stack. It updates the frame status and memory accordingly.

5.3 CI - Create Integer

```
81  // create an integer variable on the current frame
82  void create_integer(char *name, int *value, struct framestatus *framestatus_array,
         int *top, char *memory, int *frame_head) {
83      printf("create integer function called\n");
84      printf("integer name: %s\n", name);
85      printf("integer value: %d\n", *value);
86
87      printf("top: %d\n", *top);
88
89      // Check if the stack is empty
```

```
90      if (*top == -1) {
91          printf("stack is empty\n");
92          return;
93      }
94
95      // Update the frame_head
96      printf("frame_head: %d\n", *frame_head);
97      *frame_head -= sizeof(int);
98      printf("frame_head: %d\n", *frame_head);
99
100
101     sprintf(&memory[(*frame_head)], "%d", *value);
102     memcpy(&memory[(*frame_head)], value, sizeof(int));
103
104
105     printf("Integer variable '%s' created at address %d with value %d\n", name, *
        frame_head, *value);
106     printf("frame_head: %d\n", *frame_head);
107     printf("\n");
108     return;
109 }
```

This function creates an integer variable on the current frame. It updates the frame_head and memory with the new variable.

5.4 CD - Create Double

```
110 // create a double variable on the current frame
111 void create_double(char *name, double *value, struct framestatus *
        framestatus_array, int *top, char *memory, int *frame_head){
112     printf("create double function called\n");
113     printf("double name: %s\n", name);
114     printf("double value: %f\n", *value);
115
116         // Check if the stack is empty
117     if (*top == -1) {
118         printf("stack is empty\n");
119         return;
120     }
121
122     // Update the frame_head
123     printf("frame_head: %d\n", *frame_head);
124     *frame_head -= sizeof(double);
125     printf("frame_head: %d\n", *frame_head);
126
127     // Create the DOUBLE variable at the LOCTAION OF frame_head
128     // sprintf(&memory[(*frame_head)], "%f", *value);
129     memcpy(&memory[(*frame_head)], value, sizeof(double));
130     printf("frame_head3: %d\n", *frame_head);
131
132
133
134     printf("Double variable '%s' created at address %d with value %f\n", name, *
        frame_head, *value);
135     printf("\n");
136     return;
137 }
```

This function creates a double variable on the current frame. It updates the frame_head and memory with the new variable.

5.5 CC - Create Character

```
138 // create a character variable on the current frame
139 void create_character(char *name, char *value, struct framestatus *
        framestatus_array, int *top, char *memory, int *frame_head){
140     printf("create character function called\n");
141     printf("character name: %s\n", name);
142     printf("character value: %c\n", *value);
143 
144         printf("top: %d\n", *top);
145 
146     // Check if the stack is empty
147     if (*top == -1) {
148         printf("stack is empty\n");
149         return;
150     }
151 
152     // Update the frame_head
153     printf("frame_head: %d\n", *frame_head);
154     *frame_head -= sizeof(char);
155     printf("frame_head: %d\n", *frame_head);
156 
157     // Create the CHARCTER variable at the LOCTAION OF frame_head
158     // sprintf(&memory[(*frame_head)], "%c", *value);
159     memcpy(&memory[(*frame_head)], value, sizeof(char));
160     printf("frame_head3: %d\n", *frame_head);
161 
162     printf("Character variable '%s' created at address %d with value %c\n", name,
        *frame_head, *value);
163     return;
164     printf("\n");
165 }
```

This function creates a character variable on the current frame. It updates the frame_head and memory with the new variable.

5.6 SM - Snapshot Memory

```
166 // print the stack or heap for debugging purposes
167 void print(char *memory){
168     printf("print function called\n");
169     // prints hex values of memory
170     for(int i = 0; i < MEMSIZE; i++){
171         printf("%d = %x\n", i, memory[i]);
172     }
173     printf("\n");
174 
175     // uncomment if using sprintf
176     // printf("==============================For Checking Purposes
        ==========================\n");
177     // To see decimal values of memory
178     // for(int i = 0; i < MEMSIZE; i++){
179     //     printf("%d = %d\n", i, memory[i]);
180     // }
181     // printf("\n");
182     // To see characters of memory
183     // for (int i = 0; i < MEMSIZE; i++) {
184     //     printf("%d = %c  ", i, memory[i]);
185     //     if ((i + 1) % 10 == 0) {
186     //         printf("\n");
```

```
187     //      }
188     // }
189     printf("\n");
190 }
```

This function prints the content of the memory buffer for debugging purposes, showing hexadecimal values.

5.7 CH - Create Buffer Heap

```
191 /// Heap cannot be created without frame
192 void create_buffer_heap(char* name, int* size, struct freelist** head, struct
        allocated** head_allocated, char* memory, int* frame_head, int *top) {
193     printf("create buffer function called\n");
194     printf("buffer name: %s\n", name);
195     printf("buffer size: %d\n", *size);
196 
197 
198     // Finding free node in the freelist
199     struct freelist* temp = *head;
200     while (temp->size < (*size) + 8) {
201         temp = temp->next;
202         if (temp == NULL) {
203             printf("Error: Not enough free space in heap.\n");
204             return;
205         }
206     }
207 
208     // Checking if frame is open
209     if(*top == -1){
210         printf("Error: No frame is open. No function created\n");
211         return;
212     }
213 
214 
215     // Updating allocated list
216     (*head_allocated)->startaddress = temp->start;
217     append_allocated(head_allocated, name, (*head_allocated)->startaddress);
218     printf("Allocated list: \n");
219     printAllocated(*head_allocated);
220     printf("\n");
221 
222     // updating frame
223     int store = (*head_allocated)->startaddress;
224     *frame_head -= sizeof(int);
225     memcpy(&memory[*frame_head], &store, sizeof(int));
226 
227     // Updating free list
228     temp->size = temp->size - (*size) - 8;
229     temp->start = temp->start + (*size) + 8;
230     printf("Free list: \n");
231     printFreelist(*head);
232     printf("\n");
233 
234     // Create the buffer
235     int magic_no = rand() % 1000;
236     char* str = (char*)malloc(*size);
237     srand(time(NULL));
238     for (int i = 0; i < *size - 1; i++) {
239         str[i] = 'A' + (rand() % 26);
```

```
240         printf("%c", str[i]);
241     }
242     printf("\n");
243 
244     // Update the memory buffer. Remove comment from lines if sprintf if u want to
     view the memory for checking
245     // sprintf(&memory[store], "%d", *size);
246     memcpy(&memory[store], size, sizeof(int));
247     // sprintf(&memory[store + 4], "%d", magic_no);
248     memcpy(&memory[store + 4], &magic_no, sizeof(int));
249     // sprintf(&memory[store + 8], "%s", str);
250     memcpy(&memory[store + 8], str, *size);
251 
252     free(str);  // Free dynamically allocated memory
253 
254     printf("\n");
255 
256     return;
257 }
```

This function creates a buffer on the heap. It allocates space in the freelist, updates the allocated list, and initializes the buffer with random characters.

5.8 DH - Deallocate Buffer Heap

```
258 // delete a buffer from the heap
259 void delete_buffer_heap(char *name, struct freelist **head, struct allocated **
    head_allocated, char *memory, int *frame_head)
260 {
261     printf("delete buffer function called\n");
262     printf("buffer name: %s\n", name);
263 
264     // Finding the buffer in the allocated list
265     struct allocated *temp = *head_allocated;
266     if (temp == NULL)
267     {
268         printf("Error: Allocated list is empty.\n");
269         return;
270     }
271     struct allocated *temp1 = NULL;
272     while (strcmp(temp->name, name) != 0)
273     {
274         temp1 = temp; // buffer before the buffer to be deleted in the list
275         temp = temp->next; // buffer to be deleted
276         if (temp == NULL)
277         {
278             printf("Error: Buffer to delete not found in heap\n");
279             return;
280         }
281     }
282 
283     // Updating allocated list
284     if (temp1 != NULL){
285         temp1->next = temp->next;
286     }
287     else{
288         *head_allocated = temp->next;
289     }
290     temp->next = NULL;
291     printf("Allocated list after deallocating buffer: \n");
```

```
292     printAllocated(*head_allocated);
293 
294     // Updating free list
295     int size;
296     memcpy(&size, &memory[temp->startaddress], sizeof(int)); // fetching size of
        buffer to be deleted from its metadata
297     struct freelist *temp2 = (struct freelist *)malloc(sizeof(struct freelist));
298     if (temp2 == NULL){
299         printf("Memory allocation failed\n");
300         return;
301     }
302     struct freelist *temp3 = *head;
303     if (temp3 == NULL)
304     { // if freelist was empty
305         temp2->start = temp->startaddress;
306         temp2->size = size + 8;
307         temp2->next = NULL;
308         *head = temp2;
309         printf("Free list after deallocating buffer: \n");
310         printFreelist(*head);
311         printf("\n");
312         free(temp2); // Free the dynamically allocated temp2
313         return;
314     }
315     temp2->size = size + 8; // updating size of the first node in free list (
        including the 8 bytes of metadata)
316     temp2->start = temp->startaddress; // updating start address of the first node
         in free list
317     temp2->next = temp3;               // updating next of the first node in free
        list
318     *head = temp2;                     // updating head of free list
319     printf("Free list after deallocating buffer: \n");
320     printFreelist(*head);
321     printf("\n");
322 
323     // Updating memory
324     memset(&memory[temp->startaddress], '0', size + 8); // Zero out the memory of
        the deleted buffer
325 
326     printf("\n");
327 }
```

This function deletes a buffer from the heap. It updates the freelist, allocated list, and sets the memory of the deleted buffer to zero.

6 Output

Most functions are supposed to be void. The only function that prints the snapshot of the entire memory array is the print function or the SM command. However, I have printed necessary elements in each function in order to check that they are working properly. I have added screenshots for the out of each command. If use the sprintf command it helps see the snapshot of memory clearly. However, that is not the requirement of the assignment so I have commented out those lines and have put screenshot of the output I was getting for each command in this report after using sprintf for verifcation purposes.

CF command

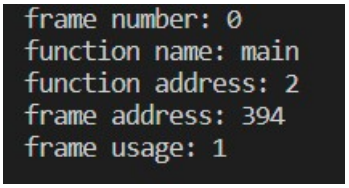
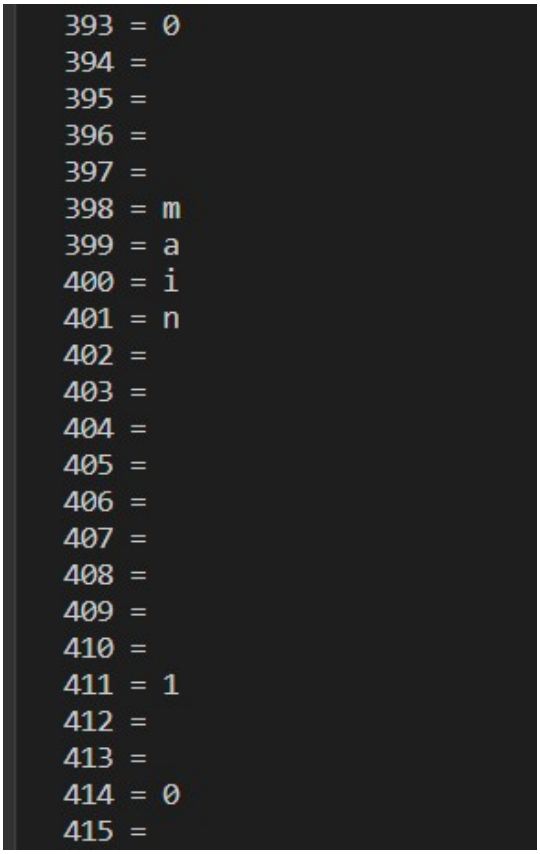
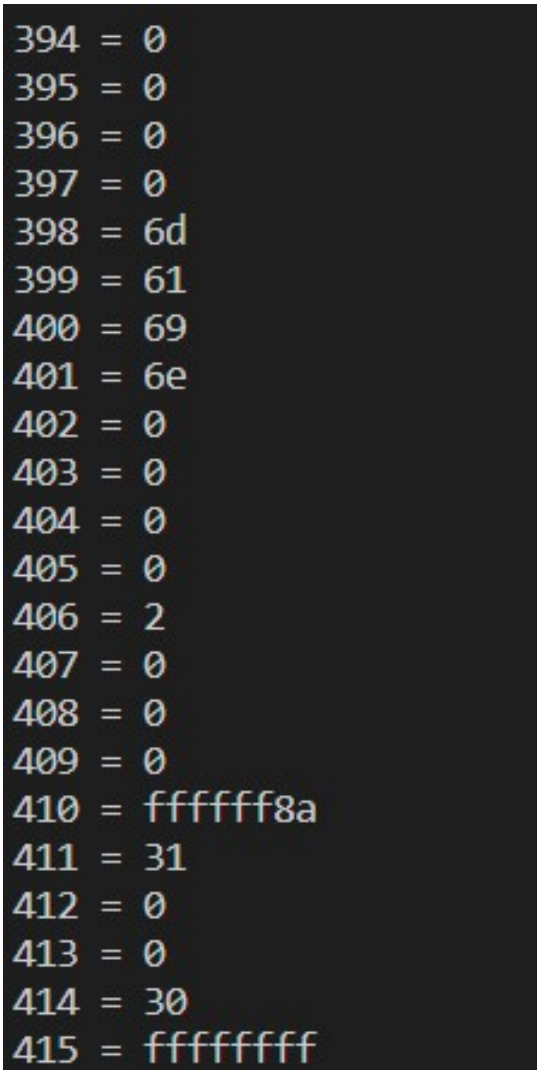


Figure 1: Output of CF Main 2 Command: Simple Output

In case of hex, the first four bytes represent the frame number, which is 0. The following 8 bytes represent the function name in hexadecimal. The next 4 bytes represent function address '2', and so on.



(a) SM Function Output with Hex Format



(b) SM Function Output with Hex Format

Figure 2: Different Outputs of CF Main 2 Command

DF command

```
prompt>DF
delete frame function called
Head before deletion: 390
Head after deletion: 394
prompt>
```

Figure 3: Output of DF Command: Simple Output

Simply moves the frame head back and changes the values in the frame status array.

CI command

```
prompt>CI x 6
create integer function called
integer name: x
integer value: 6
top: 0
frame_head: 394
frame_head: 390
Integer variable 'x' created at address 390 with value 6
frame_head: 390
```

(a) Basic Check

```
387 = 30
388 = 30
389 = 30
390 = 6
391 = 0
392 = 0
393 = 0
394 = 0
```

(b) SM Function Output with Hex Format (using sprintf)

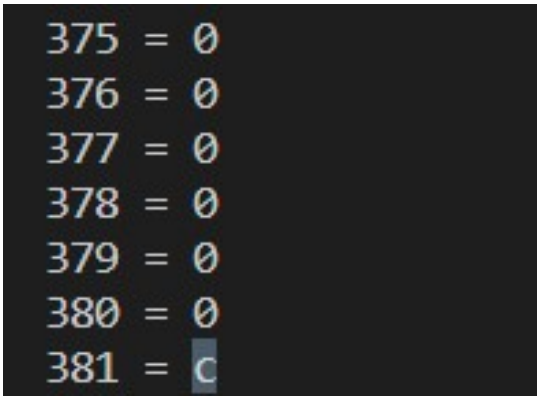
Figure 4: Outputs of **CI x 6** Command

At index 390 of the memory array 6 is clearly visible.

CC command

```
prompt>CC b c
create character function called
character name: b
character value: c
top: 0
frame_head: 382
frame_head: 381
frame_head3: 381
Character variable 'b' created at address 381 with value c
prompt>
```

(a) Basic Check



(b) SM Function Output with Char Format (using sprintf)

Figure 5: Outputs of **CC b c** Command

At index 381 of the memory array character 'c' is clearly visible.

CD command

Even though the output of Cd command does not make much sense when use sprintf and hex format specifier, following is the screenshot of the output. It is important to note that values can always be printed in hex without using sprintf and then analysed as well.

```
prompt>CD c 5.5
create double function called
double name: c
double value: 5.500000
frame_head: 394
frame_head: 386
frame_head3: 386
Double variable 'c' created at address 386 with value 5.500000
```

(a) Basic Check

```
prompt>CD v 6.8
create double function called
double name: v
double value: 6.800000
frame_head: 390
frame_head: 382
frame_head3: 382
Double variable 'v' created at address 382 with value 6.800000
```

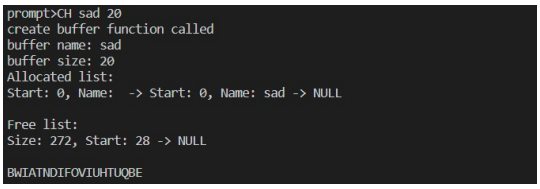
(b) SM Function Output with Hex Format (using sprintf)

Figure 6: Outputs of **CD c 5.5** Command

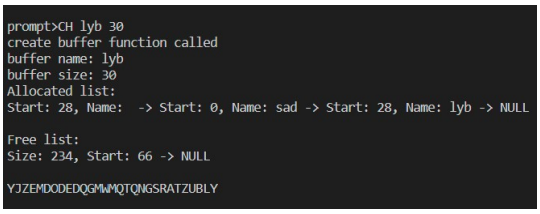
Even though the output of Cd command does not make much sense when use sprintf and hex format specifier, following is the screenshot of the output. It is important to note that values can always be printed in hex without using sprintf, and then analysed.

CH command

We can simply print the allocation list and freelist to verify that heap is correctly allocated.



(a) Outputs of **CH sad 20** Command

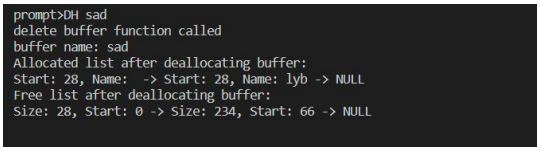


(b) Outputs of **CH lyb 30** Command)

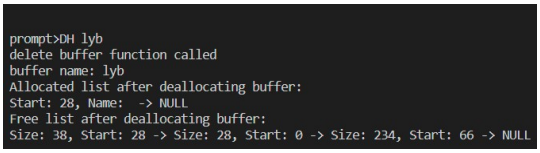
Figure 7: Outputs of two **CH** Command (b followed by a)

DH command

We can simply print the allocation list and freelist to verify that heap is correctly deallocated.



(a) Outputs of **DH sad** Command



(b) Outputs of **DH lyb** Command)

Figure 8: Outputs of two **DH** Command (b followed by a)

SM command

I am testing this with the following test case and without the sprintf command.

```
    prompt>CF main 5
create frame function called
top: 0
prompt>CI x 5
create integer function called
integer name: x
integer value: 5
top: 0
frame_head: 394
frame_head: 390
Integer variable 'x' created at address 390 with value 5
frame_head: 390

prompt>CI v 5.5
create integer function called
integer name: v
integer value: 5
top: 0
frame_head: 390
frame_head: 386
Integer variable 'v' created at address 386 with value 5
frame_head: 386

prompt>Invalid command
prompt>CD b 6.6
create double function called
double name: b
double value: 6.600000
frame_head: 386
frame_head: 378
frame_head3: 378
Double variable 'b' created at address 378 with value 6.600000

prompt>CC s v
create character function called
character name: s
character value: v
top: 0
frame_head: 378
frame_head: 377
frame_head3: 377
Character variable 's' created at address 377 with value v
prompt>CH H2 10
create buffer function called
buffer name: H2
buffer size: 10
```

```
Allocated list:
Start: 28, Name:  -> Start: 0, Name: H1 -> Start: 28, Name: H2 -> NULL

Free list:
Size: 254, Start: 46 -> NULL

PRWSDXGYU

prompt>DH H1
delete buffer function called
buffer name: H1
Allocated list after deallocating buffer:
Start: 28, Name:  -> Start: 28, Name: H2 -> NULL
Free list after deallocating buffer:
Size: 28, Start: 0 -> Size: 254, Start: 46 -> NULL

prompt>SM
print function called
0 = 30
1 = 30
2 = 30
3 = 30
4 = 30
5 = 30
6 = 30
7 = 30
8 = 30
9 = 30
10 = 30
11 = 30
12 = 30
13 = 30
14 = 30
15 = 30
16 = 30
17 = 30
18 = 30
19 = 30
20 = 30
21 = 30
22 = 30
23 = 30
24 = 30
25 = 30
26 = 30
27 = 30
28 = a
29 = 0
30 = 0
```

```
31 = 0
32 = 2e
33 = 2
34 = 0
35 = 0
36 = 50
37 = 52
38 = 57
39 = 53
40 = 44
41 = 58
42 = 47
43 = 59
44 = 55
45 = 0
46 = 30
47 = 30
48 = 30
49 = 30
50 = 30
51 = 30
52 = 30
53 = 30
54 = 30
55 = 30
56 = 30
57 = 30
58 = 30
59 = 30
60 = 30
61 = 30
62 = 30
63 = 30
64 = 30
65 = 30
66 = 30
67 = 30
68 = 30
69 = 30
70 = 30
71 = 30
72 = 30
73 = 30
74 = 30
75 = 30
76 = 30
77 = 30
78 = 30
79 = 30
```

```
80 = 30
81 = 30
82 = 30
83 = 30
84 = 30
85 = 30
86 = 30
87 = 30
88 = 30
89 = 30
90 = 30
91 = 30
92 = 30
93 = 30
94 = 30
95 = 30
96 = 30
97 = 30
98 = 30
99 = 30
100 = 30
101 = 30
102 = 30
103 = 30
104 = 30
105 = 30
106 = 30
107 = 30
108 = 30
109 = 30
110 = 30
111 = 30
112 = 30
113 = 30
114 = 30
115 = 30
116 = 30
117 = 30
118 = 30
119 = 30
120 = 30
121 = 30
122 = 30
123 = 30
124 = 30
125 = 30
126 = 30
127 = 30
128 = 30
```

```
129 = 30
130 = 30
131 = 30
132 = 30
133 = 30
134 = 30
135 = 30
136 = 30
137 = 30
138 = 30
139 = 30
140 = 30
141 = 30
142 = 30
143 = 30
144 = 30
145 = 30
146 = 30
147 = 30
148 = 30
149 = 30
150 = 30
151 = 30
152 = 30
153 = 30
154 = 30
155 = 30
156 = 30
157 = 30
158 = 30
159 = 30
160 = 30
161 = 30
162 = 30
163 = 30
164 = 30
165 = 30
166 = 30
167 = 30
168 = 30
169 = 30
170 = 30
171 = 30
172 = 30
173 = 30
174 = 30
175 = 30
176 = 30
177 = 30
```

```
178 = 30
179 = 30
180 = 30
181 = 30
182 = 30
183 = 30
184 = 30
185 = 30
186 = 30
187 = 30
188 = 30
189 = 30
190 = 30
191 = 30
192 = 30
193 = 30
194 = 30
195 = 30
196 = 30
197 = 30
198 = 30
199 = 30
200 = 30
201 = 30
202 = 30
203 = 30
204 = 30
205 = 30
206 = 30
207 = 30
208 = 30
209 = 30
210 = 30
211 = 30
212 = 30
213 = 30
214 = 30
215 = 30
216 = 30
217 = 30
218 = 30
219 = 30
220 = 30
221 = 30
222 = 30
223 = 30
224 = 30
225 = 30
226 = 30
```

```
227 = 30
228 = 30
229 = 30
230 = 30
231 = 30
232 = 30
233 = 30
234 = 30
235 = 30
236 = 30
237 = 30
238 = 30
239 = 30
240 = 30
241 = 30
242 = 30
243 = 30
244 = 30
245 = 30
246 = 30
247 = 30
248 = 30
249 = 30
250 = 30
251 = 30
252 = 30
253 = 30
254 = 30
255 = 30
256 = 30
257 = 30
258 = 30
259 = 30
260 = 30
261 = 30
262 = 30
263 = 30
264 = 30
265 = 30
266 = 30
267 = 30
268 = 30
269 = 30
270 = 30
271 = 30
272 = 30
273 = 30
274 = 30
275 = 30
```

```
276 = 30
277 = 30
278 = 30
279 = 30
280 = 30
281 = 30
282 = 30
283 = 30
284 = 30
285 = 30
286 = 30
287 = 30
288 = 30
289 = 30
290 = 30
291 = 30
292 = 30
293 = 30
294 = 30
295 = 30
296 = 30
297 = 30
298 = 30
299 = 30
300 = 30
301 = 30
302 = 30
303 = 30
304 = 30
305 = 30
306 = 30
307 = 30
308 = 30
309 = 30
310 = 30
311 = 30
312 = 30
313 = 30
314 = 30
315 = 30
316 = 30
317 = 30
318 = 30
319 = 30
320 = 30
321 = 30
322 = 30
323 = 30
324 = 30
```

```
325 = 30
326 = 30
327 = 30
328 = 30
329 = 30
330 = 30
331 = 30
332 = 30
333 = 30
334 = 30
335 = 30
336 = 30
337 = 30
338 = 30
339 = 30
340 = 30
341 = 30
342 = 30
343 = 30
344 = 30
345 = 30
346 = 30
347 = 30
348 = 30
349 = 30
350 = 30
351 = 30
352 = 30
353 = 30
354 = 30
355 = 30
356 = 30
357 = 30
358 = 30
359 = 30
360 = 30
361 = 30
362 = 30
363 = 30
364 = 30
365 = 30
366 = 30
367 = 30
368 = 30
369 = 1c
370 = 0
371 = 0
372 = 0
373 = 0
```

```
374 = 0
375 = 0
376 = 0
377 = 76
378 = 66
379 = 66
380 = 66
381 = 66
382 = 66
383 = 66
384 = 1a
385 = 40
386 = 5
387 = 0
388 = 0
389 = 0
390 = 5
391 = 0
392 = 0
393 = 0
394 = 0
395 = 0
396 = 0
397 = 0
398 = 6d
399 = 61
400 = 69
401 = 6e
402 = 0
403 = 0
404 = 0
405 = 0
406 = ffffffff
407 = ffffffff
408 = ffffffff
409 = ffffffff
410 = ffffffff
411 = ffffffff
412 = ffffffff
413 = ffffffff
414 = 30
415 = ffffffff
416 = ffffffff
417 = ffffffff
418 = ffffffff
419 = 4e
420 = 2f
421 = 41
422 = 0
```

```
423 = 0
424 = 0
425 = 0
426 = 0
427 = ffffffff
428 = ffffffff
429 = ffffffff
430 = ffffffff
431 = ffffffff
432 = ffffffff
433 = ffffffff
434 = ffffffff
435 = 30
436 = ffffffff
437 = ffffffff
438 = ffffffff
439 = ffffffff
440 = 4e
441 = 2f
442 = 41
443 = 0
444 = 0
445 = 0
446 = 0
447 = 0
448 = ffffffff
449 = ffffffff
450 = ffffffff
451 = ffffffff
452 = ffffffff
453 = ffffffff
454 = ffffffff
455 = ffffffff
456 = 30
457 = ffffffff
458 = ffffffff
459 = ffffffff
460 = ffffffff
461 = 4e
462 = 2f
463 = 41
464 = 0
465 = 0
466 = 0
467 = 0
468 = 0
469 = ffffffff
470 = ffffffff
471 = ffffffff
```

```
472 = ffffffff
473 = ffffffff
474 = ffffffff
475 = ffffffff
476 = ffffffff
477 = 30
478 = ffffffff
479 = ffffffff
480 = ffffffff
481 = ffffffff
482 = 4e
483 = 2f
484 = 41
485 = 0
486 = 0
487 = 0
488 = 0
489 = 0
490 = ffffffff
491 = ffffffff
492 = ffffffff
493 = ffffffff
494 = ffffffff
495 = ffffffff
496 = ffffffff
497 = ffffffff
498 = 30
499 = 30
```

A Appendix

A.1 Code

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>


#define MEMSIZE 500
#define MAX_COMMAND_LENGTH 1024
#define MAX_NAME 5
#define MAX_FRAMES 5

/*

Create a frame
syntax: CF functionname functionaddressx
This command should create a frame on stack. The functionname is a maximum of 8 characters.
The functionaddress is an assumed address of the function in program code.
If theres not enough space on stack, it should output an error saying  stack overflow, not enough
memory available for new function. If all the maximum number of frames have reached, it should
output an error saying  cannot  create another frame, maximum number of frames have reached.
If a function with the given name already exists, it should give an error  function  already exists .In
case of no errors, it should create a frame on stack and create an entry in framestatus_array.
4.2 Delete a Function
syntax: DF
This command deletes the function on top of the stack.
If no function exists on stack, it should output an error message saying  stack is empty.
4.3 Create integer local variable
syntax: CI integername integervalue
This command creates an integer of size 4 bytes on the current frame. If the frame is full, it should
output an error message saying  the  frame is full, cannot create more data on it .
4.4 Create double local variable
syntax: CD doublename doublevalue
This command creates a double of size of 8 bytes on the current frame. If the frame is full, it should
output an error message saying  the  frame is full, cannot create more data on it .
4.5 Create character local variable
syntax: CC charactername charactervalue
This command creates a character of size of 1 byte on the current frame. If the frame is full, it should
output an error message saying  the  frame is full, cannot create more data on it .
4.6 Create character buffer on heap
syntax: CH buffername size
```

```
371 This command allocates a buffer of bytes size plus 4 bytes on heap. It also
        creates a local pointer on stack
372 and stores the starting address of the allocated region. The buffer is filled with
         random characters. If the
373 heap is full, it should output an error message saying   the   heap is full, cannot
         create more  d a t a .
374 4.7 Deallocate a buffer on heapsyntax: DH buffername
375 This command de-allocates a buffer on stack. A total of buffer size plus 4 bytes
        are deallocated.
376 The data in the deallocated region is replaced with zeros. If the buffer was
        already de-allocated or the
377 pointer is invalid, output an error message saying   the   pointer is NULL or
        already de-allocated .
378 4.8 Show memory image
379 syntax: SM
380 This command should output the stack and heap snapshots.
381 */
382
383 #pragma pack(1)
384 struct framestatus {        // for stack
385     int number;             // frame number
386     char name[8];           // function name representing the frame
387     int functionaddress;    // address of function in the code section (will be
        randomly generated in this case)
388     int frameaddress;       // starting address of the frame belonging to this
        header in Stack
389     char used;              // a boolean value indicating whether the frame status
         entry is in use or not
390 };
391 #pragma pack()
392
393
394 struct freelist {            // for heap
395     int start;               // start address of free region
396     int size;                // size of free region
397     struct freelist* next;   // pointer to the next free region
398 };
399
400 struct allocated {
401     char name[8];
402     int startaddress;
403     struct allocated* next;
404 };
405
406 /*
407 Helper functions
408 */
409
410 void append_allocated(struct allocated** head, char* newName, int newAddress) {
411     struct allocated* newallocated = (struct allocated*)malloc(sizeof(struct
        allocated));
412     if (newallocated == NULL) {
413         printf("Memory allocation failed\n");
414         return;
415     }
416
417     newallocated->startaddress = newAddress;
418     strcpy(newallocated->name, newName);
419     newallocated->next = NULL;
420
421     if (*head == NULL) {
422         *head = newallocated;
```

```
423         return;
424     }
425 
426     struct allocated* lastallocated = *head;
427     while (lastallocated->next != NULL) {
428         lastallocated = lastallocated->next;
429     }
430 
431     lastallocated->next = newallocated;
432     // free(newallocated);
433 }
434 
435 void printAllocated(struct allocated* head_allocated) {
436     // Check if the allocated list is empty
437     if (head_allocated == NULL) {
438         printf("Empty\n");
439         return;
440     }
441 
442     // Print each node in the allocated list
443     while (head_allocated != NULL) {
444         printf("Start: %d, Name: %s -> ", head_allocated->startaddress,
    head_allocated->name);
445         head_allocated = head_allocated->next;
446     }
447     printf("NULL\n");
448 }
449 
450 void printFreelist(struct freelist* head) {
451     // Check if the freelist is empty
452     if (head == NULL) {
453         printf("Empty\n");
454         return;
455     }
456 
457     // Print each node in the freelist
458     while (head != NULL) {
459         printf("Size: %d, Start: %d -> ", head->size, head->start);
460         head = head->next;
461     }
462     printf("NULL\n");
463 }
464 
465 // fuction to free the allocated list
466 void free_allocated_list(struct allocated* head_allocated) {
467     struct allocated* current = head_allocated;
468     struct allocated* next;
469 
470     while (current != NULL) {
471         next = current->next;
472         current = next;
473     }
474 }
475 /*
476 Required Functions
477 */
478 
479 // create a new frame on the stack
480 void create_frame(char *name, int address, struct framestatus *framestatus_array,
    int *top, char *memory, int *frame_head){
481     printf("create frame function called\n");
482 
```

```
483     //
        ================================================================================================
484
485     // check if the stack is full
486     if (*top == MAX_FRAMES - 1) {
487         printf("cannot create another frame, maximum number of frames have reached
        \n");
488         return;
489     }
490
491     //check if the frame with the same name already exists
492     for (int i = 0; i <= *top; i++) {
493         if (strcmp(framestatus_array[i].name, name) == 0) {
494             printf("frame already exists\n");
495             return;
496         }
497     }
498
499     // check if there is enough memory available for new frame
500     if (105 - sizeof(struct framestatus) * (*top + 1) < sizeof(struct framestatus)
        ) {
501         printf("stack overflow, not enough memory available for new frame\n");
502         return;
503     }
504
505     // create a new frame
506     struct framestatus fs;
507     fs.frameaddress = (long long int)*frame_head;
508     fs.used = '1';
509     fs.number = *top + 1;
510     fs.functionaddress = address;
511     strcpy(fs.name, name);
512     // fs.size = 0;
513     (*top)++;
514     printf("top: %d\n", *top);
515     framestatus_array[*top] = fs;
516
517     //
        ================================================================================================
518     // update the memory buffer. Remove comment to view with sprintf
519     // sprintf(&memory[(*frame_head)], "%d", fs.number);
520     // memcpy(&memory[(*frame_head)], &fs.number, sizeof(int));
521     // sprintf(&memory[(*frame_head) + 4], "%s", fs.name);
522     // memcpy(&memory[(*frame_head) + 4], fs.name, sizeof(char) * 8);
523     // sprintf(&memory[(*frame_head) + 12], "%d", fs.functionaddress);
524     // memcpy(&memory[(*frame_head) + 12], &fs.functionaddress, sizeof(int));
525     // sprintf(&memory[(*frame_head) + 16], "%d", fs.frameaddress);
526     // memcpy(&memory[(*frame_head) + 16], &fs.frameaddress, sizeof(int));
527     // sprintf(&memory[(*frame_head) + 17], "%c", fs.used);
528     // memcpy(&memory[(*frame_head) + 17], &fs.used, sizeof(char));
529
530     memcpy(&memory[394], framestatus_array, sizeof(framestatus_array));
531 }
532
533 // delete the current frame from the stack
534 void delete_frame(struct framestatus *framestatus_array, char *memory, int *
        frame_head, int *top){
535     printf("delete frame function called\n");
536     if(top <0){
537         printf("stack is empty\n");
```

```
538         return;
539     }
540     printf("Head before deletion: %d\n", *frame_head);
541     *frame_head = framestatus_array[*top].frameaddress;
542     framestatus_array[*top].used = '0';
543     framestatus_array[*top].number = -1;
544     framestatus_array[*top].functionaddress = -1;
545     framestatus_array[*top].frameaddress = -1;
546     strcpy(framestatus_array[*top].name, "0");
547     printf("Head after deletion: %d\n", *frame_head);
548 
549     (*top)--;
550     memcpy(&memory[394], framestatus_array, sizeof(framestatus_array));
551 
552 }
553 
554 // create an integer variable on the current frame
555 void create_integer(char *name, int *value, struct framestatus *framestatus_array,
      int *top, char *memory, int *frame_head) {
556     printf("create integer function called\n");
557     printf("integer name: %s\n", name);
558     printf("integer value: %d\n", *value);
559 
560     printf("top: %d\n", *top);
561 
562     // Check if the stack is empty
563     if (*top == -1) {
564         printf("stack is empty\n");
565         return;
566     }
567 
568     // Update the frame_head
569     printf("frame_head: %d\n", *frame_head);
570     *frame_head -= sizeof(int);
571     printf("frame_head: %d\n", *frame_head);
572 
573 
574     sprintf(&memory[(*frame_head)], "%d", *value);
575     memcpy(&memory[(*frame_head)], value, sizeof(int));
576 
577 
578     printf("Integer variable '%s' created at address %d with value %d\n", name, *
      frame_head, *value);
579     printf("frame_head: %d\n", *frame_head);
580     printf("\n");
581     return;
582 }
583 
584 
585 // create a double variable on the current frame
586 void create_double(char *name, double *value, struct framestatus *
      framestatus_array, int *top, char *memory, int *frame_head){
587     printf("create double function called\n");
588     printf("double name: %s\n", name);
589     printf("double value: %f\n", *value);
590 
591         // Check if the stack is empty
592     if (*top == -1) {
593         printf("stack is empty\n");
594         return;
595     }
596 
```

```
597     // Update the frame_head
598     printf("frame_head: %d\n", *frame_head);
599     *frame_head -= sizeof(double);
600     printf("frame_head: %d\n", *frame_head);
601 
602     // Create the DOUBLE variable at the LOCTAION OF frame_head
603     // sprintf(&memory[(*frame_head)], "%f", *value);
604     memcpy(&memory[(*frame_head)], value, sizeof(double));
605     printf("frame_head3: %d\n", *frame_head);
606 
607 
608 
609     printf("Double variable '%s' created at address %d with value %f\n", name, *
    frame_head, *value);
610     printf("\n");
611     return;
612 }
613 
614 // create a character variable on the current frame
615 void create_character(char *name, char *value, struct framestatus *
    framestatus_array, int *top, char *memory, int *frame_head){
616     printf("create character function called\n");
617     printf("character name: %s\n", name);
618     printf("character value: %c\n", *value);
619 
620         printf("top: %d\n", *top);
621 
622     // Check if the stack is empty
623     if (*top == -1) {
624         printf("stack is empty\n");
625         return;
626     }
627 
628     // Update the frame_head
629     printf("frame_head: %d\n", *frame_head);
630     *frame_head -= sizeof(char);
631     printf("frame_head: %d\n", *frame_head);
632 
633     // Create the CHARCTER variable at the LOCTAION OF frame_head
634     // sprintf(&memory[(*frame_head)], "%c", *value);
635     memcpy(&memory[(*frame_head)], value, sizeof(char));
636     printf("frame_head3: %d\n", *frame_head);
637 
638     printf("Character variable '%s' created at address %d with value %c\n", name,
    *frame_head, *value);
639     return;
640     printf("\n");
641 }
642 
643 // print the stack or heap for debugging purposes
644 void print(char *memory){
645     printf("print function called\n");
646     // prints hex values of memory
647     for(int i = 0; i < MEMSIZE; i++){
648         printf("%d = %x\n", i, memory[i]);
649     }
650     printf("\n");
651 
652     // uncomment if using sprintf
653     // printf("==============================For Checking Purposes
    ============================\n");
654     // To see decimal values of memory
```

```
655     // for(int i = 0; i < MEMSIZE; i++){
656     //     printf("%d = %d\n", i, memory[i]);
657     // }
658     // printf("\n");
659     // To see characters of memory
660     // for (int i = 0; i < MEMSIZE; i++) {
661     //     printf("%d = %c  ", i, memory[i]);
662     //     if ((i + 1) % 10 == 0) {
663     //         printf("\n");
664     //     }
665     // }
666     printf("\n");
667 }
668 
669 // freelist start, size, next
670 // allocated address, name, next
671 // create a buffer on the heap
672 
673 // Heap cannot be created without frame
674 void create_buffer_heap(char* name, int* size, struct freelist** head, struct
        allocated** head_allocated, char* memory, int* frame_head, int *top) {
675     printf("create buffer function called\n");
676     printf("buffer name: %s\n", name);
677     printf("buffer size: %d\n", *size);
678 
679 
680     // Finding free node in the freelist
681     struct freelist* temp = *head;
682     while (temp->size < (*size) + 8) {
683         temp = temp->next;
684         if (temp == NULL) {
685             printf("Error: Not enough free space in heap.\n");
686             return;
687         }
688     }
689 
690     // Checking if frame is open
691     if(*top == -1){
692         printf("Error: No frame is open. No function created\n");
693         return;
694     }
695 
696 
697     // Updating allocated list
698     (*head_allocated)->startaddress = temp->start;
699     append_allocated(head_allocated, name, (*head_allocated)->startaddress);
700     printf("Allocated list: \n");
701     printAllocated(*head_allocated);
702     printf("\n");
703 
704     // updating frame
705     int store = (*head_allocated)->startaddress;
706     *frame_head -= sizeof(int);
707     memcpy(&memory[*frame_head], &store, sizeof(int));
708 
709     // Updating free list
710     temp->size = temp->size - (*size) - 8;
711     temp->start = temp->start + (*size) + 8;
712     printf("Free list: \n");
713     printFreelist(*head);
714     printf("\n");
715 
```

```
716     // Create the buffer
717     int magic_no = rand() % 1000;
718     char* str = (char*)malloc(*size);
719     srand(time(NULL));
720     for (int i = 0; i < *size - 1; i++) {
721         str[i] = 'A' + (rand() % 26);
722         printf("%c", str[i]);
723     }
724     printf("\n");
725 
726     // Update the memory buffer. Remove comment from lines if sprintf if u want to
        view the memory for checking
727     // sprintf(&memory[store], "%d", *size);
728     memcpy(&memory[store], size, sizeof(int));
729     // sprintf(&memory[store + 4], "%d", magic_no);
730     memcpy(&memory[store + 4], &magic_no, sizeof(int));
731     // sprintf(&memory[store + 8], "%s", str);
732     memcpy(&memory[store + 8], str, *size);
733 
734     free(str);  // Free dynamically allocated memory
735 
736     printf("\n");
737 
738     return;
739 }
740 
741 
742 // delete a buffer from the heap
743 void delete_buffer_heap(char *name, struct freelist **head, struct allocated **
        head_allocated, char *memory, int *frame_head)
744 {
745     printf("delete buffer function called\n");
746     printf("buffer name: %s\n", name);
747 
748     // Finding the buffer in the allocated list
749     struct allocated *temp = *head_allocated;
750     if (temp == NULL)
751     {
752         printf("Error: Allocated list is empty.\n");
753         return;
754     }
755     struct allocated *temp1 = NULL;
756     while (strcmp(temp->name, name) != 0)
757     {
758         temp1 = temp; // buffer before the buffer to be deleted in the list
759         temp = temp->next; // buffer to be deleted
760         if (temp == NULL)
761         {
762             printf("Error: Buffer to delete not found in heap\n");
763             return;
764         }
765     }
766 
767     // Updating allocated list
768     if (temp1 != NULL){
769         temp1->next = temp->next;
770     }
771     else{
772         *head_allocated = temp->next;
773     }
774     temp->next = NULL;
775     printf("Allocated list after deallocating buffer: \n");
```

```
776     printAllocated(*head_allocated);
777 
778     // Updating free list
779     int size;
780     memcpy(&size, &memory[temp->startaddress], sizeof(int)); // fetching size of
        buffer to be deleted from its metadata
781     struct freelist *temp2 = (struct freelist *)malloc(sizeof(struct freelist));
782     if (temp2 == NULL){
783         printf("Memory allocation failed\n");
784         return;
785     }
786     struct freelist *temp3 = *head;
787     if (temp3 == NULL)
788     { // if freelist was empty
789         temp2->start = temp->startaddress;
790         temp2->size = size + 8;
791         temp2->next = NULL;
792         *head = temp2;
793         printf("Free list after deallocating buffer: \n");
794         printFreelist(*head);
795         printf("\n");
796         free(temp2); // Free the dynamically allocated temp2
797         return;
798     }
799     temp2->size = size + 8; // updating size of the first node in free list (
        including the 8 bytes of metadata)
800     temp2->start = temp->startaddress; // updating start address of the first node
         in free list
801     temp2->next = temp3;               // updating next of the first node in free
        list
802     *head = temp2;                     // updating head of free list
803     printf("Free list after deallocating buffer: \n");
804     printFreelist(*head);
805     printf("\n");
806 
807     // Updating memory
808     memset(&memory[temp->startaddress], '0', size + 8); // Zero out the memory of
        the deleted buffer
809 
810     printf("\n");
811 }
812 
813 
814 
815 int main() {
816 
817 
818     /*
819      START INITIALIZATION
820     */
821     char command[3];           // array to store the command
822     char name[MAX_NAME + 1]; // variable to store the function or variable name
823     int address; // variable to store the function address
824     int val; // variable to store the integer value
825     double dval; // variable to store the double value
826     char cval; // variable to store the character value
827     char memory[MEMSIZE];     // Buffer that will emulate stack and heap memory
828     struct framestatus framestatus_array[MAX_FRAMES];     // array of framestatus
        structures (size should be 5*21 = 105)
829     memcpy(&memory[394], framestatus_array, sizeof(framestatus_array));
830     int top = -1;
831     struct framestatus fs;
```

```
832     struct freelist freelist[300];      // array of freelist structures (size
        should be 500*12 = 6000)
833     int frame_head = 394;                   // pointer to the head of the current
        frame
834     struct allocated allocatedlist[300];
835     int currentstacksize = 200;
836     int currentheapsize = 100;
837 
838     // initializing memory buffer with '0'
839     for(int i = 0; i < 500; i++) {
840         memory[i] = '0';
841     }
842 
843     // initializing framestatus_array
844     for(int i = 0; i < 5; i++) {
845         framestatus_array[i].used = '0';
846         framestatus_array[i].number = -1;
847         framestatus_array[i].functionaddress = -1;
848         framestatus_array[i].frameaddress = -1;
849         strcpy(framestatus_array[i].name, "N/A");
850         printf("size Of %ld\n", sizeof(framestatus_array[i]));
851     }
852     memcpy(&memory[394], framestatus_array, sizeof(framestatus_array));
853 
854 
855     // initializing freelist
856     for(int i = 0; i < 300; i++) {
857         freelist[i].start = -1;
858         freelist[i].size = 0;
859         freelist[i].next = NULL;
860     }
861 
862     struct freelist* head;
863     struct freelist freeallocated;
864     head = &freeallocated;
865 
866 
867     head->start = 0;
868     head->size = 300;
869     head->next = NULL;
870 
871 
872     // Initializing allocated list
873     for(int i = 0; i < 300; i++) {
874         strcpy(allocatedlist[i].name, "N/A");
875         allocatedlist[i].startaddress = -1;
876         allocatedlist[i].next = NULL;
877     }
878 
879     struct allocated* head_allocated = NULL;
880     struct allocated freeallocated_allocated;
881     head_allocated = &freeallocated_allocated;
882 
883 
884 
885     /*
886      END INITIALIZATION
887     */
888 
889     while (1) {
890     printf("prompt>");
891     scanf("%s", command); // read the command
```

```
892
893     if (strcmp(command, "CF") == 0) { // create frame command
894         if (scanf("%s %d", name, &address) != 2) {
895             printf("Invalid parameters for CF command\n");
896             continue;
897         }
898         else{
899             create_frame(name, address, framestatus_array, &top, memory, &
    frame_head); // call the create frame function
900         }
901     }
902     else if (strcmp(command, "DF") == 0) { // delete frame command
903         delete_frame(framestatus_array, memory, &frame_head, &top); // call the
    delete frame function
904     }
905     else if (strcmp(command, "CI") == 0) { // create integer command
906         if (scanf("%s %d", name, &val) != 2) {
907             printf("Invalid parameters for CI command\n");
908             continue;
909         }
910         else{
911             create_integer(name, &val, framestatus_array, &top, memory, &
    frame_head); // call the create integer function
912         }
913     }
914     else if (strcmp(command, "CD") == 0) { // create double command
915         if (scanf("%s %lf", name, &dval) != 2) {
916             printf("Invalid parameters for CD command\n");
917             continue;
918         }
919         else{
920             create_double(name, &dval, framestatus_array, &top, memory, &
    frame_head); // call the create double function
921         }
922     } else if (strcmp(command, "CC") == 0) { // create character command
923         if (scanf("%s %c", name, &cval) != 2) {
924             printf("Invalid parameters for CC command\n");
925             continue;
926         }
927         else{
928             create_character(name, &cval, framestatus_array, &top, memory, &
    frame_head); // call the create character function
929         }
930     }
931     else if (strcmp(command, "CH") == 0) { // print stack command
932         if (scanf("%s %d", name, &val) != 2) {
933             printf("Invalid parameters for CH command\n");
934             continue;
935         }
936         else{
937             create_buffer_heap(name, &val, &head, &head_allocated, memory, &
    frame_head, &top); // call the create buffer heap function
938         }
939     }
940     else if (strcmp(command, "DH") == 0) {
941         if (scanf("%s", name) != 1) {
942             printf("Invalid parameters for DH command\n");
943             continue;
944         }
945         else{
946             delete_buffer_heap(name, &head, &head_allocated, memory, &frame_head);
     // call the delete buffer heap function
```

```
947         }
948     }
949     else if (strcmp(command, "SM") == 0) { // print stack command
950         print(memory); // call the print stack function
951     }
952     else if (strcmp(command, "exit") == 0) {
953         break; // exit the program
954     }
955     else {
956         printf("Invalid command\n"); // print error message if the command is
    invalid
957     }
958 }
959     for(int i = 0; i < 5; i++){
960         if(framestatus_array[i].used == '1'){
961             printf("frame number: %d\n", framestatus_array[i].number);
962             printf("function name: %s\n", framestatus_array[i].name);
963             printf("function address: %d\n", framestatus_array[i].functionaddress)
    ;
964             printf("frame address: %d\n", framestatus_array[i].frameaddress);
965             printf("frame usage: %c\n", framestatus_array[i].used);
966             printf("\n");
967         }
968         else break;
969     }
970 
971     // Free the allocated blocks
972     free_allocated_list(head_allocated);
973 
974     return 0;
975 }
```