CSE 340 FALL 2021

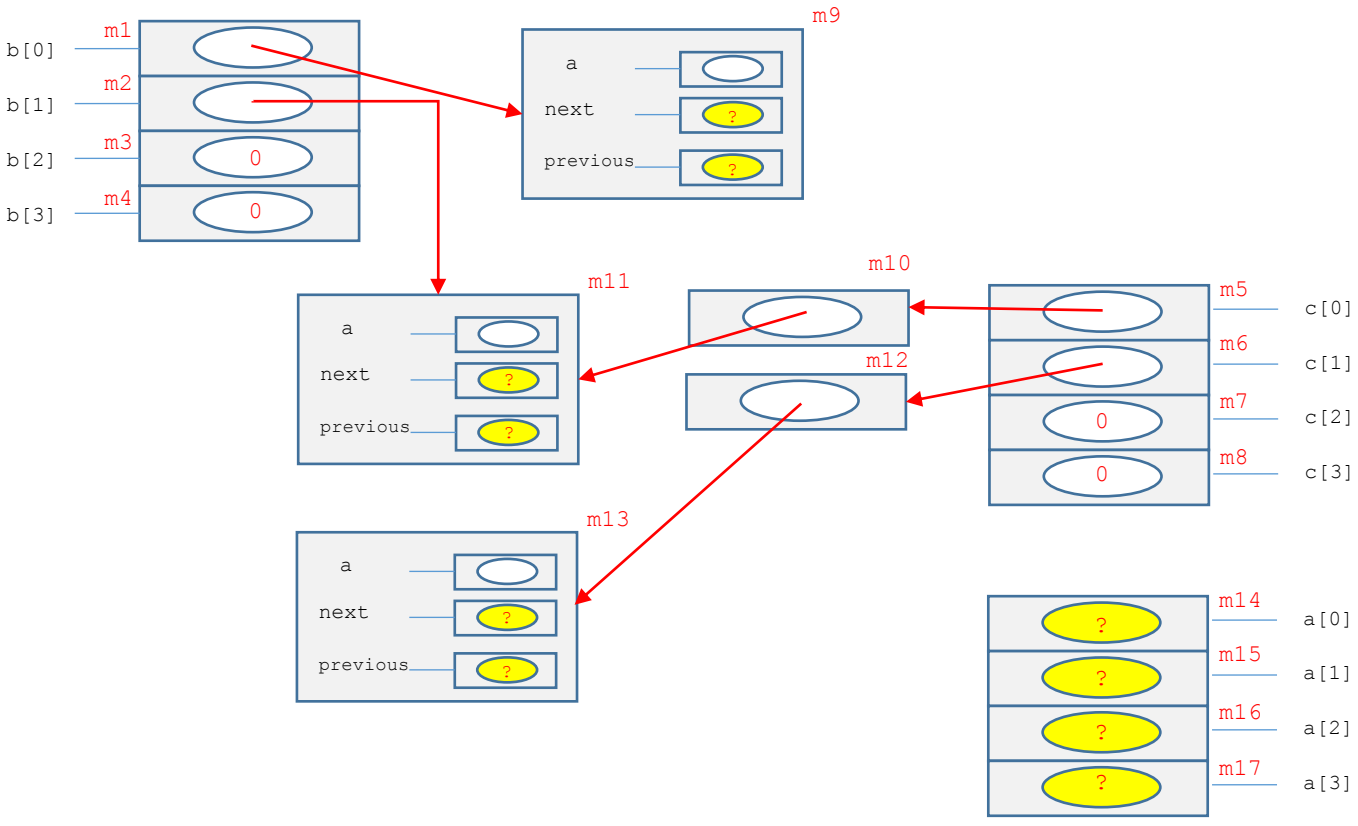
HW4

Problem 3 Solution

Prepared by Rida Bazzi

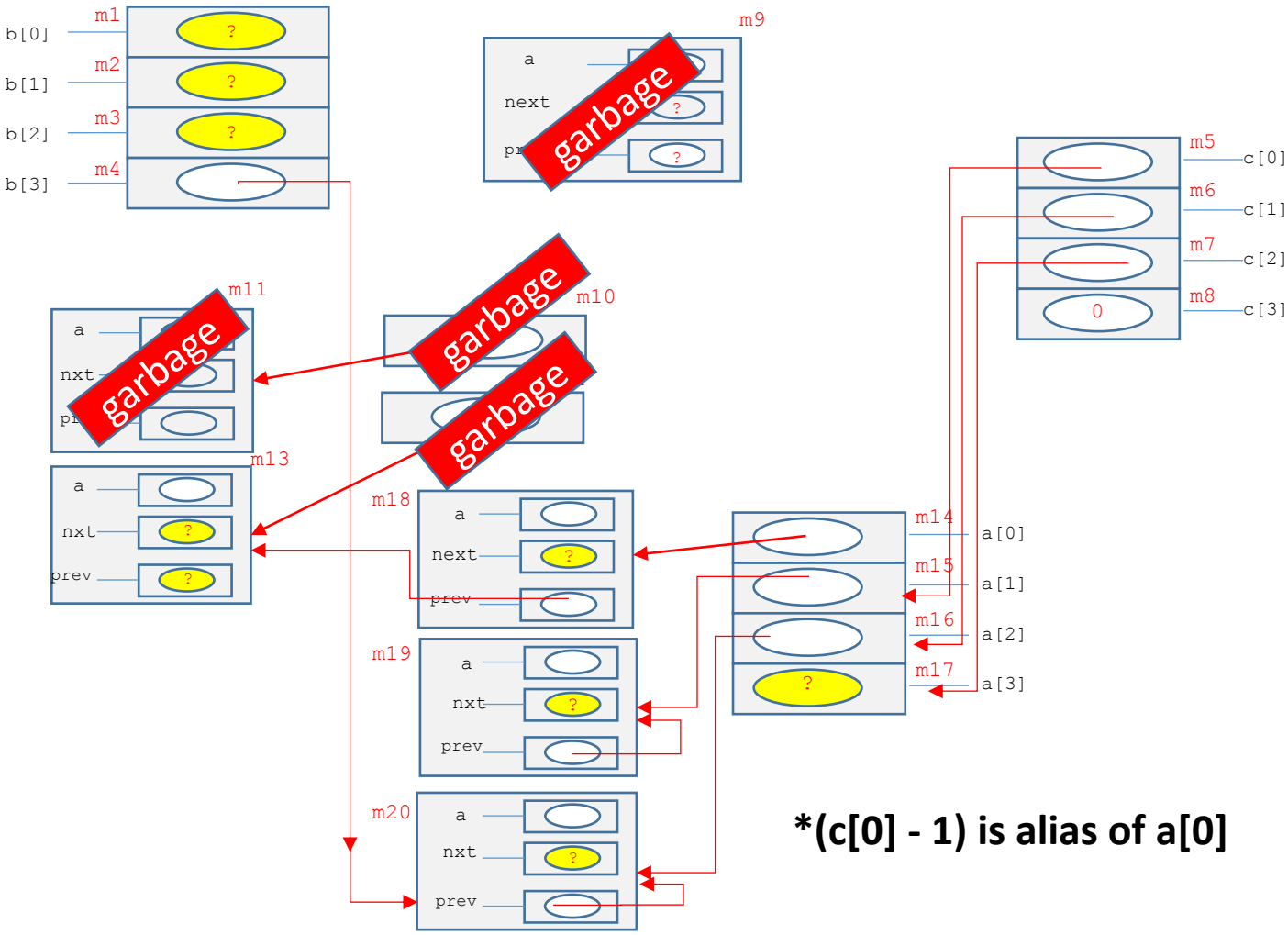
```c
#include <stdio.h>
#include <stdlib.h>

struct T {
        int i;
        struct T * next;
        struct T * previous;
};

struct T *b[4];         // Global variable. Locations m1 through m4 are
                        // associated with b[0] through b[3].
struct T **c[4];        // global variable. locations m5 through m8 are
                        // associated with c[0] through c[3].

int main()
{
   b[0] = (struct T *) malloc(sizeof(struct T));     // location m9 allocated
   c[0] = (struct T **) malloc(sizeof(struct T *));  // location m10 allocated
   *c[0] = (struct T *) malloc(sizeof(struct T));    // location m11 allocated
   c[1] = (struct T **) malloc(sizeof(struct T *));  // location m12 allocated
   *c[1] = (struct T *) malloc(sizeof(struct T));    // location m13 allocated
   b[1] = *c[0];

   { struct T *a[4]; // a[0] through a[3] are in locations m14 through m17
     // point 1
```

Question 1. Here is the box-circle diagram at point 1. There are no dangling references, no garbage locations and only 10 wild pointers (yellow circles below). Note that global variables are initialized to zero, so b[2] for example is not a wild pointer



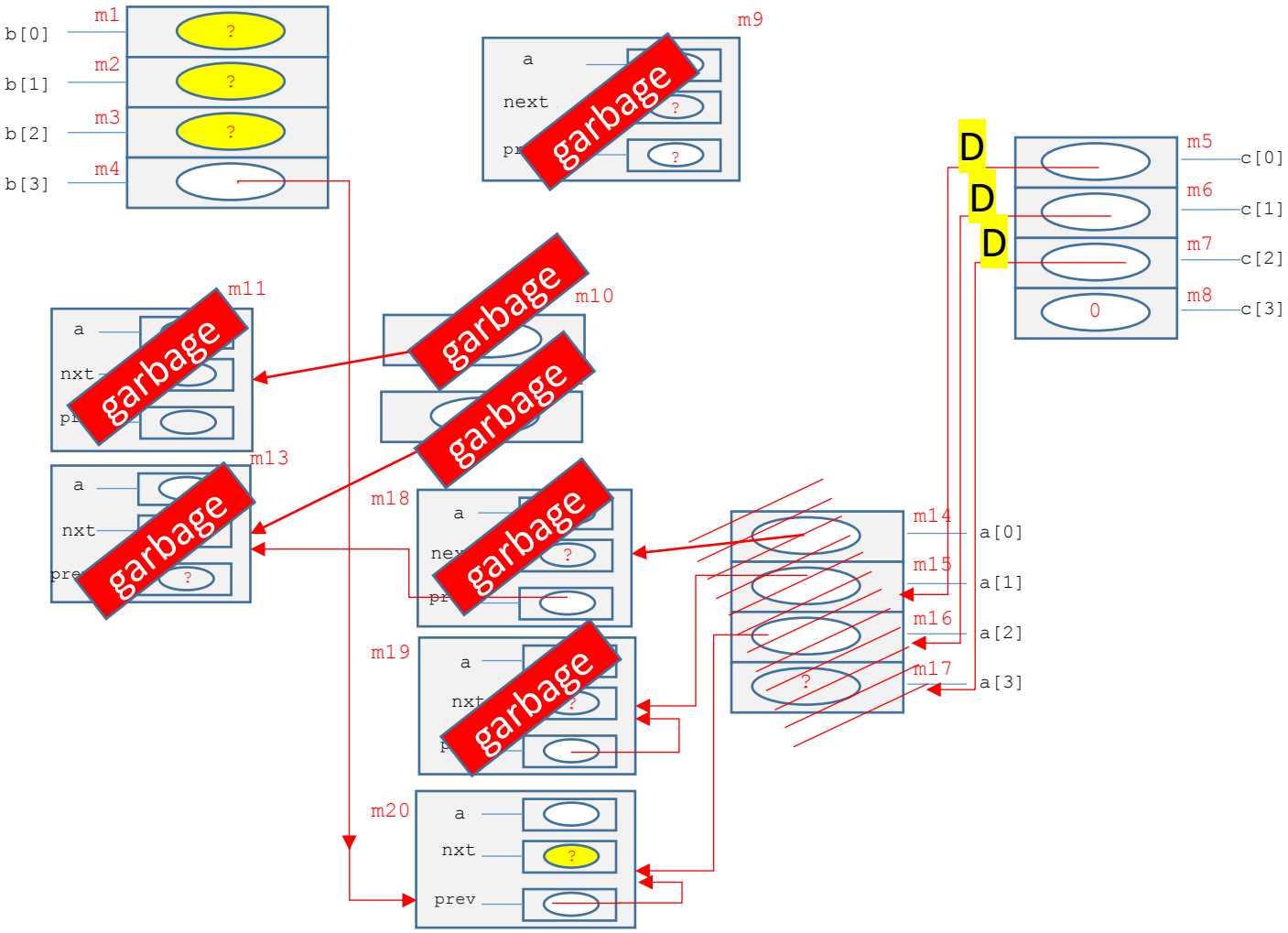
```
...

int main()
{
   b[0] = (struct T *) malloc(sizeof(struct T));      // location m9 allocated
   c[0] = (struct T **) malloc(sizeof(struct T *));  // location m10 allocated
   *c[0] = (struct T *) malloc(sizeof(struct T));     // location m11 allocated
   c[1] = (struct T **) malloc(sizeof(struct T *));  // location m12 allocated
   *c[1] = (struct T *) malloc(sizeof(struct T));     // location m13 allocated
   b[1] = *c[0];

   { struct T *a[4]; // a[0] through a[3] are in locations m14 through m17
     // point 1
     for (int i = 0; i < 3; i++)
     {
           a[i] = (struct T *) malloc(sizeof(struct T));// locations m18 through
                                                        // m20 allocated in
                                                        // successive iterations

           b[i+1] = a[i];
           c[i]   = &a[i+1];
           b[i]   = *c[i];
           a[i]->next = b[i];
           a[i]->previous= *c[abs(i-1)];      // abs() is the absolute value
     }
     // point 2
```

Point 2. Below is the box-circle diagram at point 2. There are no dangling references. m9, m10, m11 and m12 are garbage. The wild pointers are shown in yellow highlight with ?



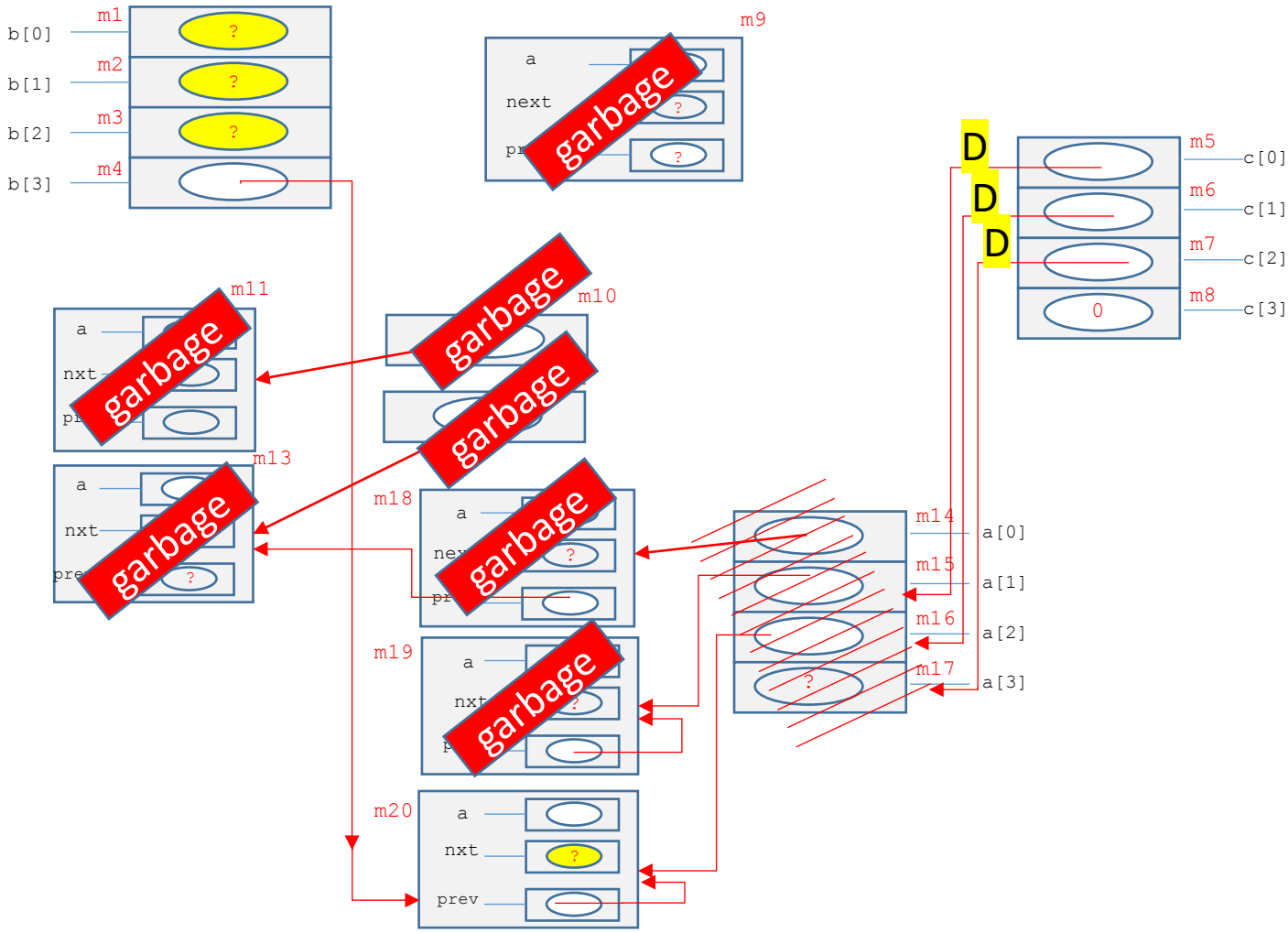
```
...
   *c[0] = (struct T *) malloc(sizeof(struct T));    // location m11 allocated
   c[1] = (struct T **) malloc(sizeof(struct T *));  // location m12 allocated
   *c[1] = (struct T *) malloc(sizeof(struct T));    // location m13 allocated
   b[1] = *c[0];

   { struct T *a[4]; // a[0] through a[3] are in locations m14 through m17
     // point 1
     for (int i = 0; i < 3; i++)
     {
           a[i] = (struct T *) malloc(sizeof(struct T));// locations m18 through
                                                        // m20 allocated in
                                                        // successive iterations
           b[i+1] = a[i];
           c[i]   = &a[i+1];
           b[i]   = *c[i];
           a[i]->next = b[i];
           a[i]->previous= *c[abs(i-1)];      // abs() is the absolute value
     }
     // point 2
  }
  // point 3
```

Point 3. Below is the box-circle diagram at point 3. At point 3, a[] is deallocated, the dangling references (**D**) and garbage locations are shown below. The wild pointer are shown in the oval highlighted in yellow ? **c[0],c[1] and c[2] are dangling**



```
...
   *c[0] = (struct T *) malloc(sizeof(struct T));    // location m11 allocated
   c[1] = (struct T **) malloc(sizeof(struct T *));  // location m12 allocated
   *c[1] = (struct T *) malloc(sizeof(struct T));    // location m13 allocated
   b[1] = *c[0];

   { struct T *a[4]; // a[0] through a[3] are in locations m14 through m17
     // point 1
     for (int i = 0; i < 3; i++)
     {
           a[i] = (struct T *) malloc(sizeof(struct T));// locations m18 through
                                                        // m20 allocated in
                                                        // successive iterations
           b[i+1] = a[i];
           c[i]   = &a[i+1];
           b[i]   = *c[i];
           a[i]->next = b[i];
           a[i]->previous= *c[abs(i-1)];      // abs() is the absolute value
     }
     // point 2
  }
  // point 3
 free(b[2]); // point 4
```

Point 4. Below is the box-circle diagram at point 4. At point 4, free(b[2]) is executed, but b[2] is not pointing to a properly allocated memory location. In general, this has undefined behavior. Here I show the situation unchanged from the way it was at point 3.



```
...
c[0] = &b[0];    // assignment 1
c[1] = &b[1];    // assignment 2
*c[0] = *c[1];    // assignment 3


//point 5
```

Statement 1 results in an arrow from m5 to m1
Statement 2 results in an arrow from m6 to m2
Statement 3 results in copying the value from location m2 to location m1. These values are wild pointers to start with and the assignment just make them the same value

Point 5. Below is the box-circle diagram at point 5. Statement 1 and Statement 2 reduced the dangling references to 1, but otherwise the situation is unchanged from what it was at point 4.