Learning C by example

C programs that I have found useful during my studies

Tag Archives: C interview questions

Semaphores

This is a problem from an interview I found online. There are three lists (implemented as array in my solution), and three threads. Each thread access one list, and prints one number from the list. The threads must sync to print in this order: T1 -> T2 -> T3 -> T1 -> T2, …

The synchronization method used is semaphores. Semaphores are usually used to sync multiple threads. One thread can let another run by posting a semaphore for the other thread.

This is a solution with pthreads:

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <semaphore.h>
 4
 5  #define N 10
 6  sem_t sem1, sem2, sem3;
 7  int a1[N], a2[N], a3[N];
 8
 9  void * thread1(void *arg) {
10      int i;
11      for (i = 0; i< N; i++) {
12          sem_wait(&sem1);
13          printf("%d, ", a1[i]);
14          sem_post(&sem2);
15      }
16      pthread_exit(0);
17  }
18
```

```
19  void * thread2(void *arg) {
20      int i;
21      for (i = 0; i< N; i++) {
22          sem_wait(&sem2);
23          printf("%d, ", a2[i]);
24          sem_post(&sem3);
25      }
26      pthread_exit(0);
27  }
28
29  void * thread3(void *arg) {
30      int i;
31      for (i = 0; i< N; i++) {
32          sem_wait(&sem3);
33          printf("%d, ", a3[i]);
34          sem_post(&sem1);
35      }
36      pthread_exit(0);
37  }
38
39  int main() {
40      pthread_t threads[3];
41      int i;
42
43     int no = 0;
44      for (i = 0; i < N; i++) {
45          a1[i] = no++;
46          a2[i] = no++;
47          a3[i] = no++;
48      }
49
50      sem_init(&sem1, 0, 1);
51      sem_init(&sem2, 0, 0);
52      sem_init(&sem3, 0, 0);
53
54      pthread_create(&threads[0], NULL, thread1, NULL);
55      pthread_create(&threads[1], NULL, thread2, NULL);
56      pthread_create(&threads[2], NULL, thread3, NULL);
57
58      for (i = 0; i < 3; i++)
59          pthread_join(threads[i], NULL);
60      printf("\n");
61
62      sem_destroy(&sem1);
63      sem_destroy(&sem2);
64      sem_destroy(&sem3);
65
66      return 0;
67  }
68
69
70  gcc -pthread threelists.c
71  ./a.out
72  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
```

```
73
74
```

Reverse order of words within string

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

#define SWAP(a, b) (a^=b, b^=a, a^=b)

void reversestring(char *a, char *b) {
    assert(a != NULL && b != NULL);
    /* while (a++ < b++) runs the while loop with
     * parameters incremented, but check condition before
     * parameters are incremented
     * while (++a < ++b) increments parameters before
     * checking condition
     * both ( ;i<max ; i++) and ( ;i<max; ++i) run the
     * loop and then increment the parameter. they are both
     * identical */
    while (a < b) {
        SWAP(*a, *b);
        a++; b--;
    }
}
void reverseorderwords(char *str) {
    assert(str != NULL);
    char * ptra = str;
    char * ptrb = str;

    while (*ptrb != '\0') {
        while (*ptrb != '\0' && *ptrb != ' ') {
            ptrb++;
        }
        reversestring(ptra, ptrb-1);
        if (*ptrb != '\0') {
            ptrb++;
            ptra = ptrb;
        }
    }
    reversestring(str, --ptrb);
}


int main() {
    //char * str = "this is a test"; // static string read-only.
```

```
                                        // will segfault when changed
    char str[] = "this is a test";
    printf("%s\n", str);
    reverseorderwords(str);
    printf("%s\n", str);
    return 0;
}


./a.out
this is a test
test a is this
```

Linked List interview problems

Collection of linked list interview problems implemented in C:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


#define N 19 // hash table size
#define EMPTY -1
#define TRUE 1
#define FALSE 0

struct node {
    struct node * next;
    int value;
};
typedef struct node node_t;


int haskey(int table[N][5], int value) {
    int key = value % N;
    int i = 0;
    while (table[key][i] != -1) {
        if (table[key][i] == value)
            return TRUE;
        i++;
    }
    return FALSE;
}
```

```
29
30  void insertvalue(int table[][5], int value) {
31      int key = value % N;
32      int i = 0;
33      // checking if collision
34      while (table[key][i] != -1) {
35          i++;
36          if (i == 5) // too many collisions
37              return; // discard value
38      }
39      // insert value at key position
40      table[key][i] = value;
41      if (i < 4)
42          table[key][++i] = -1;
43  }
44
45  /*
46   * remove duplicates in unsorted list
47   */
48  void remove_duplicates(node_t * list) {
49      /* solution 1: sweep in two nested loos (O(N2))
50         solution 2: put elements in buffer (O(N)) and
51         sort elements (O(N*logN)) and sweep list (O(N)),
52         checking if element in buffer (O(logN). Total O(NlogN)
53         solution 3: use hash or bit array. O(N)
54      */
55      if (list == NULL || list->next == NULL)
56          return;
57
58      /* create hash table */
59      int i;
60      int hashtable[N][5]; // using array for collisions (max 5 key colli
61      // for (i = 0; i < N; i++)
62      //    hashtable[i][0] = EMPTY; // -1 indicates no value for the key
63      memset(hashtable, EMPTY, sizeof(hashtable));
64
65      /* sweep list */
66      insertvalue(hashtable, list->value); // first element
67
68      node_t * current = list->next;
69      node_t * prev = list;
70      node_t * duplicate;
71      while (current != NULL) {
72          if (haskey(hashtable, current->value)) {
73              duplicate = current;
74              /* skip node */
75              prev->next = current->next;
76              /* move current, prev doesnt move */
77              current = current->next;
78              /* free duplicate node */
79              free(duplicate);
80          } else {
81              insertvalue(hashtable, current->value);
82              /* move forward */
```

```
            prev = current;
            current = current->next;
        }
    }
}

void insert_beginning(node_t ** list, int value) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode->value = value;
    // newnode->next = NULL;
    newnode->next = *list;
    *list = newnode;
}

void print_list(node_t * list) {
    while (list != NULL) {
        printf("%d, ", list->value);
        list = list->next;
    }
    printf("\n");
}

/*
 * Remove node of list having access to only that node
 */
void remove_node(node_t *node) {
    node_t *next = node->next;
    memcpy(node, node->next, sizeof(node_t));
    free(next);
}

node_t * nodefromend(node_t *list, int k) {
    node_t *first = list;
    node_t *second = list;
    int i = 0;
    while (first != NULL && i < k) {
        first = first->next;
        i++;
    }
    if (first == NULL)
        return NULL;
    while (first != NULL) {
        first = first->next;
        second = second->next;
    }
    return second;
}


/*
 * detect if list has loop
 */
int detectloop(node_t *list) {
    /* using slow ptr and fast ptr that
```

```
137         advances two nodes at a time. If there is
138         loop, they both will get into loop, and eventually
139         fast ptr will point reach slow ptr and point to
140         same node
141      */
142     if (list == NULL)
143         return FALSE;
144
145
146     node_t *slow = list;
147     node_t *fast = list;
148     while (fast->next->next != NULL) {
149         fast = fast->next->next;
150         slow = slow->next;
151         if (fast == slow)
152             return TRUE;
153     }
154     return FALSE;
155 }
156
157 int ispalindrome(node_t * list) {
158
159     if (list == NULL || list->next == NULL)
160         return FALSE;
161
162     int queue[40];
163     int index = 0;
164     node_t * fast = list;
165     node_t * slow = list;
166     /* traverse list up to middle and put elements
167        in a queue. use fast/slow method to stop at
168        middle */
169     while (fast != NULL && fast->next != NULL &&\
170            fast->next->next != NULL) {
171         queue[index++] = slow->value;
172         slow = slow->next;
173         fast = fast->next->next;
174     }
175     /* if list odd lenght, skip element in middle */
176     if (fast->next != NULL)
177         slow = slow->next;
178
179     /* move slow to end, and compare with LIFO queue */
180     while (slow != NULL) {
181         if (slow->value != queue[--index])
182             return FALSE;
183         slow = slow->next;
184     }
185     return TRUE;
186 }
187
188 int main() {
189
190     node_t * list = NULL;
```

```
191         int a[8] = {4, 2, 7, 4, 3, 7, 9, 2};
192         int i;
193         for (i = 0; i < 8; i++) {
194             insert_beginning(&list, a[i]);
195             print_list(list);
196         }
197         /* remove duplicate nodes in unsorted list */
198         remove_duplicates(list);
199         print_list(list);
200 
201         /* return node k nodes from the end */
202         node_t * node = nodefromend(list, 3);
203         assert(node!=NULL);
204         printf("%d\n", node->value);
205 
206         /* remove a node, give ptr to that node only */
207         remove_node(node);
208         print_list(list);
209 
210         /* detect if there is loop */
211         if (detectloop(list))
212             printf("Loop detected\n");
213         else
214             printf("Loop not detected\n");
215 
216         /* check if list is palindrome */
217         if (ispalindrome(list))
218             printf("List is palindrome\n");
219         else
220             printf("List is not palindrome\n");
221 
222         return 0;
223     }
224 
225 
226 
227     4,
228     2, 4,
229     7, 2, 4,
230     4, 7, 2, 4,
231     3, 4, 7, 2, 4,
232     7, 3, 4, 7, 2, 4,
233     9, 7, 3, 4, 7, 2, 4,
234     2, 9, 7, 3, 4, 7, 2, 4,
235     2, 9, 7, 3, 4,
236     7
237     2, 9, 3, 4,
238     Loop not detected
239     List is not palindrome
```

all types of linked lists

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


/* node for single linked list */
typedef struct node {
    struct node *next;
    int value;
} node_ts;

/* node for double linked list */
typedef struct noded {
    struct noded *next;
    struct noded *prev;
    int value;
} node_td;


/*
 * Routines for single linked list
 */
void insert_single(node_ts ** list, int value) {
    node_ts * new = malloc(sizeof(node_ts));
    new->value = value;
    new->next = NULL;

    /* empty */
    if (*list == NULL) {
        *list = new;
        return;
    }
    /* element becomes first. can be merged with case above */
    if ((*list)->value > value) {
        new->next = *list;
        *list = new;
        return;
    }

    node_ts *prev = *list;
    node_ts *ptr = (*list)->next;
    while (ptr != NULL && ptr->value < value) {
        prev = ptr;
        ptr = ptr->next;
    }
```

```
47      prev->next = new;
48      new->next = ptr;
49  }
50
51  void print_single(node_ts *list) {
52      while (list != NULL) {
53          printf("%d, ", list->value);
54          list = list->next;
55      }
56      printf("\n");
57  }
58
59  void reverse_single(node_ts **list) {
60
61      /* empty or one element */
62      if (*list == NULL || (*list)->next == NULL) {
63          return;
64      }
65
66      node_ts * prev = NULL;
67      node_ts * current = *list;
68      node_ts * next = (*list)->next;
69
70      while (next != NULL) {
71          current->next = prev;
72          /* move ptrs forward */
73          prev = current;
74          current = next;
75          next = next->next;
76      }
77      /* current points to last element */
78      current->next = prev;
79      *list = current;
80  }
81
82  /*
83   * Routines for circular single linked list
84   */
85  void insert_singlecircular(node_ts ** list, int value) {
86
87      node_ts * new = malloc(sizeof(node_ts));
88      new->value = value;
89      new->next = NULL;
90
91      /* empty list */
92      if (*list == NULL) {
93          *list = new;
94          new->next = new;
95          return;
96      }
97
98      /* place at first location */
99  #ifdef TRAVERSE
100     if ((*list)->value > value) {
```

```
        /* if list has only one node */
        if ((*list)->next == *list) {
            new->next = *list;
            (*list)->next = new;
            *list = new;
            return;
        }
        /* if list has more nodes, get last node */
        node_t *ptr = (*list)->next;
        while (ptr->next != *list) {
            assert(ptr->next != NULL);
            ptr = ptr->next;
        }
        /* insert node */
        ptr->next = new;
        new->next = *list;
        *list = new;
        return;
    }
#else
    /* another way to insert at beginning
       without having to traverse list */
    if ((*list)->value > value) {
        memcpy(new, *list, sizeof(node_t));
        (*list)->value = value;
        (*list)->next = new;
        return;
    }
#endif


    node_ts *prev = *list;
    node_ts *ptr = (*list)->next;
    while (ptr != *list && ptr->value < value) {
        prev = ptr;
        ptr = ptr->next;
    }
    prev->next = new;
    new->next = ptr;
}

void print_singlecircular(node_ts * list) {
    node_ts * ptr = list;
    do {
        printf("%d, ", ptr->value);
        ptr = ptr->next;
    } while (ptr != list);
    printf("\n");
}

void reverse_singlecircular(node_ts ** list) {

    if (*list == NULL || (*list)->next == *list)
        return;
```

```
155
156     node_ts * prev = *list;
157     node_ts * current = (*list)->next;
158     node_ts * next = current->next;
159
160     while(current != *list) {
161         current->next = prev;
162         prev = current;
163         current = next;
164         next = next->next;
165     }
166     /* prev points to last element,
167        current points to first */
168     current->next = prev;
169     *list = prev;
170 }
171
172
173
174 /*
175  * Routines for double linked list
176  */
177 void insert_double(node_td ** list, int value) {
178     node_td * new = malloc(sizeof(node_td));
179     new->value = value;
180     new->prev = NULL;
181     new->next = NULL;
182
183     /* empty list */
184     if (*list == NULL) {
185         *list = new;
186         return;
187     }
188
189     /* first place */
190     if ((*list)->value > value) {
191         new->next = *list;
192         (*list)->prev = new;
193         *list = new;
194         return;
195     }
196
197     node_td *ptr = *list;
198     /* looking one node ahead */
199     while (ptr->next != NULL && ptr->next->value < value) {
200         ptr = ptr->next;
201     }
202
203     /* place at end of list */
204     if (ptr->next == NULL) {
205         ptr->next = new;
206         new->prev = ptr;
207         return;
208     }
```

```
209
210     /* place new node */
211     new->prev = ptr;
212     new->next = ptr->next;
213     /* adjust previous node */
214     ptr->next = new;
215     /* adjust next node */
216     new->next->prev = new;
217 }
218
219 void reverse_double(node_td **list) {
220     /* empty or one element */
221     if (*list == NULL || (*list)->next == NULL) {
222         return;
223     }
224
225     node_td * ptr = *list;
226     node_td * next = (*list)->next;
227
228     while (next != NULL) {
229         /* reverse prev and next pointers */
230         ptr->next = ptr->prev;
231         ptr->prev = next;
232         /* move forward */
233         ptr = next;
234         next = next->next;
235     }
236     /* make last element the first one */
237     ptr->next = ptr->prev;
238     ptr->prev = NULL;
239     *list = ptr;
240 }
241
242 void print_double(node_td * list) {
243     while (list != NULL) {
244         printf("%d, ", list->value);
245         list = list->next;
246     }
247     printf("\n");
248 }
249
250
251 /*
252  * Routines for double circular linked list
253  */
254 void insert_double_circular(node_td ** list, int value) {
255     node_td * new = malloc(sizeof(node_td));
256     new->value = value;
257     new->next = NULL;
258     new->prev = NULL;
259
260     /* empty list */
261     if (*list == NULL) {
262         *list = new;
```

```
263             new->prev = new;
264             new->next = new;
265             return;
266         }
267 
268         /* insert at beggining */
269         if (value < (*list)->value) {
270             node_td * prev = (*list)->prev;
271             /* insert new node */
272             new->next = *list;
273             new->prev = (*list)->prev;
274             /* adjust next (previoulsy first node) */
275             (*list)->prev = new;
276             /* adjust previous (previously last node) */
277             prev->next = new;
278             /* adjust list pointer */
279             *list = new;
280             return;
281         }
282 
283 
284         node_td * ptr = (*list)->next;
285         node_td * prev;
286         while (ptr != *list && ptr->value < value) {
287             ptr = ptr->next;
288         }
289         prev = ptr->prev;
290 
291         prev->next = new;
292         ptr->prev = new;
293         new->next = ptr;
294         new->prev = prev;
295     }
296 
297     void print_double_circular(node_td * list) {
298 
299         if (list == NULL) {
300             printf("\n");
301             return;
302         }
303         node_td *ptr = list;
304         do {
305             printf("%d, ", ptr->value);
306             ptr = ptr->next;
307         } while (ptr != list);
308         printf("\n");
309     }
310 
311     void reverse_double_circular(node_td ** list) {
312 
313         if (*list == NULL || (*list)->next == *list)
314             return;
315 
316         node_td *ptr = *list;
```

```
317         node_td *next = ptr->next;
318         while(next != *list) {
319             /* switch next and prev pointers */
320             ptr->next = ptr->prev;
321             ptr->prev = next;
322             /* move forward */
323             ptr = next;
324             next = next->next;
325         }
326         /* last element becomes first */
327         ptr->next = ptr->prev;
328         ptr->prev = next;
329 
330         *list = ptr;
331     }
332 
333     int main() {
334         int a[5] = {3, 1, 6, 5, 9};
335         int i;
336 
337         // --------
338         printf("testing single linked list\n");
339         node_ts * singlell = NULL;
340         for (i = 0; i<5; i++) {
341             insert_single(&singlell, a[i]);
342             print_single(singlell);
343         }
344         reverse_single(&singlell);
345         print_single(singlell);
346 
347         // ----------
348         printf("testing double linked list\n");
349         node_td * doublell = NULL;
350         for (i = 0; i<5; i++) {
351             insert_double(&doublell, a[i]);
352             print_double(doublell);
353         }
354         reverse_double(&doublell);
355         print_double(doublell);
356 
357         // ---------
358         printf("testing single circular linked list\n");
359         node_ts * singlec = NULL;
360         for (i = 0; i<5; i++) {
361             insert_singlecircular(&singlec, a[i]);
362             print_singlecircular(singlec);
363         }
364         reverse_singlecircular(&singlec);
365         print_singlecircular(singlec);
366 
367         // ----------
368         printf("testing double circular linked list\n");
369         node_td * doublecirculardll = NULL;
370         for (i = 0; i<5; i++) {
```

```
371             insert_double_circular(&doublecircularll, a[i]);
372             print_double_circular(doublecircularll);
373         }
374         reverse_double_circular(&doublecircularll);
375         print_double_circular(doublecircularll);
376 
377 
378         return 0;
379     }
380 
381 
382 
383     testing single linked list
384     3,
385     1, 3,
386     1, 3, 6,
387     1, 3, 5, 6,
388     1, 3, 5, 6, 9,
389     9, 6, 5, 3, 1,
390     testing double linked list
391     3,
392     1, 3,
393     1, 3, 6,
394     1, 3, 5, 6,
395     1, 3, 5, 6, 9,
396     9, 6, 5, 3, 1,
397     testing single circular linked list
398     3,
399     1, 3,
400     1, 3, 6,
401     1, 3, 5, 6,
402     1, 3, 5, 6, 9,
403     9, 6, 5, 3, 1,
404     testing double circular linked list
405     3,
406     1, 3,
407     1, 3, 6,
408     1, 3, 5, 6,
409     1, 3, 5, 6, 9,
410     9, 6, 5, 3, 1,
```

Semaphores

The problem below synchronizes two threads, one prints odd numbers, and the other prints even numbers. In this solution each thread posts a semaphore to wake up the other thread when a number is printed. Since the same lock is acquire by a thread (wait), and released (post) by a different thread that did not own it, mutex cannot be used, since for a mutex the same thread must acquire it and release it.

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

/* two threads, one prints odd numbers,
the other even numbers, up to 10
*/

sem_t odd_lock;
sem_t even_lock;

void * even(void *arg) {
int a = 1;
while (a < 10) {
sem_wait(&even_lock);
printf("number: %d\n", a);
sem_post(&odd_lock);
a+=2;
}
pthread_exit(0);
}

void * odd(void *arg) {
int a = 2;
while (a < 10) {
sem_wait(&odd_lock);
printf("number: %d\n", a);
sem_post(&even_lock);
a+=2;
}
pthread_exit(0);
}

int main() {
pthread_t threads[2];
sem_init(&odd_lock, 0, 0);
sem_init(&even_lock, 0, 1);
pthread_create(&threads[0], NULL, even, NULL);
pthread_create(&threads[1], NULL, odd, NULL);
pthread_join(threads[0], NULL);
pthread_join(threads[1], NULL);
sem_destroy(&odd_lock);
sem_destroy(&even_lock);
return 0;
}

gcc -pthread sem.c

./a.out
number: 1
number: 2
number: 3
number: 4
```

```
number: 5
number: 6
number: 7
number: 8
number: 9
```

malloc memory at n-byte memory boundary

I was asked in a interview once to write a c function that uses malloc, but aligns the pointer to n-byte boundary. And then to write a function to free the memory for that pointer. The solution I am posting here calls malloc,, then aligns the pointer to n-byte memory boundary, and then it stores the padding used for alignment at the first bytes. In this way, the free function can retrieve the address for the original memory allocated by malloc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>  // for uintptr_t type, needed for boolean
                       // operations on pointers


/* Aligning pointer to  nbyte memory boundary
   padding = n - (offset & ( -1)) = -offset & (n-1)
   aligned offset = (offset + n-1) & ~(n-1)
 */
void * mallocaligned(size_t size, int align) {
    if (align < sizeof(int))
        align = sizeof(int);
    /* allocate pointer with space to store original address at top and
     * to move to align-byte boundary */
    void *ptr1 = malloc(size + align + align - 1);
    printf("%d bytes of memory allocated at %p\n", size+2*align-1, ptr1);
    /* align pointer to align-byte boundary */
    void *ptr2 = (void *)(((uintptr_t)ptr1 + align - 1) & ~(align-1));
    /* store there the original address from malloc */
    *(unsigned int *)ptr2 = (unsigned int)ptr1;
    /* move pointer to next align-byte boundary */
    ptr2 = ptr2 + align;
    printf("aligned memory at %p\n", ptr2);

    return ptr2;
}

void freealigned(void *ptr, int align) {
    /* move pointer back align bytes */
    ptr = (void *)((uintptr_t)ptr - align);
    /* retreive from there the original malloced pointer */
```

```
    ptr = (void *)(*(unsigned int *)ptr);
    printf("free memory at address %p\n", ptr);
    /* free that pointer */
    free(ptr);
}

int main() {
    void *ptr = mallocaligned(1000, 64);
    printf("allocated pointer at %p", ptr);
    freealigned(ptr, 64);
    return 0;
}
```

Linked list remove all duplicates

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

struct node {
    struct node *next;
    int value;
};
typedef struct node node_t;

void removeallduplicates(node_t *head) {
    node_t *ptr1 = head;
    node_t *current;
    node_t *prev;
    node_t *deleteit;
    while (ptr1 != NULL) {
        current = ptr1->next;
        prev = ptr1;
        while (current != NULL) {
            if (current->value == ptr1->value) {
                deleteit = current;
                prev->next = current->next;
                current = current->next;
                free(deleteit);
            } else {
                prev = current;
                current = current->next;
            }
        }
        ptr1 = ptr1->next;
    }
}

```

```c
void push(node_t **head, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode->value = value;
    newnode->next = *head;
    *head = newnode;  // newnode becomes the new head
}

void printlist(node_t *head) {
    node_t *ptr = head;
    while (ptr != NULL) {
        printf("%d, ", ptr->value);
        ptr = ptr->next;
    }
    printf("\n");
}

int main() {

    node_t * headptr = NULL;
    int numbs[] = {1,2,3,4,5,1,5,2,4,3};
    int i;
    for (i = 0; i < 10; i++)
        push(&headptr, numbs[i]);
    printlist(headptr);

    removeallduplicates(headptr);
    printlist(headptr);

    return 0;
}


> gcc linkedlistremovedup.c  -g
> ./a.out
3, 4, 2, 5, 1, 5, 4, 3, 2, 1,
3, 4, 2, 5, 1,
```

Linked list functions with dummy head and without it

Some linked list routines need to modify the head node (first node) of the list, if for example the head node is a new node, or the head node needs to be removed. This means that these routines need to modify the pointer to the head node, to point to a different node. In this cases, these routines can either return the new head node pointer, so that the function calling can update it, or get a pointer to the head node pointer, so that they can modify the head node pointer without any problems.

A different approach for this is to use a dummy head, which next pointer points to the head node. In this way the list functions can just pass a pointer to the dummy head, instead of a pointer to the head node pointer.

The following program illustrates both approaches:

```
#include &amp;lt;stdio.h&amp;gt;
#include &amp;lt;stdlib.h&amp;gt;

struct node {
    struct node *next;
    int value;
};
typedef struct node node_t;

void insert_ordered_dummy(node_t * dummy, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode->value = value;
    newnode->next = NULL;
    if (dummy->next == NULL) {
        dummy->next = newnode;
        return;
    }
    node_t * current = dummy->next;
    node_t * prev = dummy;
    while (current != NULL && current->value < value) {
        prev = current;
        current = current->next;
    }
    prev->next = newnode;
    newnode->next = current;
}

void insert_ordered_nodummy(node_t ** list, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode->next = NULL;
    newnode->value = value;

    // *list->value is equivalent to *(list->value)
    if (*list == NULL || (*list)->value > value) {
        newnode->next = *list;
        *list = newnode;
        return;
    }
```

```
    node_t * prev = *list;
    node_t * current = (*list)->next;
    while (current != NULL && current->value < value) {
        prev = current;
        current = current->next;
    }
    prev->next = newnode;
    newnode->next = current;
}
void insertend_nodummy(node_t **head, int value) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode-&amp;gt;value = value;
    newnode-&amp;gt;next = NULL;

    if (*head == NULL) {
        *head = newnode; // if list empty, newnode is head
        return;
    }

    node_t *ptr = *head;
    while (ptr-&amp;gt;next != NULL) { // move ptr to last node
        ptr = ptr-&amp;gt;next;
    }
    ptr-&amp;gt;next = newnode; // insert after last node
}

void insertbeginning_withdummy(node_t *dummyhead, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode-&amp;gt;value = value;
    newnode-&amp;gt;next = dummyhead-&amp;gt;next;
    dummyhead-&amp;gt;next = newnode;  // now dummy head points to new node
}
void insertend_withdummy(node_t *dummyhead, int value) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode-&amp;gt;value = value;
    newnode-&amp;gt;next = NULL;

    if (dummyhead-&amp;gt;next == NULL) {
        dummyhead-&amp;gt;next = newnode;
        return;
    }

    node_t *ptr = head;
    while (ptr-&amp;gt;next != NULL) { // move ptr to last node
        ptr = ptr-&amp;gt;next;
    }
```

```
    ptr-&amp;gt;next = newnode; // insert after last node
}

void printlist(node_t *head) {
    node_t *ptr = head;
    while (ptr != NULL) {
        printf(&amp;quot;%d, &amp;quot;, ptr-&amp;gt;value);
        ptr = ptr-&amp;gt;next;
    }
    printf(&amp;quot;\n&amp;quot;);
}

int main() {

    /* without dummy node: the first node is the head. main keeps a pointer
      to the first node (head). If a function needs to modify the head of
      the list (say a new node replaces the head node), then the head pointe
      needs to be modified, which means that a pointer to the head pointer n
      to be passed to the function. Or otherwise the function could return t
      new head pointer */
    node_t * headptr = NULL; // dont forget NULL!!
    int numbs[] = {1,2,3,4,5};
    int i;
    for (i = 0; i &amp;lt; 5; i++) {
        insertbeginning_nodummy(&amp;amp;headptr, numbs[i]);
        printlist(headptr);
    }
    for (i = 0; i &amp;lt; 5; i++) {
        insertend_nodummy(&amp;amp;headptr, numbs[i]);
        printlist(headptr);
    }


    /* with dummy node: in this case you can just pass pointer to
       dummy head to any function, and the function will be able to
       modify the next pointer in the dummy head. So no need for
       double pointers */
    node_t dummy_head;
    dummy_head.next = NULL;
    dummy_head.value = -1;
    for (i = 0; i &amp;lt; 5; i++) {
        insertbeginning_withdummy(&amp;amp;dummy_head, numbs[i]); // pass pt
        printlist(dummy_head.next);
    }
    for (i = 0; i &amp;lt; 5; i++) {
        insertend_withdummy(&amp;amp;dummy_head, numbs[i]);
```

```
        printlist(dummy_head.next);
    }

    return 0;
}



&amp;gt;  gcc linkedlist.c  -g
&amp;gt; ./a.out
1,
2, 1,
3, 2, 1,
4, 3, 2, 1,
5, 4, 3, 2, 1,
5, 4, 3, 2, 1, 1,
5, 4, 3, 2, 1, 1, 2,
5, 4, 3, 2, 1, 1, 2, 3,
5, 4, 3, 2, 1, 1, 2, 3, 4,
5, 4, 3, 2, 1, 1, 2, 3, 4, 5,
1,
2, 1,
3, 2, 1,
4, 3, 2, 1,
5, 4, 3, 2, 1,
5, 4, 3, 2, 1, 1,
5, 4, 3, 2, 1, 1, 2,
5, 4, 3, 2, 1, 1, 2, 3,
5, 4, 3, 2, 1, 1, 2, 3, 4,
5, 4, 3, 2, 1, 1, 2, 3, 4, 5,
```

Function to check if singly linked list is palindrome

This problem comes from the 'Cracking the coding interview' book. It uses the slow/fast approach to traverse a linked list, and a stack to check if list is palindrome.

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

#define TRUE 1
#define FALSE 0

struct node {
    struct node *next;
    int key;
};
typedef struct node node_t;

void insertendlist(node_t *head, int key) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode->key = key;
    newnode->next = NULL;

    node_t *ptr = head;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = newnode;
}

void printlist(node_t *head) {
    node_t *ptr = head->next;
    while (ptr != NULL) {
        printf("%d, ", ptr->key);
        ptr = ptr->next;
    }
    printf("\n");
}

int islistpalindrome(node_t *head) {
    int stack[20];
    int index = 0;
    node_t *fast = head->next;  // fast goes two nodes at a time
    node_t *slow = head->next;  // when fast at end, slow at middle
    // load stack with half list
    while (fast != NULL && fast->next != NULL) {
        stack[index++] = slow->key;
        fast = fast->next->next;
        slow = slow->next;
    }
    // check second half of link lised
```

```
    while (slow != NULL) {
        if (stack[--index] != slow->key)
            return FALSE;
        slow = slow->next;
    }
    return TRUE;
}

int main() {
    int i;

    node_t list1;
    list1.next = NULL;
    int nopalindrome[] = {2, 6, 3, 7, 5, 6};
    for (i = 0; i < 6; i++)
        insertendlist(&list1, nopalindrome[i]);
    printlist(&list1);
    if (islistpalindrome(&list1))
        printf("List is palindrome\n");
    else
        printf("List is not palindrome\n");



    node_t list2;
    list2.next = NULL;
    int palindrome[] = {2, 6, 4, 4, 6, 2};
    for (i = 0; i < 6; i++)
        insertendlist(&list2, palindrome[i]);
    printlist(&list2);
    if (islistpalindrome(&list2))
        printf("List is palindrome\n");
    else
        printf("List is not palindrome\n");

    return 0;
}


gcc llpalindrome.c -g
./a.out
2, 6, 3, 7, 5, 6,
List is not palindrome
2, 6, 4, 4, 6, 2,
List is palindrome
```

Reverse linear linked list

Function to reverse a single linear linked list

```
struct node {
        struct node *next;
        void *data;
};
void reverse(struct node *head) {
        // if list empty or one element, nothing to reverse
        if (head->next == NULL || head->next->next == NULL)
                return;
        struct node *temp;
        struct node *current = head->next->next;
        struct node *prev = head->next;
        prev->next = NULL; // first (after head) becomes last
        while (current != NULL) {
                temp = current->next;
                current->next = prev; // reverse list
                prev = current;
                current = temp;
        }
        head->next = prev; // last becomes first
}
```

return string with common characters in two strings in square and linear time

This is a common C interview question that I found online. It asks to write a function which takes two strings and returns a string containing only the characters found in both strings in the order of the first string given.

The solution explores concepts such as string manipulation, bit manipulation, const data, const pointers, dynamic memory allocation, and function scope.

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

/* Write a function f(a, b) which takes two character string
arguments and returns a string containing only the characters
 found in both strings in the order of a. Write a version which
  is order N-squared and one which is order N.
*/


int getlenght1(const char s[]) { // same as '(const char * const s)'
    int k = 0;
    while(s[k++]); // same as      while(s[k++] != '\0');
                   // and same as  while (*(s + k++) != '\0')
    k--;
    printf("size %d\n", k);
    return k;
}

int getlenght2(const char * s) {
    const char * s0 = s;
    while (*s++); // wouldnt work if s defined as const char * const s
    printf("size %s: %d\n", s0, (s-s0-1));
    return (s-s0-1);
}

char * solution_nsquare(const char * const a, const char * const b) {
    char *common;
    int i, j, k;
    int na = getlenght2(a);
    int nb = getlenght2(b);
    common = malloc(sizeof(*common) * na);
    k = 0;
    for (i = 0; i < na; i++) {
        for (j = 0; j < nb; j++) {
            if (b[j] == a[i]) {
                common[k] = a[i];
                k++;
```

```
            }
        }
    }
    common[k] = '\0';
    return common;
}

/* Review of bitwise operations (bit manipulation:
    && is logical operator: result is true (1) or false (0)
    & is bitwise operator, applies && to each bit. result is a number
    set bit x:              vble |= (1<<x)
    set bits x and y:       vble |= (1<<x|1<<y)
    clear bit x:            vble &= ~(1<<x)
    toggle (change) bit x:  vble ^= (1<<x)
    check if bit x set:     if(vble & (1<<x))
    get lower x bits:       v = vble & (2**x-1)
    get higher x bits:      v = vble & ~(2**(16-x)-1)
 */
char * solution_linear(const char a[], const char b[]) {
    int i, letter, k;
    unsigned long bitarray = 0x0;
    i = 0;
    /* scan b string */
    while (b[i]) {
        letter = b[i] - 'a' + 1;
        bitarray |= 0x1<<letter; /* set bit for that letter */
        i++;
    }
    /* now scan a, so common letters are saved in same order as a */
    char *common = malloc(sizeof(char) * i);
    i = 0; k = 0;
    while (a[i]) {
        letter = a[i] - 'a' + 1; // 'x' for characters, "x" for terminated s
        if (bitarray & (0x1<<letter)) { /* check if bit set */
            common[k++] = a[i];
        }
        i++;
    }
    common[k] = '\0';

    /* 'common' was allocated dynamically, so it won't be erased when functi
        returns. But it needs to be freed in main */
    return common;
}

int main() {
```

```
    char *a = "asdfqwer";
    char b[] = "skelrpfa";
    char *common1 = solution_nsquare(a, b);
    char *common2 = solution_linear(a, b);
    printf("a: %s\nb: %s\ncommon square: %s\ncommon linear: %s\n",
            a, b, common1, common2);
    free(common1);
    free(common2);
    return 0;
}

gcc commonchar.c
./a.out
size asdfqwer: 8
size skelrpfa: 8
a: asdfqwer
b: skelrpfa
common square: asfer
common linear: asfer
```

Producer-consumer problem with semaphores

The producer and consumer problem is easier and more reliable to solve and with condition variables than with semaphores. However here I wanted to use semaphores so we can see how they work.
Parallel to the other program I wrote with condition variables, here I used two semaphores, one for buffer full, and one for buffer empty. The cool thing about Unix (counting) semaphores they have a value (as opposed to binary semaphores which don't), so I basically used them as a buffer index. Each time a producer queues an element into buffer, the producer posts to the empty buffer semaphore. This will increment its value, and if it were 0 it would wake up consumer threads waiting on that semaphore. The consumer thread that waits on that semaphore, upon waking up it will decrement the value of the semaphore. So empty buffer semaphore basically follows up with the buffer index.
The full buffer semaphore does the opposite. Consumer threads post to it when they dequeue from the buffer, which increments the semaphore value. And producer threads wait in them, which means they sleep if buffer is full, and decrement the semaphore value each time they add element to buffer.

A mutex is used to protect access to buffer.
This is not an ideal good producer-consumer implementation, and I believe there is possibility for race condition (explained in code).

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define SIZE 5
#define NUMB_THREADS 6
#define PRODUCER_LOOPS 2

typedef int buffer_t;
buffer_t buffer[SIZE];
int buffer_index;

pthread_mutex_t buffer_mutex;
/* initially buffer will be empty.  full_sem
   will be initialized to buffer SIZE, which means
   SIZE number of producer threads can write to it.
   And empty_sem will be initialized to 0, so no
   consumer can read from buffer until a producer
   thread posts to empty_sem */
sem_t full_sem;  /* when 0, buffer is full */
sem_t empty_sem; /* when 0, buffer is empty. Kind of
                    like an index for the buffer */

/* sem_post algorithm:
    mutex_lock  sem_t->mutex
    sem_t->value++
    mutex_unlock sem_t->mutex

   sem_wait algorithn:
    mutex_lock sem_t->mutex
    while (sem_t->value > 0) {
        mutex_unlock sem_t->mutex
        sleep... wake up
        mutex_lock sem_t->mutex
    }
    sem_t->value--
    mutex_unlock sem_t->mutex
*/
```

```
void insertbuffer(buffer_t value) {
    if (buffer_index < SIZE) {
        buffer[buffer_index++] = value;
    } else {
        printf("Buffer overflow\n");
    }
}

buffer_t dequeuebuffer() {
    if (buffer_index > 0) {
        return buffer[--buffer_index]; // buffer_index-- would be error!
    } else {
        printf("Buffer underflow\n");
    }
    return 0;
}


void *producer(void *thread_n) {
    int thread_numb = *(int *)thread_n;
    buffer_t value;
    int i=0;
    while (i++ < PRODUCER_LOOPS) {
        sleep(rand() % 10);
        value = rand() % 100;
        sem_wait(&full_sem); // sem=0: wait. sem>0: go and decrement it
        /* possible race condition here. After this thread wakes up,
           another thread could aqcuire mutex before this one, and add to li
           Then the list would be full again
           and when this thread tried to insert to buffer there would be
           a buffer overflow error */
        pthread_mutex_lock(&buffer_mutex); /* protecting critical section */
        insertbuffer(value);
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&empty_sem); // post (increment) emptybuffer semaphore
        printf("Producer %d added %d to buffer\n", thread_numb, value);
    }
    pthread_exit(0);
}

void *consumer(void *thread_n) {
    int thread_numb = *(int *)thread_n;
    buffer_t value;
    int i=0;
    while (i++ < PRODUCER_LOOPS) {
```

```
        sem_wait(&empty_sem);
        /* there could be race condition here, that could cause
           buffer underflow error */
        pthread_mutex_lock(&buffer_mutex);
        value = dequeuebuffer(value);
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&full_sem); // post (increment) fullbuffer semaphore
        printf("Consumer %d dequeue %d from buffer\n", thread_numb, value);
    }
    pthread_exit(0);
}

int main(int argc, int **argv) {
    buffer_index = 0;

    pthread_mutex_init(&buffer_mutex, NULL);
    sem_init(&full_sem, // sem_t *sem
             0, // int pshared. 0 = shared between threads of process,  1 = s
             SIZE); // unsigned int value. Initial value
    sem_init(&empty_sem,
             0,
             0);
    /* full_sem is initialized to buffer size because SIZE number of
       producers can add one element to buffer each. They will wait
       semaphore each time, which will decrement semaphore value.
       empty_sem is initialized to 0, because buffer starts empty and
       consumer cannot take any element from it. They will have to wait
       until producer posts to that semaphore (increments semaphore
       value) */
    pthread_t thread[NUMB_THREADS];
    int thread_numb[NUMB_THREADS];
    int i;
    for (i = 0; i < NUMB_THREADS; ) {
        thread_numb[i] = i;
        pthread_create(thread + i, // pthread_t *t
                       NULL, // const pthread_attr_t *attr
                       producer, // void *(*start_routine) (void *)
                       thread_numb + i);  // void *arg
        i++;
        thread_numb[i] = i;
        // playing a bit with thread and thread_numb pointers...
        pthread_create(&thread[i], // pthread_t *t
                       NULL, // const pthread_attr_t *attr
                       consumer, // void *(*start_routine) (void *)
                       &thread_numb[i]);  // void *arg
        i++;
```

```
    }

    for (i = 0; i < NUMB_THREADS; i++)
        pthread_join(thread[i], NULL);

    pthread_mutex_destroy(&buffer_mutex);
    sem_destroy(&full_sem);
    sem_destroy(&empty_sem);

    return 0;
}
```

This is the result:

```
gcc producer_consumer_sem.c -lpthread
./a.out

Producer 0 added 15 to buffer
Consumer 1 dequeue 15 from buffer
Producer 2 added 35 to buffer
Consumer 3 dequeue 35 from buffer
Producer 0 added 92 to buffer
Consumer 5 dequeue 92 from buffer
Producer 4 added 49 to buffer
Consumer 1 dequeue 49 from buffer
Producer 4 added 62 to buffer
Consumer 3 dequeue 62 from buffer
Producer 2 added 27 to buffer
Consumer 5 dequeue 27 from buffer
```

There is a better way to solve this problem, checking for spurios wakeups for full and empty conditions. The following solution is similar to how this problem would be approached with conditional variables, except that conditional variables are able to release lock and sleep on condition atomically, while here it's done in two steps.

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

```
#include <pthread.h>
#include <semaphore.h>



#define SIZE 10
#define NUMB_THREADS 10
#define PRODUCER_LOOPS 2
#define CONSUMER_LOOPS 2

#define TRUE 1
#define FALSE 0

typedef int buffer_t;
buffer_t buffer[SIZE];
int buffer_index;

pthread_mutex_t buffer_mutex;
/* initially buffer will be empty.  full_sem
   will be initialized to buffer SIZE, which means
   SIZE number of producer threads can write to it.
   And empty_sem will be initialized to 0, so no
   consumer can read from buffer until a producer
   thread posts to empty_sem */
sem_t full_sem;  /* when 0, buffer is full */
sem_t empty_sem; /* when 0, buffer is empty. Kind of
                    like an index for the buffer */

/* sem_post algorithm:
    mutex_lock  sem_t->mutex
    sem_t->value++
    mutex_unlock sem_t->mutex

   sem_wait algorithn:
    mutex_lock sem_t->mutex
    while (sem_t->value > 0) {
        mutex_unlock sem_t->mutex
        sleep... wake up
        mutex_lock sem_t->mutex
    }
    sem_t->value--
    mutex_unlock sem_t->mutex
*/


void insertbuffer(buffer_t value) {
```

```
    if (buffer_index < SIZE) {
        buffer[buffer_index++] = value;
    } else {
        printf("Buffer overflow\n");
    }
}

buffer_t dequeuebuffer() {
    if (buffer_index > 0) {
        return buffer[--buffer_index]; // buffer_index-- would be error!
    } else {
        printf("Buffer underflow\n");
    }
    return 0;
}


int isempty() {
    if (buffer_index == 0)
        return TRUE;
    return FALSE;
}

int isfull() {
    if (buffer_index == SIZE)
        return TRUE;
    return FALSE;
}

void *producer2(void *thread_n) {
    int thread_numb = *(int *)thread_n;
    buffer_t value;
    int i=0;
    while (i++ < PRODUCER_LOOPS) {
        sleep(rand() % 10);
        value = rand() % 100;
        pthread_mutex_lock(&buffer_mutex);
        do {
            // cond variables do the unlock/wait and wakeup/lock atomically,
            // which avoids possible race conditions
            pthread_mutex_unlock(&buffer_mutex);
            // cannot go to slepp holding lock
            sem_wait(&full_sem); // sem=0: wait. sem>0: go and decrement it
            // there could still be race condition here. another
            // thread could wake up and aqcuire lock and fill up
```

```
            // buffer. that's why we need to check for spurious wakeups
            pthread_mutex_lock(&buffer_mutex);
        } while (isfull()); // check for spurios wake-ups
        insertbuffer(value);
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&empty_sem); // post (increment) emptybuffer semaphore
        printf("Producer %d added %d to buffer\n", thread_numb, value);
    }
    pthread_exit(0);
}

void *consumer2(void *thread_n) {
    int thread_numb = *(int *)thread_n;
    buffer_t value;
    int i=0;
    while (i++ < PRODUCER_LOOPS) {
        pthread_mutex_lock(&buffer_mutex);
        do {
            pthread_mutex_unlock(&buffer_mutex);
            sem_wait(&empty_sem);
            pthread_mutex_lock(&buffer_mutex);
        } while (isempty()); //check for spurios wakeups
        value = dequeuebuffer(value);
        pthread_mutex_unlock(&buffer_mutex);
        sem_post(&full_sem); // post (increment) fullbuffer semaphore
        printf("Consumer %d dequeue %d from buffer\n", thread_numb, value);
   }
    pthread_exit(0);
}

int main(int argc, int **argv) {
    buffer_index = 0;

    pthread_mutex_init(&buffer_mutex, NULL);
    sem_init(&full_sem, // sem_t *sem
             0, // int pshared. 0 = shared between threads of process,  1 = s
             SIZE); // unsigned int value. Initial value
    sem_init(&empty_sem,
             0,
             0);
    /* full_sem is initialized to buffer size because SIZE number of
       producers can add one element to buffer each. They will wait
       semaphore each time, which will decrement semaphore value.
       empty_sem is initialized to 0, because buffer starts empty and
       consumer cannot take any element from it. They will have to wait
```

```
        until producer posts to that semaphore (increments semaphore
        value) */
    pthread_t thread[NUMB_THREADS];
    int thread_numb[NUMB_THREADS];
    int i;
    for (i = 0; i < NUMB_THREADS; ) {
        thread_numb[i] = i;
        pthread_create(thread + i, // pthread_t *t
                       NULL, // const pthread_attr_t *attr
                       producer2, // void *(*start_routine) (void *)
                       thread_numb + i);  // void *arg
        i++;
        thread_numb[i] = i;
        // playing a bit with thread and thread_numb pointers...
        pthread_create(&thread[i], // pthread_t *t
                       NULL, // const pthread_attr_t *attr
                       consumer2, // void *(*start_routine) (void *)
                       &thread_numb[i]);  // void *arg
        i++;
    }

    for (i = 0; i < NUMB_THREADS; i++)
        pthread_join(thread[i], NULL);

    pthread_mutex_destroy(&buffer_mutex);
    sem_destroy(&full_sem);
    sem_destroy(&empty_sem);

    return 0;
}
```

C++ concentrated

I wrote a program that includes all the most common C++ topics asked in interviews. There are lots of comments in the code that explain all the theory behind the code.

```
#include <iostream>
#include <cstring>
```

```
using std::cout;
using std::endl;
using std::strncpy;
using std::strlen;

#define FIRSTNAME_LEN 10

/* Concepts to remember in this program:
    - Class, class' member functions (or methods),
      class' data members (or attributes)
    - const object
    - const member function
    - static data member and static member function
    - inheritance, base class, derived class
    - composition
    - friend class and friend function
    - polymorphism
    - virtual function
    - abstract class
    - dynamic memory allocation
    - class' constructor and destructor
    - default constractor
    - dangling pointer

*/

/* const == read-only -> must be initialized when created, and cant be modif
   const variable: can be initialized, but not modified after that: const in
   const obj: object cannot be modified: const Date d(1, 1, 2000)
              If it is member of class, must be initialized via member initi
              It can only call methods defined as const
   const member funcs: the only methods that can be called by const object
   const data member: must be initialized via member initializer (':' in the
                      cannot be modified after */

/**** CLASSES INTERFACES ****/

class Date {

/* public data can be accessed by methods of class, and by objects of the cl
public:
    Date(int = 1, int = 1, int = 2014);  // constructor. Defaults to 1, 1, 2
    void setDate(int, int, int);         // non const method: cannot be call
    void printDate() const;              // const method: can be called by c

/* private data can only be accessed by the methods of the class
```

```
   They cannot be accessed by objects of the class (data encapsulation or da
   Objects of the class usually access them via public setters and getters
   They cannot be accessed by methods of a derived class */
private:
    int day;
    int month;
    int year;

/* protected data, like private data, cannot be accessed by objects
   of the class. Only by methods of the class.
   But protected data can be accessed by methods of a derived class */
//protected:

/* by default (if not public/private keyword is used), the data members
   will be private. This is a difference with struct in C, where all
   data is by default public. */
};

/* an abstract class is a class with at least one pure virtual
   function. Since they have one or more pure virtual functions,
   which do not have the function implementation, they canNOT
   instantiate objects */
class AbstractEmployee {
public:
     /* pure virtual function is a virtual function with the '=0'
        It means that implementation in the base class is not provided
        Therefore all derived class must implement this function,
        which will overried the function of the abstract class */
     virtual void printInfovirtual() const = 0;
};

class Employee : public AbstractEmployee{

public: // public members can be accessed only by objects of the class
    Employee(const char firstname[],        // array of const data (firstnam
             const char * const middlename, // const pointer to const data
             int salary,
/* Using composition. constant object of Date
   Passed as reference for better performance
   If this function (constructor) changed datehire, it would
   also change for the function calling, because it is passed
   by reference. To protect it, it is declared const, so constructor
   cannot change it */
             const Date &datehire);

/* this class requires explicit constructor, destructor, copy constructor
```

```
   and assignment operator because some data members (middlename)
   use dynamic memory */
    ~Employee();                    // Destructor
    Employee(const Employee &src); // copy constructor
    Employee& operator=(const Employee &rhs); // assignment operator
    void setnames(const char [], const char * const);
    Employee& setsalary(int);   // cascading method. Returns pointer to obje
    static int getnumbemployees();  // static member function. Used to acces
    void printname() const;    // const method, which means it can be used b

    void printInfo() const;
/* next function is pure virtual function in AbstractEmployee, so it
   needs to be implemented in this class.
   declared as virtual to enable polymorphism.
   Pointer to base class Employee will still call method of derived class
   Manager if object is from derived class */
    virtual void printInfovirtual() const;

private:
/* first and middle name could be defined const, but then it wouldnt be poss
   to modify them after employee object was created. */
    char firstname_array[FIRSTNAME_LEN]; // using array.
    char *middlename_dynamic;  // using dynamic memory allocation

    int salary;

/* employee HAS a datehire of type Date (composition)
   Defined as const, so it cannot be modified after object initialized
   It CANNOT be changed after object employee is created
   It must be initialized in the constructor via member initializer (:)
   This object data member can only call Date methods defined with const */
    const Date dateHire;
/* static data member means one copy of the variable is shared by
   all objects of the class(as opposed to non static, for which
   each object has its own copy of the data members)
   So in this case all objects will see the same number
   of employees in the company
   If public, it could be accessed as object.totalnumbemployees or
   Employee::totalnumbemployees
   Private static data member needs a public static member
   function to be accessed */
    static int totalnumbemployees;

/* static, so one copy for all objects.
   const, so it cannot be modified after it is defined
   private, so it will need a static method to access it*/
```

```
    const static char companyname[];

};

class Manager : public Employee {
/* manager IS an employee (inheritance)
   public inh: public members of base class become public members of derived
               protected members of base class become protected members of d
               derived class canNOT access private members of base class.
               Need to use public getters/setters to access private members
   private inh: public and protected members of base clase become
                private member of derived class. This means that all data of
                derived class object will be inaccessible (private data is o
                by methods of class, not by object, and not by methods of de
                It is not a 'is-a' relationship, but more like a 'has-a' rel
   protected inh: public and protected members of base clase become
                protected members of derived class (data accessible by metho
                and methods of derived class, not by object)
                Usually used for last level of inheritance in a hierarchy */
public:
    Manager(const char firstname[],
            const char * const middlename,
            int salary, int bonus,
            const Date &datehire,
            int teamSize);
    ~Manager();                              // destructor
    int getteamsize();
    void setteamsize(int );

    void printInfo() const;
    virtual void printInfovirtual() const;

private:
    int bonus;
    int teamSize;
/* class inherites all the data members from base class (firstname, etc)
   but it wont be able to access private data member from base class */
};

/***** CLASSES IMPLEMENTATION *****/

/**** Date member functions *****/
Date::Date(int d, int m, int y) :
    day (d), month (m), year (y) { // member initializer
    cout << "Date constructor for: " << m << "/" << d << "/" << y << endl;
}
```

```
void Date::setDate(int d, int m, int y) // cannot be called by const object
{
    this->day = d;
    month = m;
    year = y;
}
void Date::printDate() const {  // defined as const, so it can be called by c
    cout << month << "/" << day << "/" << year << endl;
}

/***** Employee member functions ******/

// static data member needs to be
// initialized ONCE
int Employee::totalnumbemployees = 0;
const char Employee::companyname[] = "Medina Corp";

Employee::Employee(const char first[],
                   const char * const middle,
                   int s, const Date &date) :
                   dateHire (date) {  // dateHire is const member,
                                      // so it must be initialized here
    cout << "Employee constructor for " << first << endl;
    setnames(first, middle);
    salary = s;
    totalnumbemployees++;
    // dateHire = date; // this would fail because dateHire
                        // is const member
}
Employee::~Employee() {
    cout << "Employee destructor for " << firstname_array << endl;
    delete [] middlename_dynamic;
    totalnumbemployees--;
}

/* copy constructor is required when a class uses dynamically allocated data
   The default copy constructor makes shallow copies of data. This means
   that the copied object will get a copy of the data member pointer, but the
   underlying data on the heap will be the same. So if the copied object char
   the data or exists and frees the data, the original object will have its
   data member pointers pointing to non-valid memory (dangling pointers).
   The explicit copy constructor makes deep copies, so that copied object has
  own allocated memory */
Employee::Employee(const Employee &src) :
    dateHire (src.dateHire) {
    cout << "Employee copy constructor for " << src.firstname_array << " " <<
```

```
    setnames(src.firstname_array, src.middlename_dynamic);

    // setnames makes deep copy of dynamic memory
    salary = src.salary;
}

/* assignment operator. The difference with copy constructor is
   that the assignment operator is used on a already initialized object.
   When two objects are instantiated for a class, each
   will have its own memory.
   Manager manager1(), manager2();
   manager2 = manager1;
   When assignment is used, the class default assignment
   operator will direct the data pointers of the left
   object to the memory of right object. But the dynamically
   allocated memory of the left object will not be
   freed but will be unaccessible (orphaned memory).
   This causes leakage memory.
   To avoid this, dynamic memory of left object
   should be freed. And then deep copy of
   dynamically allocated data should be performed.
*/
Employee& Employee::operator=(const Employee &rhs) {
    cout << "Employee assignment operator for " << rhs.firstname_array << en
    // check self assignment
    if (this == &rhs) { return *this; }
    // free lhs dynamic memory
    delete [] middlename_dynamic;
    // dynamic allocation and deep copy data
    setnames(rhs.firstname_array, rhs.middlename_dynamic);
}

void Employee::setnames(const char first[], const char * const middle) {
    // array
    strcpy(firstname_array, first);
    // truncating firstname if array entered greater than firstname_len
    if (strlen(first) > FIRSTNAME_LEN) {
        memset(firstname_array + FIRSTNAME_LEN - 1, '\0', 1);
    }
    // dynamic memory
    middlename_dynamic = new char[strlen(middle) + 1];
    strcpy(middlename_dynamic, middle);  //strcpy performs deep copy of poin
    // string
}

Employee& Employee::setsalary(int s) {
```

```
    if (s > 0 && s < 200000) { // setter functions should check data
        salary = s;
    }
    return *this; // enables cascading calls such as empl.setsalar(1000).pri
}
void Employee::printname() const {
    cout << firstname_array << " " << middlename_dynamic;
}
void Employee::printInfo() const {
    cout << "Employee class (base) printInfo" << endl;
    cout << "Name: " << firstname_array << endl;
    cout << "Salary: " << salary << endl;
}

void Employee::printInfovirtual() const {
    cout << "Employee class (base) printInfo" << endl;
    cout << "Name: " << firstname_array << endl;
    cout << "Salary: " << salary << endl;
}

/*********** MANAGER CLASS IMPLEMENTATION ************/

Manager::Manager(const char first[], const char * const middle,
                 int sal, int b, const Date &hiredate,
                 int numbteam)
    : Employee(first, middle, sal, hiredate) { // calls base class
    cout << "Constructor for Manager called" << endl;
    teamSize = numbteam;
    bonus = b;
}

Manager::~Manager() {
    cout << "Manager Destructor called for "; // << firstname_array << endl;
    printname();
    cout << endl;
    // base class destructor is called implicitaly
}

void Manager::printInfo() const {
    cout << "Manager class (derived) printInfo" << endl;
    Employee::printInfo();
    cout << "Bonus: " << bonus << endl;
}

void Manager::printInfovirtual() const {
    cout << "Manager class (derived) printInfo" << endl;
```

```
    Employee::printInfo();
    cout << "Bonus: " << bonus << endl;
}

int main() {
    cout << "Instatiatiating date: " << endl;
    Date date1(10, 10, 2000);
    cout << endl << "Instantiating employee 1: " << endl;
    Employee empl1("name1", "last1", 1000, date1);
    cout << endl << "Instantiating employee 2: " << endl;
    Employee empl2("name2", "last2", 200, Date(2, 2, 2000));
    cout << endl << "Instantiating employee 3: " << endl;
    Employee empl3("name3", "last3", 300, Date(3, 3, 3000));
    cout << endl << "Copying employee 1 to employee 3:" << endl;
    empl3 = empl1;           // using assignment operator
    cout << endl << "Instantiating employee 4 as copy of employee 1:" << endl
    Employee empl4 = empl1;  // using copy constructor

    /* Polymorphism causes the same function (printInfo) to cause different a
       depending on the type of object that invokes the function
       Without virtual functions, the type of pointer to the object determine
       function called (pointer to base class will call method of base class,
       to derived class will call method of derived class, regardless of obje
       With virtual functions, the the type of object determines the function

    /* Using non virtual functions: pointer determines behavior */
    // base pointer to derived class
    cout << endl << "Instatiating manager:" << endl;
    Manager derivedobject("managername", "managerlast", 1000, 1000, Date(7, 1
    Employee * baseptr = &derivedobject;
    cout << endl << "Calling printInfo with Employee (base) pointer to Manage
    baseptr->printInfo(); // invokes printInfo of base class (Employee)

    // derived pointer to base class is illegal
    //Manager * derivedptr = &empl1; //  error: invalid conversion from ‘Empl
    //derivedptr->printInfo(); // Manager printInfo would try to print bonus,

    /* Using virtual functions: object determines behavior */
    cout << endl << "Calling printInfovirtual with Employee (base) pointer to
    baseptr->printInfovirtual();

    /* thanks to polymorphims and virtual functions, we can have an array of
       of the base or derived classes. And just use the base ptr to call func

    cout << endl << "Done:" << endl;
    return 0;
```

```
}
```

This is the results from running the code:

```
Instatiatiating date:
Date constructor for: 10/10/2000

Instantiating employee 1:
Employee constructor for name1

Instantiating employee 2:
Date constructor for: 2/2/2000
Employee constructor for name2

Instantiating employee 3:
Date constructor for: 3/3/3000
Employee constructor for name3

Copying employee 1 to employee 3:
Employee assignment operator for name1

Instantiating employee 4 as copy of employee 1:
Employee copy constructor for name1

Instatiating manager:
Date constructor for: 7/7/2000
Employee constructor for managername
Constructor for Manager called

Calling printInfo with Employee (base) pointer to Manager (derived) object
Employee class (base) printInfo
Name: managerna
Salary: 1000

Calling printInfovirtual with Employee (base) pointer to Manager (derived) ol
Manager class (derived) printInfo
Employee class (base) printInfo
Name: managerna
Salary: 1000
Bonus: 1000

Done:
```

```
Manager Destructor called for managerna managerlast
Employee destructor for managerna
Employee destructor for name1
Employee destructor for name1
Employee destructor for name2
Employee destructor for name1
```

Find if all characters are unique in string

I found this problem in 'Cracking the Code Interview' book. It's actually the first problem given (1.1) in the book. It is a pretty common question asked in interviews. The problem is about writing a function that returns true if all characters in string are unique, and false otherwise. There is actually a lot more to it that it may look like first. I implemented a few solutions in C.

```
#define FALSE 0
#define TRUE 1

/* implement function that returns true
is all characters in a string are
unique, and false otherwise
*/

// time O(n^2). No extra space needed
int uniqueCharac1(char *ptr1) {
    char *ptr2;
    while (*ptr1++ != '\0') {
        ptr2 = ptr1;
        while (*ptr2++ != '\0') {
            if (*ptr2 == *ptr1) {
                printf("Letter %c repeated\n", *ptr2);
                return FALSE;
            }
        }
    }
    return TRUE;
}

//time O(n). Extra space O(1). Using bitmap
```

```
int uniqueCharac2(char *ptr) {
    unsigned int bit_field = 0x0; // needs 27 bits (one bit per letter).
    // int gives 4 bytes (32 bits)
    int character;
    while (*ptr++ != '\0') {
        character = *ptr - 'a' + 1; // 'a' is 1
        //printf("Letter %c is number %d\n", *ptr, character);
        // checking if bit set
        if (bit_field && (1<<character)) {
            printf("Letter %c repeated\n", *ptr);
            return FALSE;
        }
        // setting bit
        bit_field |= (1&lt;&lt;character);
    }
    return TRUE;
}

char * quicksort(char *string) { // TO DO
    return string;
}
int binarysearch(char *string, char letter) { // TO DO
    return TRUE;
}

//time O(n*log n). No extra space needed. Using quicksort
int uniqueCharac3(char *ptr) {
    char *sorted = quicksort(ptr);
    while (*ptr++ != '') {
        if (binarysearch(ptr + 1, *ptr)) {
            printf("Letter %c repeated\n", *ptr);
            return TRUE;
        }
    }
    return FALSE;
}

int main(int argc, char **argv){
    char string[] = "asdfqwersdfga";
    if (uniqueCharac1(string)) {printf("Characters in %s are unique\n", stri
    if (uniqueCharac2(string)) {printf("Characters in %s are unique\n", stri

}
```