Learning C by example

C programs that I have found useful during my studies

Category Archives: C language

Programs showing interesting features of C language

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));
```

```
26      new->value = value;
27      new->next = NULL;
28
29      /* empty */
30      if (*list == NULL) {
31          *list = new;
32          return;
33      }
34      /* element becomes first. can be merged with case above */
35      if ((*list)->value > value) {
36          new->next = *list;
37          *list = new;
38          return;
39      }
40
41      node_ts *prev = *list;
42      node_ts *ptr = (*list)->next;
43      while (ptr != NULL && ptr->value < value) {
44          prev = ptr;
45          ptr = ptr->next;
46      }
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) {
101          /* if list has only one node */
102          if ((*list)->next == *list) {
103              new->next = *list;
104              (*list)->next = new;
105              *list = new;
106              return;
107          }
108          /* if list has more nodes, get last node */
109          node_t *ptr = (*list)->next;
110          while (ptr->next != *list) {
111              assert(ptr->next != NULL);
112              ptr = ptr->next;
113          }
114          /* insert node */
115          ptr->next = new;
116          new->next = *list;
117          *list = new;
118          return;
119      }
120  #else
121      /* another way to insert at beginning
122         without having to traverse list */
123      if ((*list)->value > value) {
124          memcpy(new, *list, sizeof(node_t));
125          (*list)->value = value;
126          (*list)->next = new;
127          return;
128      }
129  #endif
130
131
132      node_ts *prev = *list;
133      node_ts *ptr = (*list)->next;
```

```
134         while (ptr != *list && ptr->value < value) {
135             prev = ptr;
136             ptr = ptr->next;
137         }
138         prev->next = new;
139         new->next = ptr;
140     }
141
142     void print_singlecircular(node_ts * list) {
143         node_ts * ptr = list;
144         do {
145             printf("%d, ", ptr->value);
146             ptr = ptr->next;
147         } while (ptr != list);
148         printf("\n");
149     }
150
151     void reverse_singlecircular(node_ts ** list) {
152
153         if (*list == NULL || (*list)->next == *list)
154             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 * doublecircularll = 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,
```

```
testing double circular linked list
3,
1, 3,
1, 3, 6,
1, 3, 5, 6,
1, 3, 5, 6, 9,
9, 6, 5, 3, 1,
```

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 */
```

```
31      ptr = (void *)((uintptr_t)ptr - align);
32      /* retreive from there the original malloced pointer */
33      ptr = (void *)(*(unsigned int *)ptr);
34      printf("free memory at address %p\n", ptr);
35      /* free that pointer */
36      free(ptr);
37  }
38
39  int main() {
40      void *ptr = mallocaligned(1000, 64);
41      printf("allocated pointer at %p", ptr);
42      freealigned(ptr, 64);
43      return 0;
44  }
```

Reverse linear linked list

Function to reverse a single linear linked list

```
1   struct node {
2           struct node *next;
3           void *data;
4   };
5   void reverse(struct node *head) {
6           // if list empty or one element, nothing to reverse
7           if (head->next == NULL || head->next->next == NULL)
8                   return;
9           struct node *temp;
10          struct node *current = head->next->next;
11          struct node *prev = head->next;
12          prev->next = NULL; // first (after head) becomes last
13          while (current != NULL) {
14                  temp = current->next;
15                  current->next = prev; // reverse list
16                  prev = current;
17                  current = temp;
18          }
19          head->next = prev; // last becomes first
20  }
```

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.

```c
#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;
}
```

```
45
46  /* Review of bitwise operations (bit manipulation:
47      && is logical operator: result is true (1) or false (0)
48      & is bitwise operator, applies && to each bit. result is a number
49      set bit x:              vble |= (1<<x)
50      set bits x and y:       vble |= (1<<x|1<<y)
51      clear bit x:            vble &= ~(1<<x)
52      toggle (change) bit x:  vble ^= (1<<x)
53      check if bit x set:     if(vble & (1<<x))
54      get lower x bits:       v = vble & (2**x-1)
55      get higher x bits:      v = vble & ~(2**(16-x)-1)
56   */
57  char * solution_linear(const char a[], const char b[]) {
58      int i, letter, k;
59      unsigned long bitarray = 0x0;
60      i = 0;
61      /* scan b string */
62      while (b[i]) {
63          letter = b[i] - 'a' + 1;
64          bitarray |= 0x1<<letter; /* set bit for that letter */
65          i++;
66      }
67      /* now scan a, so common letters are saved in same order as a */
68      char *common = malloc(sizeof(char) * i);
69      i = 0; k = 0;
70      while (a[i]) {
71          letter = a[i] - 'a' + 1; // 'x' for characters, "x" for termina
72          if (bitarray & (0x1<<letter)) { /* check if bit set */
73              common[k++] = a[i];
74          }
75          i++;
76      }
77      common[k] = '\0';
78
79      /* 'common' was allocated dynamically, so it won't be erased when f
80          returns. But it needs to be freed in main */
81      return common;
82  }
83
84  int main() {
85      char *a = "asdfqwer";
86      char b[] = "skelrpfa";
87      char *common1 = solution_nsquare(a, b);
88      char *common2 = solution_linear(a, b);
89      printf("a: %s\nb: %s\ncommon square: %s\ncommon linear: %s\n",
90              a, b, common1, common2);
91      free(common1);
92      free(common2);
93      return 0;
94  }
95
96  gcc commonchar.c
97  ./a.out
98  size asdfqwer: 8
```

```
size skelrpfa: 8
a: asdfqwer
b: skelrpfa
common square: asfer
common linear: asfer
```

const pointer to const data

This post explores the difference between constant pointers and pointers to constant data, and the priority difference between reference operator (*) and increment operator (++). This is a very common question asked in interviews.

```c
#include <stdio.h>

int main (){

    int v[] = {5, 20};
    int *p = v;
    printf("p: %d, ", (*p)++); // increments data pointed
    printf("p: %d, ", *p++);   // increments pointer. Equivalent to *(p+
    printf("p: %d\n", *p);

    /* pointer to constant data (can't modified data pointed) */
    const int *p2 = v;
    //int const *p2 = v; // same as above
    //printf("const ptr: %d, ", (*p2)++); // error: increment of read-onl
    printf("const ptr: %d, ", *p2++);   // '++' has priority, so increme
    //printf("const ptr: %d, ", *(p2++)); // same as above
    printf("const ptr: %d\n", *p2);

    /* constant pointer (can't modified pointer). Same as 'int p3[] = v'
    int * const p3 = v;
    //int * p3 const = v;    // error: expected ‘=’,..., before ‘const’
    printf("ptr to const data: %d, ", (*p3)++); // incrementing data refe
    //printf("ptr to const data: %d, ", *p3++); // error: increment of re
    printf("ptr to const data: %d\n", *p3);

    /* constant pointer to constant data */
    const int * const p4 = v;
    // int const * const p4; // same as above
    //printf("const ptr to const data: %d, ", (*p4)++); // error: increme
    //printf("const ptr to const data: %d, ", *p4++);   // error: increme
    printf("const ptr to const data: p4: %d\n", *p4);

    /* array definition is equivalent to constant pointer. Same as 'int
    printf("ptr from array: %d, ", (*v)++);
    //printf("ptr from array: %d, ", *v++); // error: lvalue required as
```

```
    printf("ptr from array: %d\n", *v);

    return 1;
}

gcc sorting2.c
./a.out
p: 5, p: 6, p: 20
const ptr: 6, const ptr: 20
ptr to const data: 6, ptr to const data: 7
const ptr to const data: p4: 7
ptr from array: 7, ptr from array: 8
```

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)
```

```
28      int character;
29      while (*ptr++ != '\0') {
30          character = *ptr - 'a' + 1; // 'a' is 1
31          //printf("Letter %c is number %d\n", *ptr, character);
32          // checking if bit set
33          if (bit_field && (1<<character)) {
34              printf("Letter %c repeated\n", *ptr);
35              return FALSE;
36          }
37          // setting bit
38          bit_field |= (1&lt;&lt;character);
39      }
40      return TRUE;
41  }
42
43  char * quicksort(char *string) { // TO DO
44      return string;
45  }
46  int binarysearch(char *string, char letter) { // TO DO
47      return TRUE;
48  }
49
50  //time O(n*log n). No extra space needed. Using quicksort
51  int uniqueCharac3(char *ptr) {
52      char *sorted = quicksort(ptr);
53      while (*ptr++ != '') {
54          if (binarysearch(ptr + 1, *ptr)) {
55              printf("Letter %c repeated\n", *ptr);
56              return TRUE;
57          }
58      }
59      return FALSE;
60  }
61
62  int main(int argc, char **argv){
63      char string[] = "asdfqwersdfga";
64      if (uniqueCharac1(string)) {printf("Characters in %s are unique\n",
65      if (uniqueCharac2(string)) {printf("Characters in %s are unique\n",
66
67  }
```

memcpy implementation

I was asked once during an interview to implement memcpy in C. The questions sounds simple first, but there is a lot to it actually. I think I did learn a lot by pointers and memory by researching this question. The code below shows 4 implementations of memcpy. The first one is the trivial implementation. The second one improves security by avoiding overwriting memory from the

pointers. The third ones improves performance by copying one word at a time, instead of 1 bytes at a time. The fourth one copies a word at a time, and start copying with destination aligned to word byte boundary. The fifth one (TBD) copies with both source and destination aligned to word byte boundary.

```
#include <stdio.h>;
#include <string.h>;
#include <inttypes.h>;

void mymemcpy1(void *, const void *, size_t );
void mymemcpy2(void *, const void *, size_t );
void mymemcpy3(void *, const void *, size_t );
void mymemcpy4(void *, const void *, size_t );

int main(int argc, char **argv) {
 char source[] = &amp;quot;0123456789abdcdefghi&amp;quot;;  // 21 bytes
    char dest[100];

    // void * memcpy ( void * destination, const void * source, size_t
    memcpy(dest, source, strlen(source) + 1);
    printf(&amp;quot;Source: %s. Destination: %s\n&amp;quot;, source, d

    strcpy(source, &amp;quot;0123456789abdcdefghi&amp;quot;);
    mymemcpy1(dest, source, strlen(source) + 1);
    printf(&amp;quot;Source: %s. Destination: %s\n&amp;quot;, source, d

    strcpy(source, &amp;quot;0123456789abdcdefghi&amp;quot;);
    mymemcpy2(dest, source, strlen(source) + 1);
    printf(&amp;quot;Source: %s. Destination: %s\n&amp;quot;, source, d

    strcpy(source, &amp;quot;0123456789abdcdefghi&amp;quot;);
    mymemcpy3(dest, source, strlen(source) + 1);
    printf(&amp;quot;Source: %s. Destination: %s\n&amp;quot;, source, d

    strcpy(source, &amp;quot;0123456789abdcdefghi&amp;quot;);
    mymemcpy4(dest, source, strlen(source) + 1);
    printf(&amp;quot;Source: %s. Destination: %s\n&amp;quot;, source, d
    return 0;
}

// simple implementation
void mymemcpy1(void *dest, const void *source, size_t num) {
    int i = 0;
    // casting pointers
    char *dest8 = (char *)dest;
    char *source8 = (char *)source;
    printf(&amp;quot;Copying memory %d byte(s) at a time\n&amp;quot;, s
    for (i = 0; i &amp;lt; num; i++) {
        dest8[i] = source8[i];
    }
}
// it checks that destination array does not overwrite
// source memory
```

```
49  void mymemcpy2(void *dest, const void *source, size_t num) {
50      int i = 0;
51      // casting pointers
52      char *dest8 = (char *)dest;
53      char *source8 = (char *)source;
54      printf(&amp;quot;Copying memory %d byte(s) at a time\n&amp;quot;, s
55      for (i = 0; i &amp;lt; num; i++) {
56          // make sure destination doesnt overwrite source
57          if (&amp;amp;dest8[i] == source8) {
58              printf(&amp;quot;destination array address overwrites sourc
59              return;
60          }
61          dest8[i] = source8[i];
62      }
63  }
64
65  // copies 1 word at a time (8 bytes at a time)
66  void mymemcpy3(void *dest, const void *source, size_t num) {
67      int i = 0;
68      // casting pointers
69      long *dest32 = (long *)dest;
70      long *source32 = (long *)source;
71      char *dest8 = (char *)dest;
72      char *source8 = (char *)source;
73
74      printf(&amp;quot;Copying memory %d bytes at a time\n&amp;quot;, siz
75      for (i = 0; i &amp;lt; num/sizeof(long); i++) {
76          if (&amp;amp;dest32[i] == source32) {
77              printf(&amp;quot;destination array address overwrites sourc
78              return;
79          }
80          dest32[i] = source32[i];
81      }
82      // copy the last bytes
83      i*=sizeof(long);
84      for ( ; i &amp;lt; num; i++) {
85          dest8[i] = source8[i];
86      }
87  }
88
89  /* memory addres is n-byte aligned when address is multiple of n bytes
90     b-bit aligned is equivalent to b/8-byte aligned
91     padding = n - (offset &amp;amp; ( -1)) = -offset &amp;amp; (n-1)
92     aligned offset = (offset + n-1) &amp;amp; ~(n-1)
93
94     Copies 8 bytes at a time with destination align to 64-byte boundary
95  */
96  void mymemcpy4(void * dest, const void * source, size_t size) {
97
98  #define NBYTE 8 // n-byte boundary
99
100     int i = 0;
101     int j = 0;
102
```

```
103       // short and long pointers for copying 1 and 8 (sizeof(long))
104       //  bytes at a time
105       long * destlong = (long *)dest;
106       long * sourcelong = (long *)source;
107       char * destshort = (char *)dest;
108       char * sourceshort = (char *)dest;
109
110
111       // copy first bytes until destination hits 64-byte boundary
112       while((((uintptr_t)&amp;amp;destshort[i] &amp;amp; (uintptr_t)(NBYT
113             (&amp;amp;destshort[i] != source) &amp;amp;&amp;amp; (i &amp;
114           destshort[i] = sourceshort[i];
115           i++;
116       }
117       printf(&amp;quot;%s: copied %d bytes up to %d-byte alignment\n&amp;
118              __func__, i, NBYTE);
119
120
121       // now copy 8 (sizeof(long)) bytes at a time with dest aligned
122       // align destination pointer
123       destlong = ((uintptr_t)destlong + (NBYTE-1)) &amp;amp; ~(uintptr_t)
124        // continue copying where left off (we should align source as well
125       sourcelong = (long *)sourceshort;
126       // j+1 * 8 - 1 to avoid copying beyond last element in last iterati
127       while ((j+1)*sizeof(long) - 1 + i &amp;lt; size &amp;amp;&amp;amp;\
128               &amp;amp;destlong[j] &amp;lt; (long *)source) {
129           destlong[j] = sourcelong[j];
130           j++;
131       }
132       printf(&amp;quot;%s: copied %d bytes %d bytes at a time\n&amp;quot;
133              j*sizeof(long), sizeof(long));
134
135
136       // finally copy last bytes
137       i = i + j*sizeof(long);
138       int prev_i = i;
139       while(i &amp;lt; size &amp;amp;&amp;amp; &amp;amp;destshort[i] &amp
140           destshort[i] = sourceshort[i];
141           i++;
142       }
143       printf(&amp;quot;%s: copied last %d bytes one at a time\n&amp;quot;
144              __func__, i-prev_i);
145
146   }
147
148   /* simpler implementation of mymemcpy4 */
149   void mymemcpy4b(const void * dest, void * src, size_t size) {
150
151       char * dest1 = (char *)dest;
152       char * src1 = (char *)src;
153       int n = 0;
154   #define NBYTE 64
155       // copy up to nbyte alignment
156       while ((((uintptr_t)dest1 & ~(NBYTE-1) != 0x00) && (n < size) {
```

```
157             *dest1++ = *src1++;
158             n++;
159         }
160         printf("copied %d bytes at a time up to %d bytes\n", sizeof(char),
161
162         // copy up to end minus last sizeof(long) bytes
163         long * dest2 = (long *)dest1;
164         long * src2 = (long *)src1;
165         while (n &lt; size - sizeof(long)) {
166             *dest2++ = *src2++;
167             n+=sizeof(long);
168         }
169         printf("copied %d bytes at a time up to %d bytes\n", sizeof(long),
170
171         // copy last bytes
172         src1 = (char *)src2;
173         dest1 = (char *)dest2;
174         while (n &lt; size) {
175             *dest1++ = *src1++;
176             n++;
177         }
178         printf("copied up to %d bytes\n", n);
179     }
```

The result is this:

```
1  Source: 0123456789abdcdefghi. Destination: 0123456789abdcdefghi
2  Copying memory 1 byte(s) at a time
3  Source: 0123456789abdcdefghi. Destination: 0123456789abdcdefghi
4  Copying memory 1 byte(s) at a time
5  Source: 0123456789abdcdefghi. Destination: 0123456789abdcdefghi
6  Copying memory 8 bytes at a time
7  Source: 0123456789abdcdefghi. Destination: 0123456789abdcdefghi
8  mymemcpy4: copied 0 bytes up to 64-byte alignment
9  mymemcpy4: copied 16 bytes 8 bytes at a time
10 mymemcpy4: copied last 5 bytes one at a time
11 Source: 0123456789abdcdefghi. Destination: 0123456789abdcdefghi
```

With mymemcpy4b:

```
1  ./a.out
2  copied 1 bytes at a time up to 0 bytes
3  copied 8 bytes at a time up to 32 bytes
4  copied up to 40 bytes
5  src: 1234567890abcdefghi1234567890ABCDEFGHI
6  dest: 1234567890abcdefghi1234567890ABCDEFGHI
7
8   ./a.out
9  copied 1 bytes at a time up to 16 bytes
10 copied 8 bytes at a time up to 32 bytes
11 copied up to 40 bytes
12 src: 1234567890abcdefghi1234567890ABCDEFGHI
```

```
13 dest: 1234567890abcdefghi1234567890ABCDEFGHI
```

Pointer arithmetic

I was looking at what the '+' and '-' operators actually performs when applied to pointers and arrays, as well as other tricks, and wrote some programs to test it:

```
#include <stdio.h>

main()
{
    int Arr[16];
    unsigned int a0 = (unsigned int) &Arr[0];
    unsigned int a3 = (unsigned int) &Arr[3];
    printf("%u\n", a3 - a0);
    printf("%u\n", &Arr[3] - &Arr[0]);
}
```

The result is this:

```
12
3
```

This other program explores this feature and some others:

```
#include <stdio.h>

int main()
{
    int Arr[16];
    Arr[0] = 9;
    Arr[3] = 5;
    printf("Arr[3]:      \t%d\n",Arr[3]);
    printf("*(Arr + 3):  \t%d\n",*(Arr + 3));
    printf("*(3 + Arr):  \t%d\n",*(3 + Arr));
```

```
    printf("*(&Arr[0] + 3):\t%d\n",*(&Arr[0] + 3)); // &Arr[0] is the same a
    printf("3[Arr]:      \t%d\n",3[Arr]); // gets converted by compiler to *
    printf("\n");

    // showing how + operation behaves when used with addresses of pointers
    printf("Arr:         \t%u\n", Arr);
    printf("&Arr[0]:     \t%u\n", &Arr[0]);
    printf("&Arr[0] + 1: \t%u\n", &Arr[0] + 1);
    printf("&*(&Arr[0] + 1):%u\n", &*(&Arr[0] + 1));
    printf("sizeof(Arr): \t%d\n", sizeof(Arr));
    printf("sizeof(Arr[0]):\t%d\n", sizeof(Arr[0]));
    printf("sizeof(int):\t%d\n", sizeof(int));
    printf("\n");

    printf("sizeof(&Arr[0]):%d\n", sizeof(&Arr[0]));
    printf("sizeof(unsigned long):%d\n", sizeof(unsigned long));
    printf("\n");


    unsigned long ArrAddr = (unsigned long)&Arr[0];
    printf("ArrAddr: %u\n", ArrAddr);
    int *p;
    p = ArrAddr;
    printf("*p: %d\n", *p);
    printf("*(p+3): %d\n", *(p+3));
    printf("\n");

    // breaking operator overloading of +
    unsigned long ArrAddr3 = ArrAddr + 3;
    p = ArrAddr3;
    printf("ArrAddr3: %u\n", ArrAddr3);
    printf("*p: %d\n", *p);
    printf("\n");

    // doing manually what overloading of + does automatically
    ArrAddr3 = ArrAddr + 3*sizeof(Arr[0]);
    p = ArrAddr3;
    printf("ArrAddr3: %u\n", ArrAddr3);
    printf("*p: %d\n", *p);
    printf("\n");



    return 0;

}
```

The result is this:

```
Arr[3]:         5
*(Arr + 3):     5
*(3 + Arr):     5
*(&Arr[0] + 3): 5
3[Arr]:         5

Arr:            1587159312
&Arr[0]:        1587159312
&Arr[0] + 1:    1587159316
&*(&Arr[0] + 1):1587159316
sizeof(Arr):    64
sizeof(Arr[0]): 4
sizeof(int):    4

sizeof(&Arr[0]):8
sizeof(unsigned long):8

ArrAddr: 1587159312
*p: 9
*(p+3): 5

ArrAddr3: 1587159315
*p: 0

ArrAddr3: 1587159324
*p: 5
```

Unions and Structures

The main difference between an union and a structure is that for a structure, all the structure members are stored contiguously. However, for an union the members are stored overlapping, i.e. one of top of each other.

```
#include <stdio.h>

int main()
{
    struct ex_struct {
        char element1;  // 1 byte
        int element2;   // 4 bytes
        long element3;  // 8 bytes
    };
    union ex_union {
        char element1;
        int element2;
        long element3;
    };

    struct ex_struct a;
    union ex_union b;

    printf("Size of structure: %d\n", sizeof(a));
    printf("Size of union: %d\n", sizeof(b));

    a.element1 = 'D';
    a.element2 = 59;

    b.element1 = 'D';
    b.element2 = 59;

    printf("a.element1 = %c\n", a.element1);
    printf("a.element2 = %d\n", a.element2);
    printf("b.element1 = %c\n", b.element1);
    printf("b.element2 = %d\n\n", b.element2);

    b.element3 = 0x333333333;
    b.element1 = 0x11;
    printf("b.element3 = 0x%x\n", b.element3);
    printf("b.element1 = 0x%x\n", b.element1);

    return 0;
}
```

This is the output:

```
Size of structure: 16
Size of union: 8
a.element1 = D
a.element2 = 59
b.element1 = ;
b.element2 = 59

b.element3 = 0x33333311
b.element1 = 0x11
```

Unions are useful when a storage location is needed to represent different data types, which is similar to the concept of polymorphism in C++.

Dynamic allocation of memory for matrix (2D array) in C

I was working on an assignment that requires dynamic memory allocation for a matrix. A simple way to do it would be to use just an array. However that would not allow you to do matrix-like indexing such as matrix[i][j]. So a better way is this:

```
// dynamically allocating memory for matrix (2D array)
int **matrix;
int i, j;
int size = 5; // let's assume matrix is square (size x size)
matrix = (int **)malloc(size * sizeof(int*));
    for (i = 0; i < size; i++)
        matrix[i] = (int *)malloc(size * sizeof(int));

matrix[3][1] = 16;
```

Pointers in C

This is a small program that plays with pointers and demonstrate some features of pointers in C:

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

int main()
{
    /* variables */
    int    a = 25;
    float  b = 3.142;
    double c = 1.73217;
    char   d[]="Hello, C Programmer"; /* array of characters, which is equiva
    char   e = 'x';

    /* pointers to variables */
    int    *pa = &a; /* points to variable a */
    float  *pb = &b; /* points to variable b */
    double *pc = &c; /* points to variable c */
    char   *pd = d;  /* points to first element of array d*/
    char   *pe = &e; /* points to variable e */

    /* pointer to pointer */
    int    **ppa;
    ppa = &pa;    /* double pointer points to pointer pa */

    /* value of the variables */
    printf("Value of integer variable a: %d\n", a);
    printf("Value of floating variable b: %f\n", b);
    printf("Value of the double variable c: %f\n", c);
    printf("Value of string d: %s\n", d);
    printf("Value of character e: %c\n", e);
    printf("\n");

    /* value of variables pointers point to */
    /* prints the value of the variables pointed by the pointers */
    printf("Value of variable that pa points to: %d\n", *pa);
    printf("Value of variable that pb points to: %f\n", *pb);
    printf("Value of variable that pc points to: %f\n", *pc);
    printf("Value of variable that pd points to: %c\n", *pd);
    printf("Value of variable that pe points to: %c\n", *pe);
    printf("Value of variable that ppa points to points to: %d\n", **ppa);
    printf("Value of variable that ppa points to: %u\n", *ppa);
```

```
    printf("\n");

    /* addresses of variables */
    /* gets the addresses of the variables by deferencing the variables */
    printf("Address of a: %u\n", &a);
    printf("Address of b: %u\n", &b);
    printf("Address of c: %u\n", &c);
    printf("Address of d: %u\n", d);
    printf("Address of e: %u\n", &e);
    printf("\n");

    /* addresses of pointers */
    /* gets the address of the pointers (where the pointers are in memory) by
    printf("Address of pointer pa: %u\n", &pa);
    printf("Address of pointer pb: %u\n", &pb);
    printf("Address of pointer pc: %u\n", &pc);
    printf("Address of pointer pd: %u\n", &pd);
    printf("Address of pointer pe: %u\n", &pe);
    printf("Address of pointer ppa: %u\n", &ppa);
    printf("\n");

    /* addresses pf variables pointed by pointers */
    /* this should give the same result as the section 'addresses of variable
    printf("Address of variable pointed by pa: %u\n", &*pa);
    printf("Address of variable pointed by pb: %u\n", &*pb);
    printf("Address of variable pointed by pc: %u\n", &*pc);
    printf("Address of variable pointed by pd: %u\n", &*pd);
    printf("Address of variable pointed by pe: %u\n", &*pe);
    printf("Address of variable pointed by pointer pointed by ppa: %u\n", &*
    printf("Address of pointer pointed by ppa: %u\n", &*ppa);
    printf("\n");

    return 0;
}
```

This is the output from the program above:

```
Value of integer variable a: 25
Value of floating variable b: 3.142000
Value of the double variable c: 1.732170
Value of string d: Hello, C Programmer
Value of character e: x

Value of variable that pa points to: 25
Value of variable that pb points to: 3.142000
Value of variable that pc points to: 1.732170
Value of variable that pd points to: H
Value of variable that pe points to: x
Value of variable that ppa points to points to: 25
Value of variable that ppa points to: 2316519116

Address of a: 2316519116
Address of b: 2316519112
Address of c: 2316519104
Address of d: 2316519072
Address of e: 2316519071

Address of pointer pa: 2316519056
Address of pointer pb: 2316519048
Address of pointer pc: 2316519040
Address of pointer pd: 2316519032
Address of pointer pe: 2316519024
Address of pointer ppa: 2316519016

Address of variable pointed by pa: 2316519116
Address of variable pointed by pb: 2316519112
Address of variable pointed by pc: 2316519104
Address of variable pointed by pd: 2316519072
Address of variable pointed by pe: 2316519071
Address of variable pointed by pointer pointed by ppa: 2316519116
Address of pointer pointed by ppa: 2316519056
```