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) wh
arguments and returns a strin
 found in both strings in the
  is order N-squared and one w
*/



int getlenght1(const char s[]
    int k = 0;
    while(s[k++]); // same as
                   // and same
    k--;
```

```
    printf("size %d\n", k);
    return k;
}

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

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

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

/* Review of bitwise operatio
    && is logical operator: re
    & is bitwise operator, app
    set bit x:              vb
    set bits x and y:       vb
    clear bit x:            vb
    toggle (change) bit x:  vb
    check if bit x set:     i
    get lower x bits:       v
    get higher x bits:      v
 */
char * solution_linear(const c
    int i, letter, k;
    unsigned long bitarray = (
    i = 0;
    /* scan b string */
    while (b[i]) {
        letter = b[i] - 'a' +
```

```
        bitarray |= 0x1<<lett
        i++;
    }
    /* now scan a, so common 
    char *common = malloc(siz
    i = 0; k = 0;
    while (a[i]) {
        letter = a[i] - 'a' +
        if (bitarray & (0x1<<
            common[k++] = a[i
        }
        i++;
    }
    common[k] = '\0';

    /* 'common' was allocated
        returns. But it needs
    return common;
}

int main() {
    char *a = "asdfqwer";
    char b[] = "skelrpfa";
    char *common1 = solution_
```

```
    char *common2 = solution_
    printf("a: %s\nb: %s\ncomm
            a, b, common1, co
    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
```

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.

```
#include <stdio.h>

int main (){

    int v[] = {5, 20};
    int *p = v;
    printf("p: %d, ", (*p)++);
    printf("p: %d, ", *p++);
    printf("p: %d\n", *p);

    /* pointer to constant da
    const int *p2 = v;
    //int const *p2 = v; // s
    //printf("const ptr: %d, 
    printf("const ptr: %d, ",
    //printf("const ptr: %d, 
    printf("const ptr: %d\n",

    /* constant pointer (can'
    int * const p3 = v;
    //int * p3 const = v;
    printf("ptr to const data
```

```
    //printf("ptr to const da
    printf("ptr to const data

    /* constant pointer to co
    const int * const p4 = v;
    // int const * const p4; 
    //printf("const ptr to co
    //printf("const ptr to co
    printf("const ptr to cons

    /* array definition is eq
    printf("ptr from array: %
    //printf("ptr from array:
    printf("ptr from array: %

    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 ptr to const data: p4: 
```

```
ptr from array: 7, ptr from a
```

Graphs problems

A graph can be defined in computer science as a data structure with a set of vertices (or nodes) and a set of edges (connection between nodes). The easiest way to represent a graph is with a matrix.

There are several types of problems that can be resolve with a graph. The following summarizes the problems, and the algorithms to solve them:

-Exploring graph: given a node s, find how to reach all other nodes

- Depth-first search
- Breadth-first search

-Single-source shortest path: given a source s, find the shortest path to all other vertices.

- Dijkstra's algorithm

-All pairs shortest path: get the shortest path from any node to any node. Gives matrices dist[u][v] and pred[u][v]

- Floyd-Warshall algorithm

-Minimum Spanning Tree: spanning tree is the set of edges that ensures all vertices are connected. Minimum ST is the set of edges with minimim total weight that ensures all vertices are connected.

- Prim's algorithm

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 re
is all characters in a string
unique, and false otherwise
*/

// time O(n^2). No extra spac
int uniqueCharac1(char *ptr1)
    char *ptr2;
    while (*ptr1++ != '\0') {
        ptr2 = ptr1;
        while (*ptr2++ != '\0
            if (*ptr2 == *ptr
                printf("Lette
```

```
                return FALSE;
            }
        }
    }
    return TRUE;
}

//time O(n). Extra space O(1)
int uniqueCharac2(char *ptr)
    unsigned int bit_field =
    // int gives 4 bytes (32
    int character;
    while (*ptr++ != '\0') {
        character = *ptr - 'a
        //printf("Letter %c i
        // checking if bit se
        if (bit_field && (1<<
            printf("Letter %c
            return FALSE;
        }
        // setting bit
        bit_field |= (1&lt;&l
    }
    return TRUE;
```

```
}

char * quicksort(char *string
    return string;
}
int binarysearch(char *string,
    return TRUE;
}

//time O(n*log n). No extra sp
int uniqueCharac3(char *ptr)
    char *sorted = quicksort(p
    while (*ptr++ != '') {
        if (binarysearch(ptr -
            printf("Letter %c
            return TRUE;
        }
    }
    return FALSE;
}

int main(int argc, char **argv
    char string[] = "asdfqwers
    if (uniqueCharac1(string)
```

```
    if (uniqueCharac2(string)
}
```

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 mymemcpy2(void *, const 
```

```
void mymemcpy3(void *, const v
void mymemcpy4(void *, const v

int main(int argc, char **argv
 char source[] = &amp;quot;012
    char dest[100];

    // void * memcpy ( void *
    memcpy(dest, source, strle
    printf(&amp;quot;Source: %

    strcpy(source, &amp;quot;0
    mymemcpy1(dest, source, st
    printf(&amp;quot;Source: %

    strcpy(source, &amp;quot;0
    mymemcpy2(dest, source, st
    printf(&amp;quot;Source: %

    strcpy(source, &amp;quot;0
    mymemcpy3(dest, source, st
    printf(&amp;quot;Source: %

    strcpy(source, &amp;quot;0
```

```
    mymemcpy4(dest, source, s
    printf(&amp;quot;Source: 
    return 0;
}

// simple implementation
void mymemcpy1(void *dest, co
    int i = 0;
    // casting pointers
    char *dest8 = (char *)des
    char *source8 = (char *)s
    printf(&amp;quot;Copying 
    for (i = 0; i &amp;lt; nu
        dest8[i] = source8[i]
    }
}
// it checks that destination
// source memory
void mymemcpy2(void *dest, co
    int i = 0;
    // casting pointers
    char *dest8 = (char *)des
    char *source8 = (char *)s
    printf(&amp;quot;Copying 
```

```
    for (i = 0; i &amp;lt; nur
        // make sure destinat:
        if (&amp;amp;dest8[i]
            printf(&amp;quot;(
            return;
        }
        dest8[i] = source8[i];
    }
}

// copies 1 word at a time (8
void mymemcpy3(void *dest, co:
    int i = 0;
    // casting pointers
    long *dest32 = (long *)de:
    long *source32 = (long *):
    char *dest8 = (char *)dest
    char *source8 = (char *)s(

    printf(&amp;quot;Copying r
    for (i = 0; i &amp;lt; nur
        if (&amp;amp;dest32[i]
            printf(&amp;quot;(
            return;
```

```
        }
        dest32[i] = source32[
    }
    // copy the last bytes
    i*=sizeof(long);
    for ( ; i &amp;lt; num; i
        dest8[i] = source8[i]
    }
}

/* memory addres is n-byte al
   b-bit aligned is equivalen
   padding = n - (offset &amp
   aligned offset = (offset +

   Copies 8 bytes at a time w
*/
void mymemcpy4(void * dest, c

#define NBYTE 8 // n-byte bou

    int i = 0;
    int j = 0;
```

```
// short and long pointers
//  bytes at a time
long * destlong = (long *)
long * sourcelong = (long
char * destshort = (char
char * sourceshort = (char


// copy first bytes until
while((((uintptr_t)&amp;ar
       (&amp;amp;destshort
    destshort[i] = sources
    i++;
}
printf(&amp;quot;%s: copie
       __func__, i, NBYTE


// now copy 8 (sizeof(lon
// align destination poin
destlong = ((uintptr_t)des
 // continue copying where
sourcelong = (long *)sour
// j+1 * 8 - 1 to avoid co
```

```
    while ((j+1)*sizeof(long)
            &amp;amp;destlong
        destlong[j] = sourcel
        j++;
    }
    printf(&amp;quot;%s: copi
           j*sizeof(long), si


    // finally copy last byte
    i = i + j*sizeof(long);
    int prev_i = i;
    while(i &amp;lt; size &amp;am
        destshort[i] = source
        i++;
    }
    printf(&amp;quot;%s: copi
            __func__, i-prev_

}

/* simpler implementation of
void mymemcpy4b(const void *
```

```
    char * dest1 = (char *)des
    char * src1 = (char *)src
    int n = 0;
#define NBYTE 64
    // copy up to nbyte alignm
    while ((((uintptr_t)dest1
        *dest1++ = *src1++;
        n++;
    }
    printf("copied %d bytes a

    // copy up to end minus la
    long * dest2 = (long *)des
    long * src2 = (long *)src1
    while (n &lt; size - sizeo
        *dest2++ = *src2++;
        n+=sizeof(long);
    }
    printf("copied %d bytes a

    // copy last bytes
    src1 = (char *)src2;
    dest1 = (char *)dest2;
    while (n &lt; size) {
```

```
        *dest1++ = *src1++;
        n++;
    }
    printf("copied up to %d by
}
```

The result is this:

```
Source: 0123456789abdcdefghi.
Copying memory 1 byte(s) at a
Source: 0123456789abdcdefghi.
Copying memory 1 byte(s) at a
Source: 0123456789abdcdefghi.
Copying memory 8 bytes at a ti
Source: 0123456789abdcdefghi.
mymemcpy4: copied 0 bytes up t
mymemcpy4: copied 16 bytes 8 b
mymemcpy4: copied last 5 bytes
Source: 0123456789abdcdefghi.
```

With mymemcpy4b:

```
 ./a.out
copied 1 bytes at a time up to
copied 8 bytes at a time up to
copied up to 40 bytes
src: 1234567890abcdefghi12345
dest: 1234567890abcdefghi1234

 ./a.out
copied 1 bytes at a time up to
copied 8 bytes at a time up to
copied up to 40 bytes
src: 1234567890abcdefghi12345
dest: 1234567890abcdefghi1234
```

Reversing string

This is a very common C programming interview question. I wrote a few ways to reverse a string:

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

```
#include <stdlib.h>

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

char * reverse_newstring(const
void reverse_givenstring(char

int main()
{
        char a[] = "abcde"; //
        printf("%s\n", a);
        printf("%s\n", reverse
        reverse_givenstring(a)
        printf("%s\n", a);
        return 0;
}

char * reverse_newstring(const
{
        int i;
        int length = 0;
        while (*(a + length++)
        length--; // remove
        char *res = malloc((le
```

```
        for (i = 0; i &lt; le
                res[i] = a[le
        res[i] = '\0'; // ter
        return res;
        /* the problem with th
           but it doesnt free
           global and free it
        */
}

void reverse_givenstring(char
{
        int i;
        int length = 0;
        char tmp;
        while (a[length++] !=
        length--; // dont wan
        for (i = 0; i &lt; le
        {
                //swapping 1
                /*
                tmp = a[i];
                a[i] = a[leng
                a[length-i-1]
```

```
                    */
                    //swapping 2
                    /*
                    a[i] = a[i] +
                    a[length-i-1]
                    a[i] = a[i] -
                    */
                    //swapping 3
                    a[i] ^= a[len
                    a[length-i-1]
                    a[i] ^= a[len
                    //swapping 4
                    //SWAP(a[i], a
            }
            a[length] = '\0' // te
    }
```

This is the result I got:

```
abcde
edcba
```

```
edcba
```

Count number of bits set to 1 in variable

This is a simple program to count the number of bits set to 1 in a variable or register. A simple function to get the binary representation of a number was also written:

```
#include <stdio.h>

/* intx_t and uintx_t (x=8,16,
/* other data types need to be

typedef unsigned long uint16;

int countBitsSet(uint16 x)
{
    int count;
    for (count = 0; x; count+
        x &= x-1; /* x & x-1 :
    return count;
}
char * printBinary(uint16 x)
```

```
{
    int i = 0;
    static char buff[17]; /* s
                          /* p
    unsigned long mask;
    mask = (1 << 16);
    for(i = 0; i < 17; i++)
    {
        buff[i] = (x & mask)?
        mask >>= 1;
    }
    buff[i] = '\0';

    return buff;
}
int main()
{
    uint16 a = 8764;
    printf("Number: %u, %sb\n'
    printf("Number of bits se
    return 0;
}
```

This is the result

```
>./a.out
Number: 8764, 00010001000111100
Number of bits set to one: 6
```