

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) with two string arguments and returns a string containing only the characters found in both strings in the order of the first string. The time complexity is order N-squared and one memory allocation. */

int getlength1(const char s[])
{
    int k = 0;
    while(s[k++] != '\0'); // same as while(s[k])
    while(s[k] == '\0') // and same as while(s[k] != '\0')
        k--;
}
```

```

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

```

```

int getlength2(const char * s)
{
    const char * s0 = s;
    while (*s++); // wouldn't work
    printf("size %s: %d\n", s0, s - s0);
    return (s - s0 - 1);
}

```

```

char * solution_nsquare(const char *a, const char *b)
{
    char *common;
    int i, j, k;
    int na = getlength2(a);
    int nb = getlength2(b);
    common = malloc(sizeof(char) * (na + nb));
    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 operation
    && is logical operator: re
    & is bitwise operator, app
    set bit x:                vl
    set bits x and y:         vl
    clear bit x:              vl
    toggle (change) bit x:    vl
    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 = 0;
    i = 0;
    /* scan b string */
    while (b[i]) {
        letter = b[i] - 'a' +

```

```

        bitarray |= 0x1<<letter;
        i++;
    }
    /* now scan a, so common is
    char *common = malloc(sizeof(char)*size);
    i = 0; k = 0;
    while (a[i]) {
        letter = a[i] - 'a' + 1;
        if (bitarray & (0x1<<letter))
            common[k++] = a[i];
        i++;
    }
    common[k] = '\0';

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

```

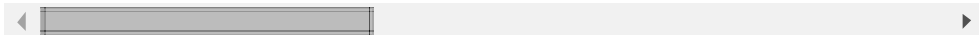
```

int main() {
    char *a = "asdfqwer";
    char b[] = "skelrpfa";
    char *common1 = solution_1(a, b);
}

```

```
char *common2 = solution_  
printf("a: %s\nb: %s\ncom  
      a, b, common1, cor  
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 data
    const int *p2 = v;
    //int const *p2 = v; // same
    //printf("const ptr: %d, ", *p2);
    printf("const ptr: %d, ", *p2);
    //printf("const ptr: %d, ", *p2);
    printf("const ptr: %d\n", *p2);

    /* constant pointer (can't change pointer)
    int * const p3 = v;
    //int * p3 const = v;
    printf("ptr to const data: %d\n", *p3);
```

```

    //printf("ptr to const data: %d\n", *p4);
    printf("ptr to const data: %d\n", *p4);

    /* constant pointer to const data
    const int * const p4 = v;
    // int const * const p4; ,
    //printf("const ptr to const data: %d\n", *p4);
    //printf("const ptr to const data: %d\n", *p4);
    printf("const ptr to const data: %d\n", *p4);

    /* array definition is equivalent
    printf("ptr from array: %d\n", *p4);
    //printf("ptr from array: %d\n", *p4);
    printf("ptr from array: %d\n", *p4);

    return 1;
}

```

gcc sorting2.c

./a.out

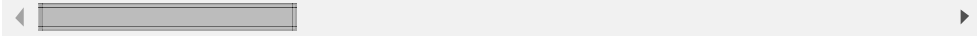
p: 5, p: 6, p: 20

const ptr: 6, const ptr: 20

ptr to const data: 6, ptr to const data: 20

const ptr to const data: p4: 6

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 resolved 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 $\text{dist}[u][v]$ and $\text{pred}[u][v]$

- 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 minimum 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 than it may look like first. I implemented a few solutions in C.

```
#define FALSE 0
#define TRUE 1

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

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

```

        return FALSE;
    }

}

return TRUE;
}

```

```

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

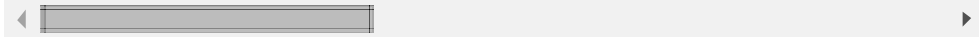
```
}
```

```
return FALSE;
```

```
}
```

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

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



memcpy implementation

I was asked once during an interview to implement memcpy in C. The question 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 one improves performance by copying one word at a time, instead of 1 byte at a time. The fourth one copies a word at a time, and starts 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 *, rsize_t);  
void mymemcpy2(void *, const void *, rsize_t);
```

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

```
int main(int argc, char **argv)
{
    char source[] = "&quot;01";
    char dest[100];
```

```
    // void * memcpy ( void *
    memcpy(dest, source, strlen(source));
    printf("&quot;Source: %s\n", source);
```

```
    strcpy(source, "&quot;01");
    mymemcpy1(dest, source, strlen(source));
    printf("&quot;Source: %s\n", source);
```

```
    strcpy(source, "&quot;01");
    mymemcpy2(dest, source, strlen(source));
    printf("&quot;Source: %s\n", source);
```

```
    strcpy(source, "&quot;01");
    mymemcpy3(dest, source, strlen(source));
    printf("&quot;Source: %s\n", source);
```

```
    strcpy(source, "&quot;01");
```

```

        memcpy4(dest, source, size);
        printf("&quot;Source: %s\n", source);
        return 0;
}

```

```

// simple implementation
void memcpy1(void *dest, const void *source, size_t n)
{
    int i = 0;
    // casting pointers
    char *dest8 = (char *)dest;
    char *source8 = (char *)source;
    printf("&quot;Copying %d bytes\n", n);
    for (i = 0; i < n; i++)
        dest8[i] = source8[i];
}
}

```

```

// it checks that destination
// source memory
void memcpy2(void *dest, const void *source, size_t n)
{
    int i = 0;
    // casting pointers
    char *dest8 = (char *)dest;
    char *source8 = (char *)source;
    printf("&quot;Copying %d bytes\n", n);
}

```

```

        for (i = 0; i &lt; n; i++)
        {
            // make sure destination is not null
            if (&dest8[i] == 0)
            {
                printf("&quot;Destination address is null.&quot;\n");
                return;
            }
            dest8[i] = source8[i];
        }
    }
}

```

```

// copies 1 word at a time (8 bytes)
void mymemcpy3(void *dest, const void *source, int n)
{
    int i = 0;
    // casting pointers
    long *dest32 = (long *)dest;
    long *source32 = (long *)source;
    char *dest8 = (char *)dest;
    char *source8 = (char *)source;

    printf("&quot;Copying 1 word at a time.&quot;\n");
    for (i = 0; i < n; i++)
    {
        if (&dest32[i] == 0)
        {
            printf("&quot;Destination address is null.&quot;\n");
            return;
        }
    }
}

```

```

        }
        dest32[i] = source32[i];
    }
    // copy the last bytes
    i*=sizeof(long);
    for ( ; i &lt; num; i+=4)
        dest8[i] = source8[i];
    }
}

```

```

/* memory address is n-byte aligned
   b-bit aligned is equivalent to
   padding = n - (offset & (b-1))
   aligned offset = (offset + padding)

```

```

    Copies 8 bytes at a time with
    */

```

```

void mymemcpy4(void * dest, const void * src, int num)

```

```

#define NBYTE 8 // n-byte boundary

```

```

    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)&ar
    (&destshort
    destshort[i] = source
    i++;
}
printf("&quot;%s: copied
    __func__, i, NBYTES

```

```

// now copy 8 (sizeof(long
// align destination point
destlong = ((uintptr_t)de
    // continue copying where
sourcelong = (long *)sourc
// j+1 * 8 - 1 to avoid co

```

```

while ((j+1)*sizeof(long)
        && destlong
        destlong[j] = source
        j++;
}
printf("&quot;%s: copied %d long words (%d bytes) from %s to %s\n",
        j*sizeof(long), size,
        source, dest, dest);

// finally copy last bytes
i = i + j*sizeof(long);
int prev_i = i;
while(i &&lt; size && destshort
        destshort[i] = source
        i++;
}
printf("&quot;%s: copied %d bytes from %s to %s using %s\n",
        __func__, i-prev_i,
        source, dest, __func__);
}

/* simpler implementation of memcpy
void mymemcpy4b(const void * src, void * dest, r

```

```

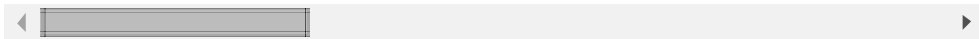
    char * dest1 = (char *)dest;
    char * src1 = (char *)src;
    int n = 0;
#define NBYTE 64
    // copy up to nbyte aligned
    while (((uintptr_t)dest1 & 0xf) != 0) {
        *dest1++ = *src1++;
        n++;
    }
    printf("copied %d bytes at %p\n", n, dest1);

    // copy up to end minus last aligned
    long * dest2 = (long *)dest;
    long * src2 = (long *)src;
    while (n < size - sizeof(long)) {
        *dest2++ = *src2++;
        n+=sizeof(long);
    }
    printf("copied %d bytes at %p\n", n, dest1);

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

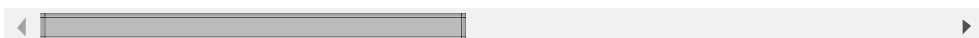
```

```
        *dest1++ = *src1++;  
        n++;  
    }  
    printf("copied up to %d bytes\n", n);  
}
```



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 t  
Source: 0123456789abdcdefghi.  
memcpy4: copied 0 bytes up t  
memcpy4: copied 16 bytes 8 b  
memcpy4: 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: 1234567890abcdefghi1234567890
dest: 1234567890abcdefghi1234567890
```

```
./a.out
copied 1 bytes at a time up to
copied 8 bytes at a time up to
copied up to 40 bytes
src: 1234567890abcdefghi1234567890
dest: 1234567890abcdefghi1234567890
```



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; }
```

```
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 < len; i++)
            res[i] = a[len-i-1];
        res[i] = '\0'; // terminate string
        return res;
    }

    /* the problem with this approach is that it doesn't free the memory
       but it doesn't free the global and free it
    */
}

```

```

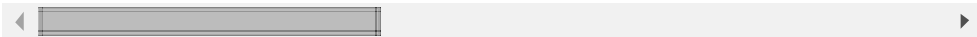
void reverse_givenstring(char a[])
{
    int i;
    int length = 0;
    char tmp;
    while (a[length++] != '\0')
        length--; // don't want to include the null terminator
    for (i = 0; i < length/2; i++)
    {
        //swapping 1
        /*
        tmp = a[i];
        a[i] = a[length-i-1];
        a[length-i-1] = tmp;
        */
    }
}

```

```

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

```



This is the result I got:

abcde

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=8,16,32,64)
/* other data types need to be defined

typedef unsigned long uint16;

int countBitsSet(uint16 x)
{
    int count;
    for (count = 0; x; count++)
        x &= x-1; /* x & x-1 : clears the rightmost 1
    return count;
}

char * printBinary(uint16 x)
```

```

{
    int i = 0;
    static char buff[17]; /* :
                                /* }

    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", a, "0123456789ABCDEF");
    printf("Number of bits set: %d\n", count_bits(a));
    return 0;
}

```

This is the result

```
>./a.out
```

```
Number: 8764, 0001000100011110
```

```
Number of bits set to one: 6
```

