This document contains the questions and solutions to the CS107 midterm given in Spring 2016 by instructors Julie Zelenski and Michael Chang. This was an 80-minute exam.

Midterm questions

Problem 1: C-strings

The Pascal language uses a different memory layout for strings than C. The first byte of a Pascal string is the number of characters and is followed by the sequence of characters. It contains no terminating null char. The char array below is a Pascal string equal to the C-string "exemplified":

11	**'e'**	**'x'**	**'e'**	**'m'**	**'p'**	**'l'**	**'i'**	**'f'**	**'i'**	**'e'**	**'d'**

a) Write the function **`equal`** to compare a Pascal string and C-string for string equality. The function arguments are **`pstr`**, a char array representing a Pascal string, and **`cstr`**, an ordinary C-string. The return value should be **`true`** if the two strings are equal, **`false`** otherwise.

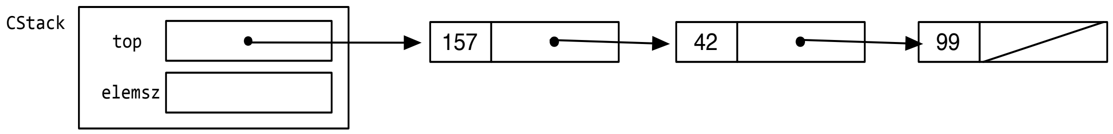
```
bool equal(const unsigned char pstr[], const char *cstr)
{
```

b) Given this design, what is the maximum allowable length for a Pascal string?

Problem 2: Implementation of C generics

A *stack* is a LIFO (last-in-first-out) collection. The *push* operation adds an element to the top of the stack, *peek* accesses the topmost element, and *pop* removes it. The **`CStack`** is a generic implementation that can store elements of any type. It uses a linked list to store elements.

For example, a **`CStack`** is created to store elements of **`sizeof(int).`** After pushing three values (99, 42, 157), its internal memory is diagrammed below. The last value pushed (157) is topmost.



Review the **`CStack`** struct definition and the **`cstack_create`** and **`cstack_peek`** functions below:

```
typedef struct CStackImplementation {
    void *top;
    size_t elemsz;
} CStack;

CStack *cstack_create(size_t elemsz)
{
    CStack *cs = malloc(sizeof(CStack));
    cs->elemsz = elemsz;
    cs->top = NULL;
    return cs;
}
```

```
// Peek accesses the topmost element of stack without removing it.
// Returns pointer to memory location where element value is stored;
// returns NULL if stack is empty
void *cstack_peek(CStack *cs)
{
   return cs->top;
}
```

You are to implement the operations `cstack_push` and `cstack_multipush`. Your functions must work with the code above unmodified.

A few important notes:

- The cell layout must be exactly as diagrammed above. A cell stores the element value followed by the pointer to the next cell. If storing elements of type `char`, `elemsz` will be 1 byte and each cell will occupy 9 bytes.
- The `cstack_multipush` operation should layer on `cstack_push`, not re-implement it.

```
 /* Function: cstack_push(cs, &val)
  * -------------------------------
  * Adds a new element to the top of the stack. addr is expected to be a valid pointer.
  * The element's value is copied from the memory pointed to by addr.
  */
void cstack_push (CStack *cs, const void *addr)
{
```

```
 /* Function: cstack_multipush(cs, arr, n)
  * ------------------------------------
  * Pushes an array of elements onto stack. The two arguments are arr, the array's
  * base address, and nelems, the count of elements in arr. Each array element is
  * cs->elemsz bytes. Elements are pushed to stack in order they appear in arr; the
  * last element in array ends up topmost on the stack.
  */
void cstack_multipush (CStack *cs, const void *arr, int nelems)
{
```

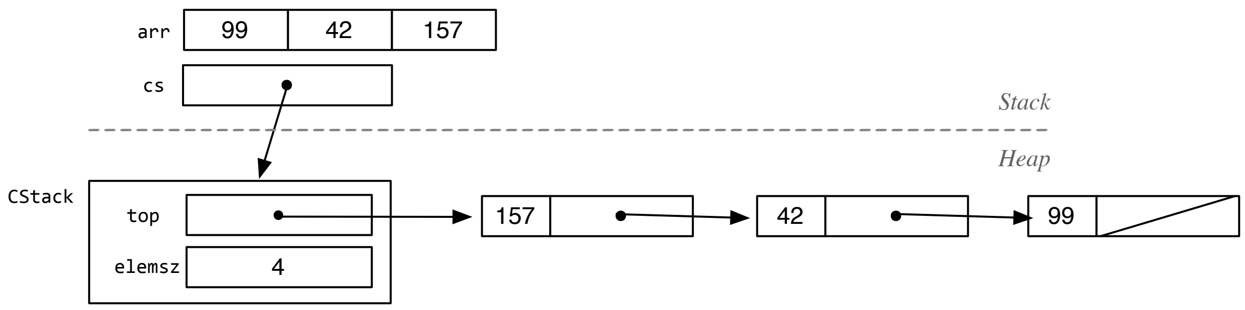
Problem 3: Generic client

Assume `CStack` is correctly implemented as documented in the previous problem. The sample client program below demonstrates correct use of `CStack` to store integers.

```
int main(void)
{
    int arr[3] = {99, 42, 157};
    CStack *cs = cstack_create(sizeof(int));

    cstack_multipush(cs, arr, 3);
    printf("%d\n", *(int *)cstack_peek(cs));
    return 0;
}
```

The program prints the value **157**. Below is a diagram that shows the state of memory at the time of the **return** statement.



On the following page is another client program that exercises the **CStack**. Not all of the calls are sensible, but the program compiles cleanly and successfully runs to completion. You are to trace its execution, describe what it prints, and draw a memory diagram. Your diagram should show all allocated memory and its contents, just as modeled in the sample diagram above.

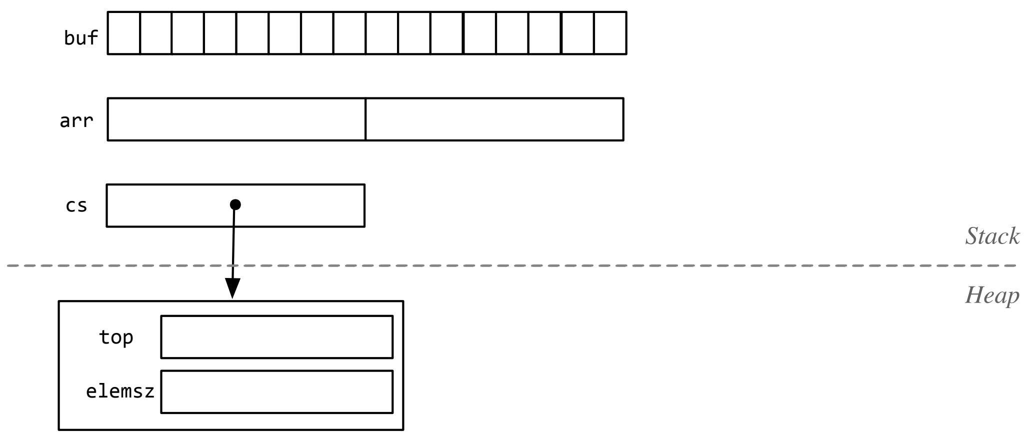
A few guidelines for memory diagrams:

- Show all allocated memory, both stack and heap.
- Draw pointers as arrows. Represent a NULL pointer with a diagonal slash.
- Use **?** to mark where contents of memory are uninitialized.

```
int main(void)
{
    char buf[16];
    char *arr[2] = { NULL, NULL };
    CStack *cs = cstack_create(sizeof(char *));
                                            // initial diagram
    strcpy(buf, "binky winky");
    arr[0] = buf;
    cstack_multipush(cs, arr, 2);
    *arr += 6;
    arr[1] = strdup(&buf[2]);
    for (int i = 0; i < 2; i++)
        cstack_push(cs, arr[i]);
    printf("%s\n", (char *)cstack_peek(cs));
    return 0;                              // complete diagram at this point
}
```

a) You run the program several times and each time its output is slightly different. Explain why.

b) The diagram started below shows the state of memory after the variable declarations. Complete the diagram to show all allocated memory and its contents at `return`.



Problem 4: Bits, bytes, and numbers

Match each bitwise expression below on the left to an equivalent C expression from the choices on the right. Assume the variable **`x`** is a 32-bit signed integer using two's complement representation. **`INT_MAX`** and **`INT_MIN`** are the maximum and minimum representable signed integers. Shifts on signed values are arithmetic, not logical, shifts.

`x + INT_MAX + INT_MIN`

☐

`(x ^ -1) + 1`

☐

`((x >> 31) & x)`

☐

A. `(x < 0) == (x % 2)`
B. `1 - x`
C. `x`
D. `0`
E. `x + 1`
F. `INT_MAX`
G. `x - 1`
H. `(x < 0) ? INT_MIN : 0`
I. `~x`
J. `INT_MIN`
K. `(x < 0) ? x : 0`
L. `-x`
M. `(x < 0) ? (x % 2) : 0`
N. `x % 2`

Solutions

Problem 1:

```
bool equal(const unsigned char pstr[], const char *cstr)
{
   return strlen(cstr) == pstr[0] && strncmp(pstr+1, cstr, pstr[0]) == 0;
}

UCHAR_MAX = 255
```

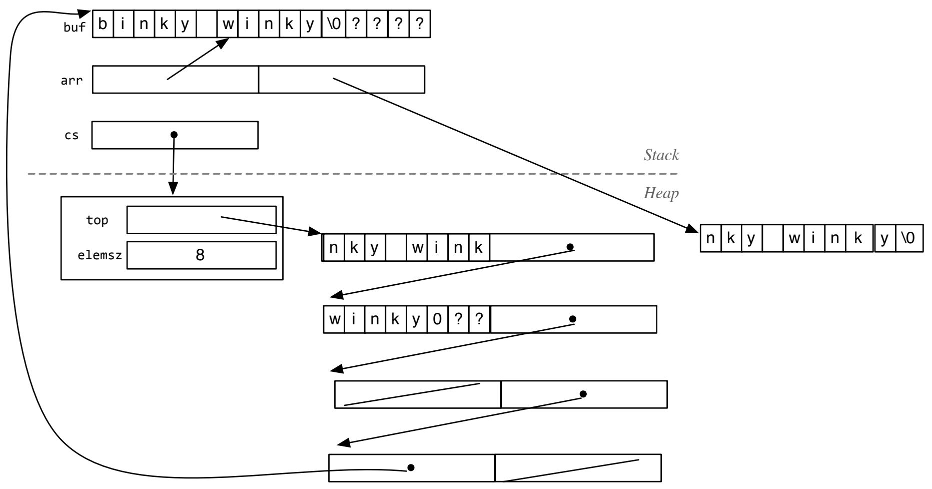
Problem 2:

```
void cstack_push (CStack *cs, const void *addr)
{
    void *blob = malloc(cs->elemsz + sizeof(void *));
    memcpy(blob, addr, cs->elemsz);
    *(void **)((char *)blob + cs->elemsz) = cs->top;   // both casts needed!
    cs->top = blob;
}

void cstack_multipush (CStack *cs, const void *arr, int nelems)
{
    for (int i = 0; i < nelems; i++)
        cstack_push(cs, (char*)arr + i*cs->elemsz);
}
```

Problem 3:

Prints "`nky wink`" followed by zero or more garbage characters. `printf` is reading from a char array that does not contain a null terminator which causes it to read into neighboring bytes and print those, stopping at first zero byte. These extra characters will vary from run to run as their value depends on the address where the next cell is located in the heap.



Problem 4:

G L K