Midterm Practice Problems

These problems will help you study for the midterm. You should not expect the exam to be exactly like this.

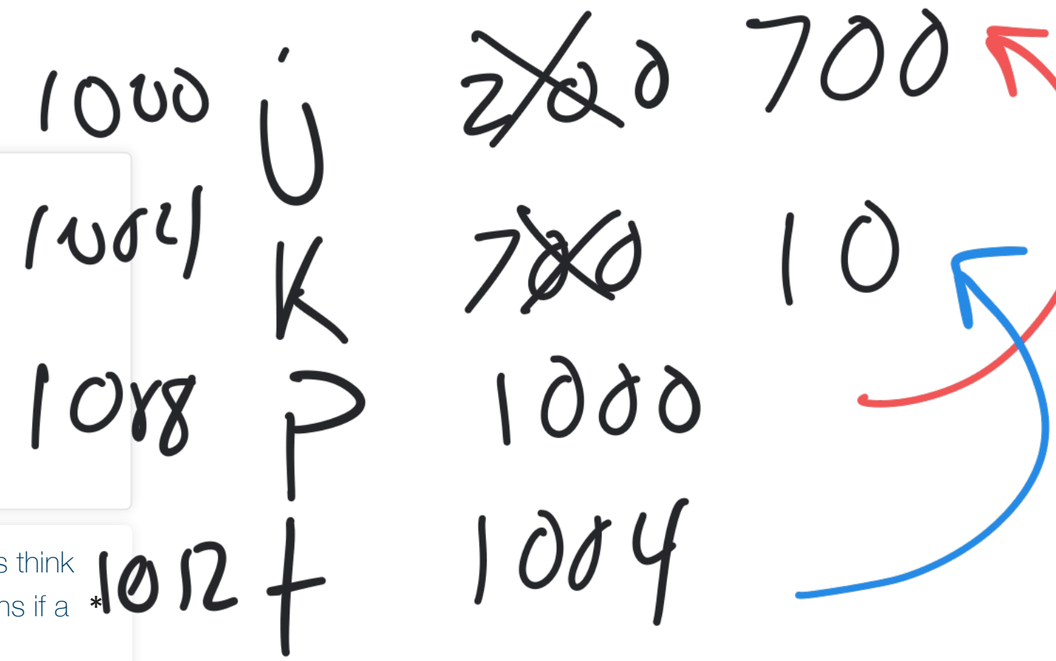
1. Arrays, Pointers, and Strings

1.1: What are `*p, *t, j,` and `k` after this code is run?

```
int j, k;
int *p = &j;
int *t;
*p = 200;
k = *p + 500;
t = &k;
*p = *t;
k = 10;
```

Final Values

Hint: You may find it easiest to write out a memory diagram for questions like this. Always think through the code carefully and you should have a precise understanding of what happens if a `*` or `&` are used.



j = 700
K = 10
*p = 700
*t = 10

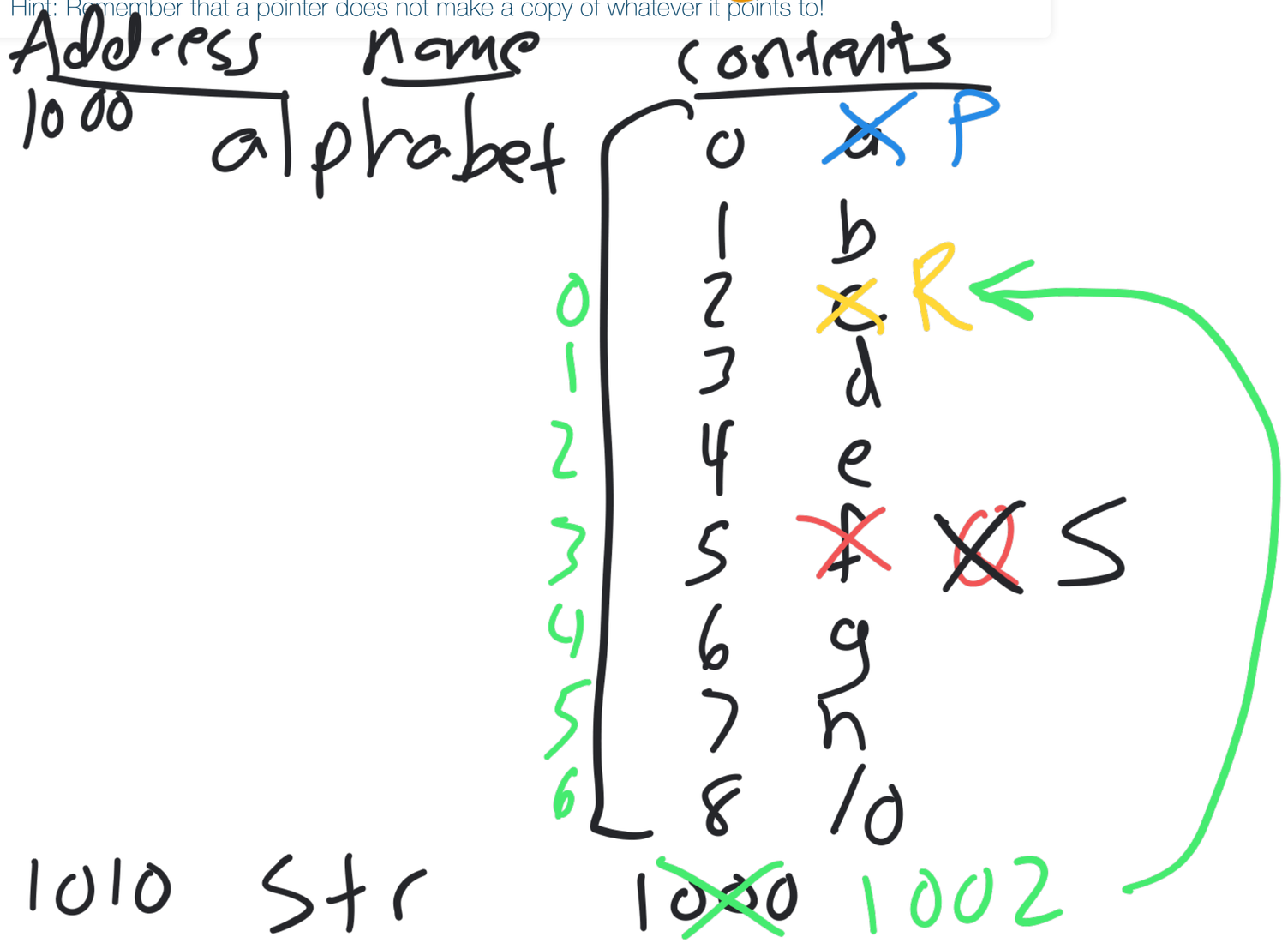
1.2: What will this program print?

```
char alphabet[9] = "abcdefgh";
char *str = alphabet;
str[0] = 'P';
str[5] = 'Q';
str += 2;
*str = 'R';
str[3] = 'S';

printf("alph: %s\n", alphabet);
printf("str : %s\n", str);
```

Pb R d e S gh

R d e S gh

Hint: Remember that a pointer does not make a copy of whatever it points to!

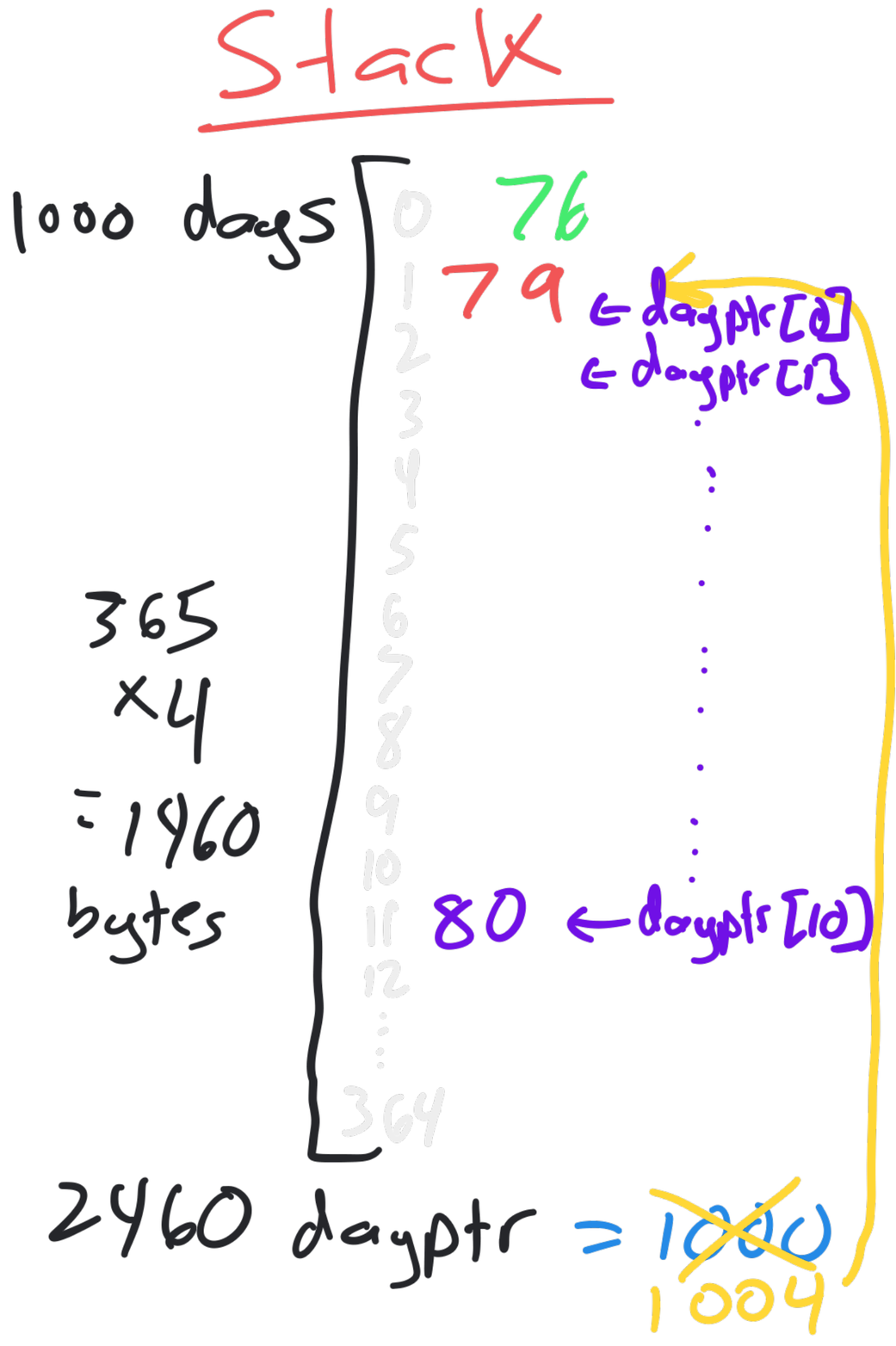


1.3: What entries in the days array does this fill in and to what values?

```
int days[365];
int *dayptr = days;
*dayptr = 76;
dayptr++;
*dayptr = 79;
dayptr[10] = 80;
```

Increases address by size of one int

Hint: We saw pointer arithmetic in Module C-1, but you may be confused about what is happening to a pointer's address when you see a line like `dayptr++`. We would expect this to increase the address of `dayptr` by one, but in reality that would be problematic since each integer in the array consumes several bytes (typically 4), so adding 1 to the address would make it point to the middle of the integer. Therefore, the C compiler automatically translates `dayptr++` into an instruction that increases the address by 4, not 1. Similarly, the line `dayptr = dayptr + 10` would actually increase the address stored in `dayptr` by `40`. *Thus any math operations performed on a pointer are automatically scaled by the size of the pointer's data type (in this case `int`).*



1.4: Write a program that uses a dynamically sized 2D array. The first row should hold the number 1, the second should hold two integers each set to 2, and so on:

```
1
22
333
4444
....
nnnnnn
```

Your array should only use the precise amount of space needed to store the `n` rows of integers. After you have allocated space and filled in all entries in the 2D array you should print it out in the format above.

Write your code in this Repl.it editor. Adjusting `n` should result in a different sized output.

> Hint: Remember, a dynamic array must be allocated from the heap, and in this case we need a different amount of space for each row (1 integer for the first row, 2 integers for the second, etc). Your overall array should be represented by a `int**` variable, and you will need to allocate space for the first dimension of the array (the rows), and then have a for loop to allocate space for the columns in each row.

https://repl.it/@twood02/c-review-2dn-solved

2. Memory

2.1: Fill in a memory diagram for when the following code is run until it reaches the HERE comment. Assume that both an int and an int* consume 4 bytes and that malloc() allocates memory in consecutive chunks out of the heap.

```
void firstFunc() {
  int x = 20;
  int b = 30;
  int *p = malloc(sizeof(int) * 2);
  p[0] = 5;
  p[1] = 6;
}
void secondFunc() {
  int *c;
  c = (int*) malloc(sizeof(int));
  int *d;
  d = (int*) malloc(sizeof(int));
  *c = 45;
  free(d);
  thirdFunc();
  free(c);
}
void thirdFunc() {
  int e = 50;
  int *q;
  q = (int*) malloc(sizeof(int));
  e = 55;
  *q = 100;
  /**** HERE ****/
  free(q);
  return;
}
void fourthFunc() {
  int z = 23;
  int p = 45;
}
int main() {
  int x = 10;
  int *y = &x;
  firstFunc();
  secondFunc();
  fourthFunc();
  return 0;
}
```

2.2: How much memory does the following code reserve, and how many 4-byte ints could it store? How could you change the code to always create exactly enough space for 20 integers, even if you weren't sure if the code would be run on a 16, 32, or 64bit platform?

```
int* intlist = (int*) malloc(10);
```

You don't need to memorize the size of every data type, but you should understand why some are larger than others.

Allocates 10 bytes
Usually an int is 4 bytes
So this holds 2.5 ints → not very useful!
The coder probably meant to write:

```
int* intlist = (int*) malloc(sizeof(int)*10);
```

2.3: Consider the code in this Repl.it editor:

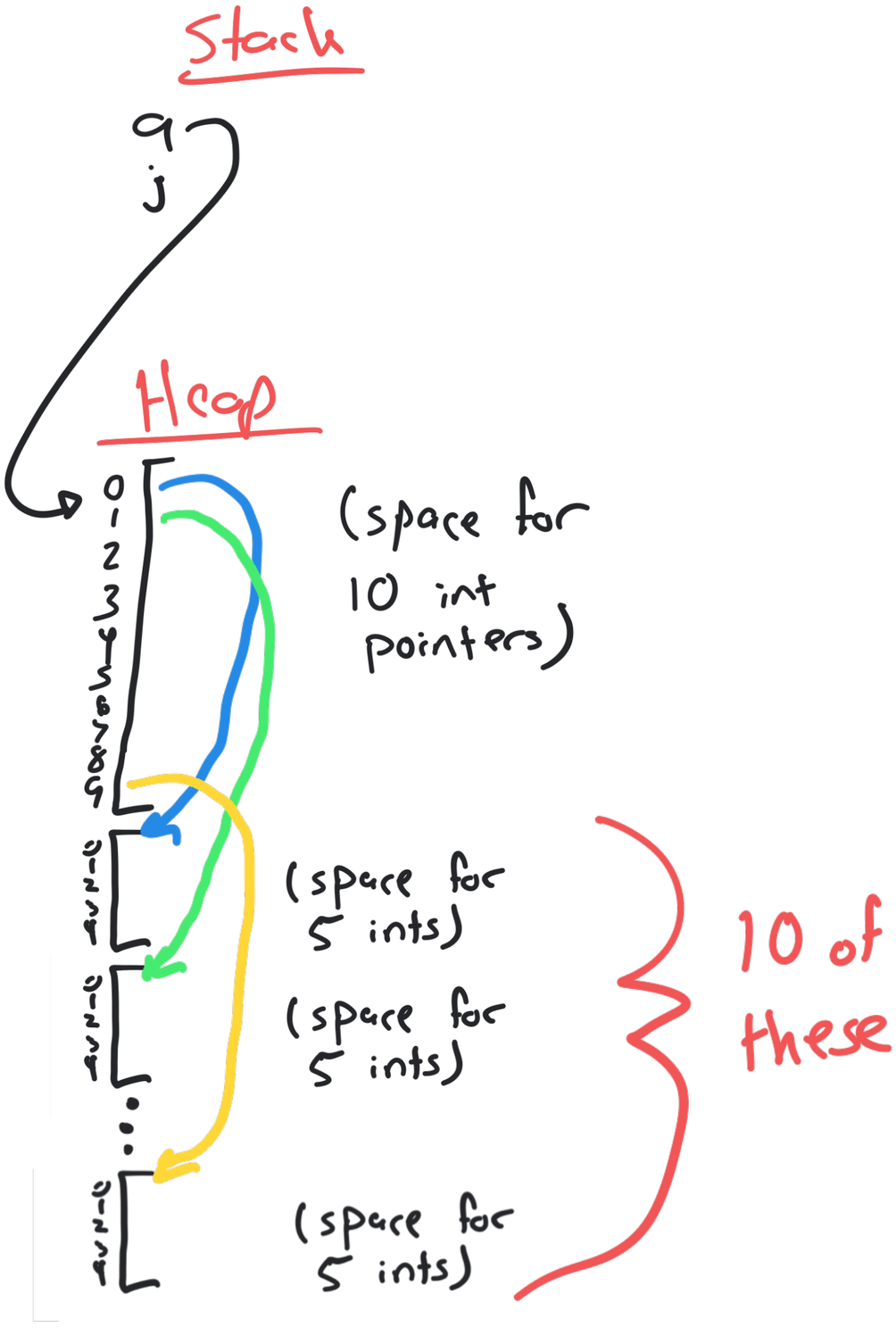
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int** a;
  a = malloc(10*sizeof(int*));
  for(int j=0; j < 10; j++) {
    a[j] = malloc(sizeof(int)*5);
    for(int k=0; k < 5; k++) {
      a[j][k] = j * k;
      printf("a[%d][%d] = %d\n", j, k, a[j][k]);
    }
  }
  free(a);
  return 0;
}
```

Never freed!

Exactly how much memory does this program allocate on the heap? Does the program correctly free all of the memory it allocates?

> Always look for a `free` that matches every `malloc`

Allocates 10 int* and 50 int
But only frees the array
of 10 int pointers.



2.4 The size of a `char` is 1 byte. Use a Repl.it editor to find out the size of a `char*`. Why are they different?

> You don't need to memorize the size of every data type, but you should understand why some are larger than others.

- A char only needs to store a letter, so one byte is sufficient (1 byte = 8 bits = 2^8 values)

- A char* needs to store an address, which could be a really big number! A 64 bit CPU needs 64 bits = 8 bytes to store an address. A 32 bit CPU needs 32 bits = 4 bytes for an address.

3. Data Structures

3.1: Consider the code below from this Repl.it editor.

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

struct car {
  char* make;
  char* model;
  int year;
};

struct house {
  char* street;
  int number;
  int sqft;
};

struct person {
  struct car *cars;
  int numCars;
  struct house house;
  char* name;
};

void printCar(struct car* car) {
  printf("Car: %s %s from %d\n", car->make, car->model, car->year);
}

void printHouse(struct house* house) {
  printf("House: %d %s with %d square feet\n", house->number, house->street, ho
}

void printPerson(struct person* person) {
  // print all cars and house info
}

int main(void) {
  struct person me;
  // fill in data structure

  printPerson(&me);
  return 0;
}
```

The program above includes struct definitions for an accounting program. For each person, it can track multiple cars that they own and a single house. The struct definitions and print functions for a car and house have been provided. You must enhance the `main()` function so that it fills in information for a person, including a house and two cars. Then you must complete the `printPerson()` function so that it displays an output similar to:

```
Prof. X owns:
Car: Honda Fit from 2014
Car: Ford Fiesta from 2017
House: 315 West St with 2459 square feet
```

https://repl.it/@twood02/c-review-structs-solved