CSc 3320: Systems Programming

Spring 2021

Midterm 2: Total points = 100

Assigned: 11th Apr 2021, Sunday 11:59 PM **Submission Deadline: 18th Apr 2021, Sunday, 11.59 PM (No extensions. If your submission is not received by this time then it will NOT be accepted.)**

Submission instructions:

1. Create a Google doc for your submission.
2. Start your responses from page 2 of the document and copy these instructions on page 1.
3. Fill in your name, campus ID and panther # in the fields provided. If this information is missing TWO POINTS WILL BE DEDUCTED.
4. Keep this page 1 intact. If this *submissions instructions* page is missing in your submission TWO POINTS WILL BE DEDUCTED.
5. Start your responses to each QUESTION on a new page.
6. If you are being asked to write code copy the code into a separate txt file and submit that as well. The code should be executable. E.g. if asked for a C script then provide myfile.c so that we can execute that script. In your answer to the specific question, provide the steps on how to execute your file (like a ReadMe).
7. If you are being asked to test code or run specific commands or scripts, provide the evidence of your outputs through a screenshot and/or screen video-recordings and copy the same into the document.
8. Upon completion, download a .PDF version of the google doc document and submit the same along with all the supplementary files (videos, pictures, scripts etc).

Full Name: Michael Anderson

Campus ID: manderson113

Panther #: 002485269

Question 1

```
#include <stdio.h>

// simple enum for the sorting order
typedef enum sorting
{
	ascending = 1,
	descending = 2
} sorting;

// returns the sorting order derived from
sorting getOrder(char*);

// sorts the array
// (pointer to array, size of array, desired order)
void sort_numeric(double*, size_t, sorting);

// sorts the array using quicksort
void quicksort(double*, double*, double*, sorting);

// returns the location of the pivot
// all elements to the left are lower than ray[pivot]
// all elements to the right are higher than ray[pivot]
double* partition(double*, double*, double*, sorting);

// swaps two elements in an array of doubles
void swap(double*, double*);

// prints the given array
void print_array(double*, size_t);

void main(int argc, char *argv[])
{
	double evalArray[] = {
		10,
		0.25,
		-2342,
		12123,
		3.145435,
		6,
		6,
		5.999,
		-2,
		-5,
		-109.56
	};
```

```
	double *evalArrayPointer = evalArray;
	size_t size = (sizeof(evalArray))/(sizeof(evalArray[0]));

	int inputProvided = 0;
	sorting order = 0;

	// attempts to read the desired sorting order from the command line
arguments
	if (argc > 1) {
		order = getOrder(argv[1]);
	}

	// if the desired order couldn't be discerned, ask user
	while (!order)
	{
		char input[1];
		printf("(A)scending or (D)escending? ");
		scanf("%s",  input);
		order = getOrder(input);
	}

	printf("\nUnsorted array:\n");
	print_array(evalArrayPointer, size);

	if (order == ascending) {
		printf("Ascending:\n");
		sort_numeric(evalArrayPointer, size, ascending);
	} else {
		printf("Descending:\n");
		sort_numeric(evalArrayPointer, size, descending);
	}

	print_array(evalArrayPointer, size);
}

sorting getOrder(char* str)
{
	sorting order = 0;
	size_t i = 0;
	// skip past '-' in the cases of '-a' or '-d'
	if (str[0] == '-') {
		i++;
	}
	if (str[i] == 'a' || str[i] == 'A') {
		order = ascending;
	} else if (str[i] == 'd' || str[i] == 'D') {
		order = descending;
	}
	return order;
```

```
}

void sort_numeric(double* ray, size_t size, sorting order)
{
	// pass the array, the location of the first element,
	// the location of the last element, and the desired order
	quicksort(ray, ray, ray+size-1, order);
}

void quicksort(double* ray, double* low, double* high, sorting order)
{
	if (low >= high) { return; } // nothing to sort here

	// after this, the pivot will be in the right place,
	// with everything else being in the correct 'half'
	double* pivot = partition(ray, low, high, order);
	// sort each 'half'
	quicksort(ray, low, pivot-1, order);
	quicksort(ray, pivot+1, high, order);
}

double* partition(double* ray, double* low, double* high, sorting order)
{
	// use the last element in this section as the pivot
	double pivotValue = *high;
	double* pivotIndex = low;
	double* index;
	// work from the left,
	// making sure everything to the left of the pivotIndex should be there
	// swaps if it needs to
	for (index = low; index <= high; index++)
	{
		if ((order == ascending && *index < pivotValue) || (order ==
descending && *index > pivotValue))
		{
			swap(index, pivotIndex);
			pivotIndex++;
		}
	}
	swap(pivotIndex, high);
	return pivotIndex;
}

void swap(double* a, double* b)
{
	double temp = *a;
	*a = *b;
	*b = temp;
}
```

```
void print_array(double* ray, size_t size)
{
	if (size <= 0) { return; }
	double* p;
	for (p = ray; p<&ray[size]; p++)
	{
		printf("%f\n", *p);
	}
	printf("\n");
}
```

```
[manderson113@gsuad.gsu.edu@snowball two]$ ./sortNumbers
(A)scending or (D)escending? a

Unsorted array:
10.000000
0.250000
-2342.000000
12123.000000
3.145435
6.000000
6.000000
5.999000
-2.000000
-5.000000
-109.560000

Ascending:
-2342.000000
-109.560000
-5.000000
-2.000000
0.250000
3.145435
5.999000
6.000000
6.000000
10.000000
12123.000000
```

```
[manderson113@gsuad.gsu.edu@snowball two]$ ./sortNumbers
(A)scending or (D)escending? d

Unsorted array:
10.000000
0.250000
-2342.000000
12123.000000
3.145435
6.000000
6.000000
5.999000
-2.000000
-5.000000
-109.560000

Descending:
12123.000000
10.000000
6.000000
6.000000
5.999000
3.145435
0.250000
-2.000000
-5.000000
-109.560000
-2342.000000
```

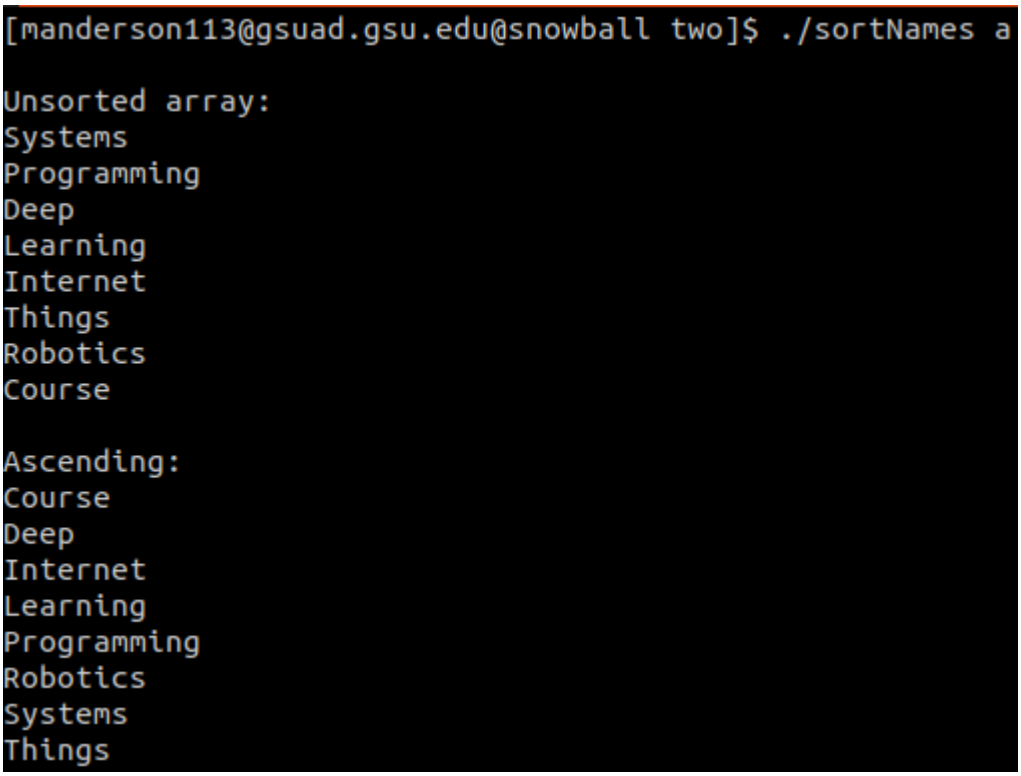
Question 2

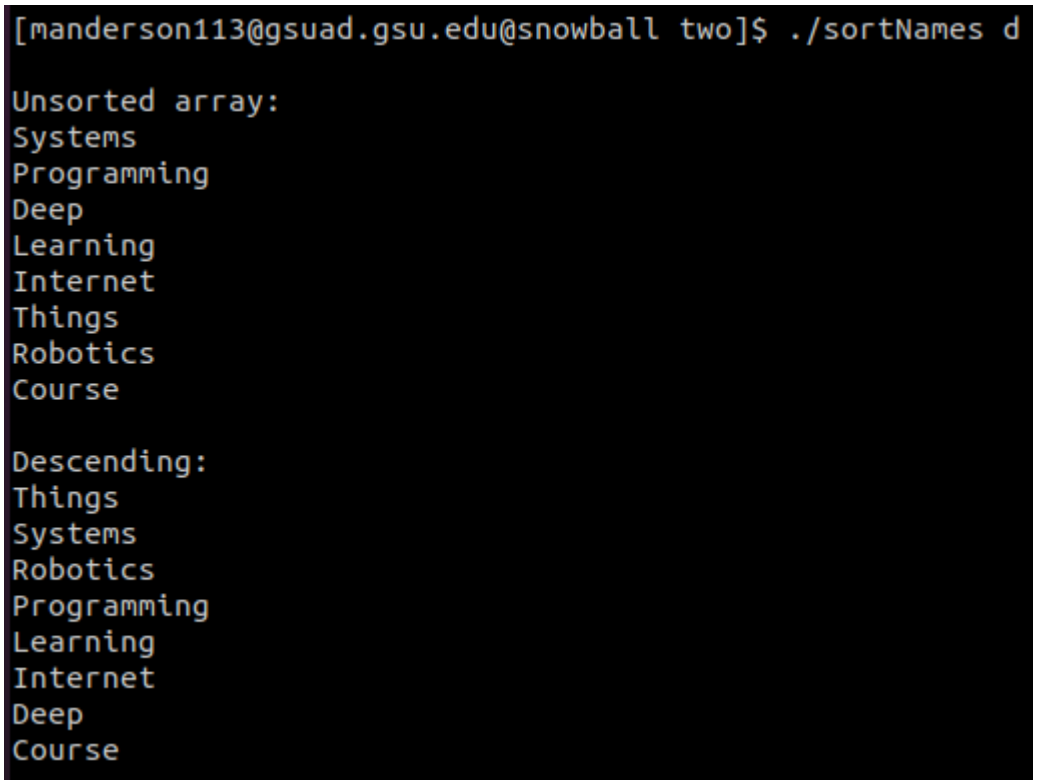
```
#include <stdio.h>

// simple enum for the sorting order
typedef enum sorting
{
	ascending = 1,
	descending = 2
} sorting;

// returns the sorting order derived from
sorting getOrder(char*);

// sorts the array
// (pointer to array, size of array, desired order)
void sort_alphabetic(char**, size_t, sorting);

// sorts the array using quicksort
void quicksort(char**, char**, char**, sorting);

// returns the location of the pivot
// all elements to the left are lower than ray[pivot]
// all elements to the right are higher than ray[pivot]
char** partition(char**, char**, char**, sorting);

// swaps two strings
void swap(char**, char**);

// prints the given array
void print_array(char**, size_t);

// the +1 is for the null character '\0'
#define MAX_NAME_SIZE 20 + 1

void main(int argc, char *argv[])
{
	char* evalArray[] = {
		"Systems",
		"Programming",
		"Deep",
		"Learning",
		"Internet",
		"Things",
		"Robotics",
		"Course"
	};
```

```
	char** evalArrayPointer = evalArray;
	size_t size = (sizeof(evalArray))/(sizeof(evalArray[0]));

	int inputProvided = 0;
	sorting order = 0;

	// attempts to read the desired sorting order from the command line
arguments
	if (argc > 1) {
		order = getOrder(argv[1]);
	}

	// if the desired order couldn't be discerned, ask user
	while (!order)
	{
		char input[1];
		printf("(A)scending or (D)escending? ");
		scanf("%s",  input);
		order = getOrder(input);
	}

	printf("\nUnsorted array:\n");
	print_array(evalArrayPointer, size);

	sort_alphabetic(evalArrayPointer, size, order);

	if (order == ascending) {
		printf("Ascending:\n");
	} else {
		printf("Descending:\n");
	}

	print_array(evalArrayPointer, size);
}

sorting getOrder(char* str)
{
	sorting order = 0;
	size_t i = 0;
	// skip past '-' in the cases of '-a' or '-d'
	if (str[0] == '-') {
		i++;
	}
	if (str[i] == 'a' || str[i] == 'A') {
		order = ascending;
	} else if (str[i] == 'd' || str[i] == 'D') {
		order = descending;
	}
	return order;
```

```
}

void sort_alphabetic(char** ray, size_t size, sorting order)
{
	// pass the array, the location of the first element,
	// the location of the last element, and the desired order
	quicksort(ray, ray, ray+size-1, order);
}

void quicksort(char** ray, char** low, char** high, sorting order)
{
	if (low >= high) { return; } // nothing to sort here

	// after this, the pivot will be in the right place,
	// with everything else being in the correct 'half'
	char** pivot = partition(ray, low, high, order);

	// sort each 'half'
	quicksort(ray, low, pivot-1, order);
	quicksort(ray, pivot+1, high, order);
}

char** partition(char** ray, char** low, char** high, sorting order)
{
	// use the last element in this section as the pivot
	char* pivotValue = *high;
	char** pivotIndex = low;
	char** index;
	// work from the left,
	// making sure everything to the left of the pivotIndex should be there
	// swaps if it needs to
	for (index = low; index <= high; index++)
	{
		int cmp = strcasecmp(*index, *high);
		if ((order == ascending && cmp<0) || (order == descending &&
cmp>0))
		{
			swap(index, pivotIndex);
			pivotIndex++;
		}
	}
	swap(pivotIndex, high);
	return pivotIndex;
}

void swap(char** a, char** b)
{
	char* temp = *a;
	*a = *b;
```

```
	*b = temp;
}

void print_array(char** ray, size_t size)
{
	if (size <= 0) { return; }
	char **p;
	for (p = &ray[0]; p < &ray[size]; p++)
	{
		printf("%s\n", *p);
	}
	printf("\n");
}
```


[manderson113@gsuad.gsu.edu@snowball two]$ ./sortNames a
Unsorted array:
Systems
Programming
Deep
Learning
Internet
Things
Robotics
Course
Ascending:
Course
Deep
Internet
Learning
Programming
Robotics
Systems
Things


[manderson113@gsuad.gsu.edu@snowball two]$ ./sortNames d
Unsorted array:
Systems
Programming
Deep
Learning
Internet
Things
Robotics
Course
Descending:
Things
Systems
Robotics
Programming
Learning
Internet
Deep
Course

Question 3

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

// simple enum for the sorting order
typedef enum sorting
{
	ascending = 1,
	descending = 2
} sorting;

// returns the sorting order derived from
sorting getOrder(char*);

// sorts the array
// (pointer to array, size of array, desired order)
void sort_numeric(double*, size_t, sorting);

// sorts the array using quicksort
void quicksort(double*, double*, double*, sorting);

// returns the location of the pivot
// all elements to the left are lower than ray[pivot]
// all elements to the right are higher than ray[pivot]
double* partition(double*, double*, double*, sorting);

// swaps two elements in an array of doubles
void swap(double*, double*);

// prints the given array
void print_array(double*, size_t);

void main(int argc, char *argv[])
{
	const size_t INITIAL_COUNT = 10;
	const double GROWTH_FACTOR = 1.5;

	double *ray;
	size_t max_size = INITIAL_COUNT;
	ray = malloc(max_size * sizeof(double));
	size_t size = 0;

	printf("Please enter any amount of doubles, 0 to quit:\n");

	size_t index = 0;
	double val = 0;
	do
```

```
	{
		scanf("%lf",&val);
		if (val != 0)
		{
			ray[index] = val;
			index++;
			size++;
			if (size >= max_size)
			{
				max_size *= GROWTH_FACTOR;
				double *newray = (double*)realloc(ray, max_size *
sizeof(double));
				if (newray == NULL)
				{
					printf("Error allocating memory\n");
					return;
				} else {
					ray = newray;
				}
			}
		}
	} while (val != 0);

	int inputProvided = 0;
	sorting order = 0;

	// attempts to read the desired sorting order from the command line
arguments
	if (argc > 1) {
		order = getOrder(argv[1]);
	}

	// if the desired order couldn't be discerned, ask user
	while (!order)
	{
		char input[1];
		printf("(A)scending or (D)escending? ");
		scanf("%s",  input);
		order = getOrder(input);
	}

	//printf("\nUnsorted array:\n");
	//print_array(ray, size);

	if (order == ascending) {
		printf("Ascending:\n");
		sort_numeric(ray, size, ascending);
	} else {
		printf("Descending:\n");
```

```
            sort_numeric(ray, size, descending);
      }

      print_array(ray, size);
      free(ray);
}

sorting getOrder(char* str)
{
      sorting order = 0;
      size_t i = 0;
      // skip past '-' in the cases of '-a' or '-d'
      if (str[0] == '-') {
            i++;
      }
      if (str[i] == 'a' || str[i] == 'A') {
            order = ascending;
      } else if (str[i] == 'd' || str[i] == 'D') {
            order = descending;
      }
      return order;
}

void sort_numeric(double* ray, size_t size, sorting order)
{
      // pass the array, the location of the first element,
      // the location of the last element, and the desired order
      quicksort(ray, ray, ray+size-1, order);
}

void quicksort(double* ray, double* low, double* high, sorting order)
{
      if (low >= high) { return; } // nothing to sort here

      // after this, the pivot will be in the right place,
      // with everything else being in the correct 'half'
      double* pivot = partition(ray, low, high, order);
      // sort each 'half'
      quicksort(ray, low, pivot-1, order);
      quicksort(ray, pivot+1, high, order);
}

double* partition(double* ray, double* low, double* high, sorting order)
{
      // use the last element in this section as the pivot
      double pivotValue = *high;
      double* pivotIndex = low;
      double* index;
      // work from the left,
```

```
	// making sure everything to the left of the pivotIndex should be there
	// swaps if it needs to
	for (index = low; index <= high; index++)
	{
		if ((order == ascending && *index < pivotValue) || (order ==
descending && *index > pivotValue))
		{
			swap(index, pivotIndex);
			pivotIndex++;
		}
	}
	swap(pivotIndex, high);
	return pivotIndex;
}

void swap(double* a, double* b)
{
	double temp = *a;
	*a = *b;
	*b = temp;
}

void print_array(double* ray, size_t size)
{
	if (size <= 0) { return; }
	double* p;
	for (p = ray; p<&ray[size]; p++)
	{
		printf("%f\n", *p);
	}
	printf("\n");
}
```

```
[manderson113@gsuad.gsu.edu@snowball two]$ ./dynamicSortNumbers
Please enter any amount of doubles, 0 to quit:
1 2 3 2 1 4 5 1 65  2 1 2 3 12 3 32.12  2 90
0
(A)scending or (D)escending? a
Ascending:
1.000000
1.000000
1.000000
1.000000
2.000000
2.000000
2.000000
2.000000
2.000000
3.000000
3.000000
3.000000
4.000000
5.000000
12.000000
32.120000
65.000000
90.000000
```

```
[manderson113@gsuad.gsu.edu@snowball two]$ ./dynamicSortNumbers d
Please enter any amount of doubles, 0 to quit:
-1 -213 -3.21 3.1415 2.78 17 12.2 12 9128 12 20 123 939 1 .211
0
Descending:
9128.000000
939.000000
123.000000
20.000000
17.000000
12.200000
12.000000
12.000000
3.141500
2.780000
1.000000
0.211000
-1.000000
-3.210000
-213.000000
```

Question 4

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

enum sexes {
      male,
      female,
      other
};

const char *sex_names[3] = {
      "Male",
      "Female",
      "Does not wish to identify"
};

enum dose_number {
      one = 1,
      two = 2
};

enum vaccine_type {
      Pfizer,
      Moderna,
      Johnson
};

const char *vaccine_names[3] = {
      "Pfizer",
      "Moderna",
      "Johnson & Johnson"
};

// accurate count of vaccine options
const unsigned int vaccine_count =
sizeof(vaccine_names)/sizeof(vaccine_names[0]);

typedef enum selection_options {
      none = 0, // no selection made
      get = 1,  // retrieving a record
      put = 2,  // creating a record
      quit = 3  // exit the program
} selection;

// contains all the information stored for the form
```

```
typedef struct form {
      char firstName[50], lastName[50], zipcode[13], code[9];
      struct tm dateOfBirth, previousDose;
      enum sexes sex;
      enum dose_number dose;
      enum vaccine_type vaccine;
} Form;

// this struct stores all the forms and meta information for them
typedef struct registry {
      Form *forms;         // pointer to all the forms
      size_t max;          // the current maximum count
      size_t count;        // number of forms stored
      double growthFactor; // how much to expand the memory by when it fills
up
} Registry;

// creates the form, and ensures there is enough
// memory allocated for all the forms
Form* create_form(Registry*);

// prompts the user for the information to fill
// out the form and returns it
Form* register_form(Form*);

// prints a form with formatting
void print_form(Form*);

// generates the code for the given form and stores it
// in the form
Form* generate_code(Form*);

// searches the registry for a form with a given code
Form* retrieve(char*, Registry*);

// initializes the registry, which is the struct that holds
void setup(Registry*);

// prompts the user for if they want to
// make a new record, search for a record, or quit
selection getSelection();

// prompts the user for a code to search the registry
void find_form(Registry *vr);

// cleans up before quitting
void end(Registry *vr);

// fills the time struct with 0's
```

```
void zero_time(struct tm*);

void main(int argc, char* argv[])
{
	Registry vr;
	setup(&vr);
	while (1) // continuous loop
	{
		selection choice = getSelection();
		if (choice == put) { // create new entry
			// creates and fills the form
			Form *newForm = create_form(&vr);
			// adds the form to the registry
			register_form(newForm);
		} else if (choice == get) { // find entry
			find_form(&vr);
		} else {
			end(&vr); // cleans up
			return; // ends the program
		}
	}
}

// initial setup for the registry to hold the forms
void setup(Registry *vr)
{
	const size_t INITIAL_COUNT = 5;
	const double GROWTH_FACTOR = 1.5; // will increase by 1.5x times when
full

	vr->max = INITIAL_COUNT;
	vr->growthFactor = GROWTH_FACTOR;
	vr->forms = (Form*)malloc(vr->max * sizeof(struct form));
	vr->count = 0;
}

// prompts the user for what action they'd like to preform
selection getSelection()
{
	selection choice = none;
	while (choice == none)
	{
		printf("(N)ew, (R)etrieve, (Q)uit\n");
		char input[20];
		scanf("%s",input);
		char c = input[0];
		if ((c == 'n') || (c == 'N')) {
			choice = put;
		} else if ((c == 'r') || (c == 'R')) {
```

```
                choice = get;
            } else if ((c == 'q') || (c == 'Q')) {
                choice = quit;
            }
      }
      printf("\n");
      return choice;
}

// prompts the user for the code, prints the matching form
void find_form(Registry *vr)
{
      printf("Enter your vaccine code: \n");
      char input[20];
      scanf("%s",input);
      Form *f = retrieve(input, vr);
      printf("\n");
      if (f == NULL) {
            // not found
            printf("Record not found.\n\n");
      } else {
            print_form(f);
      }
}

// final cleanup
void end(Registry *vr)
{
      // frees the allocated space for the forms
      free(vr->forms);
}


Form* create_form(Registry* vr)
{
      if (vr->count >= vr->max)
      {
            // need to expand allocated space
            vr->max *= vr->growthFactor;
            vr->forms = (Form*)realloc(vr->forms, vr->max * sizeof(struct
form));
      }
      // get the index for the next form location
      Form *newForm = &(vr->forms[vr->count]);
      vr->count++;
      return newForm;
}

Form* register_form(Form* f)
```

```
{
	// zero out all the members of the time structs
	zero_time(&f->dateOfBirth);
	zero_time(&f->previousDose);

	printf("First name: ");
	scanf("%s", f->firstName);

	printf("Last name: ");
	scanf("%s", f->lastName);

	printf("Date of birth (mm/dd/yyyy): ");
	scanf("%d/%d/%d", &(f->dateOfBirth.tm_mon), &(f->dateOfBirth.tm_mday),
&(f->dateOfBirth.tm_year));
	// months are from 0-11
	f->dateOfBirth.tm_mon--;
	// years are from 1900
	f->dateOfBirth.tm_year -= 1900;

	printf("(M)ale, (F)emale, (X) Do not wish to identify: ");
	char sex[20];
	scanf("%s", sex);
	char c = sex[0];
	if ((c == 'M') || (c == 'm')) {
		f->sex = male;
	} else if ((c == 'F') || (c == 'f')) {
		f->sex = female;
	} else {
		f->sex = other;
	}

	printf("Dose number: ");
	unsigned int dose;
	scanf("%d", &dose);
	if (dose == 2) {
		f->dose = two;
	} else {
		f->dose = one;
	}

	// only if the user had a previous dose
	if (f->dose > 1) {
		printf("Date of last dose (mm/dd/yyyy): ");
		scanf("%d/%d/%d", &(f->previousDose.tm_mon),
&(f->previousDose.tm_mday), &(f->previousDose.tm_year));
		// months are from 0-11
		f->previousDose.tm_mon--;
		// years are from 1900
		f->previousDose.tm_year -= 1900;
```

```
	}

	// print available vaccines
	unsigned int i;
	for (i=0; i<vaccine_count; i++) {
		printf("%d: %s\n", i, vaccine_names[i]);
	}
	printf("Type of vaccine: ");
	unsigned int vaccine_choice = -1; // becomes largest number for
unsigned ints
	while (vaccine_choice >= vaccine_count) // vaccine_choice cannot be
negative
	{
		scanf("%d", &vaccine_choice);
	}
	f->vaccine = vaccine_choice;

	printf("Zip Code: ");
	scanf("%s", f->zipcode);

	printf("\n");
	generate_code(f);
	return f;
}

// returns the form with the matching code, NULL otherwise
Form* retrieve(char* code, Registry *vr)
{
	int i;
	for (i=0; i < vr->count; i++) {
		Form *f = &(vr->forms[i]);
		char *formCode = f->code;
		if (strncmp(code, formCode, 8) == 0) {
			return f;
		}
	}
	return NULL;
}

// prints all information stored in the form
void print_form(Form *f) {
	printf("Name: %s %s\n", f->firstName, f->lastName);

	printf("Date of birth: %d/%d/%d\n",
		f->dateOfBirth.tm_mon+1, f->dateOfBirth.tm_mday,
f->dateOfBirth.tm_year + 1900
	);

	printf("Sex: %s\n", sex_names[f->sex]);
```

```
	printf("Dose number: %d\n", f->dose);

	if (f->dose > 1) {
		// ony print if there was a previous dose
		printf("Date of last dose: %d/%d/%d\n",
			f->previousDose.tm_mon+1, f->previousDose.tm_mday,
f->previousDose.tm_year + 1900
		);
	}

	printf("Type of vaccine: %s\n", vaccine_names[f->vaccine]);

	printf("Zip Code: %s\n", f->zipcode);

	printf("\n");
}

Form* generate_code(Form* f)
{
	f->code[0] = f->firstName[0]; // first letter of first name

	f->code[1] = f->lastName[0]; // first letter of last name

	// time from 1900 to birth day in seconds
	time_t birthTime = mktime( &f->dateOfBirth );
	time_t currentTime;
	time(&currentTime); // time from 1900 to now in seconds
	//  finds the user's current age                  s    min  hr   yr
fix rounding issue
	int years = ((difftime(currentTime, birthTime) / (60 * 60 * 24 *
365.25)) + 1e-9);
	// age as two digits
	if (years <= 0) { // zeroes if less or equal to zero
		f->code[2] = '0';
		f->code[3] = '0';
	} else if (years >= 99) { // caps at 99
		f->code[2] = '9';
		f->code[3] = '9';
	} else { // 0 < age < 99
		char age[3];
		sprintf(age, "%d", years); // age = (string)years
		if (years < 10) {
			f->code[2] = '0';
			f->code[3] = age[0];
		} else {
			f->code[2] = age[0];
			f->code[3] = age[1];
		}
```

```
	}

	f->code[4] = vaccine_names[f->vaccine][0]; // first letter of vaccine

	int zipSize = strlen(f->zipcode);
	int i;
	int index = 5; // starts at the 5th index of the code, 6-8th characters
	for (i=zipSize-3; i<zipSize; i++)
	{
		// gets last three characters of zip code
		if (i<0) {
			// only relevant if zip code is less than three characters
			// this shouldn't happen, but ensures the program won't
break
			continue;
		}
		f->code[index] = f->zipcode[i];
		index++;
	}

	f->code[index] = '\0'; // null character, end string

	printf("Your unique code: %s\n\n", f->code);
	return f;
}

// zeroes out the time struct, had issues with it otherwise
void zero_time(struct tm *time)
{
	time->tm_sec= 0;
	time->tm_min = 0;
	time->tm_hour = 0;
	time->tm_mday = 0;
	time->tm_mon = 0;
	time->tm_year = 0;
	time->tm_wday = 0;
	time->tm_yday = 0;
	time->tm_isdst = 0;
}
```

```
[manderson113@gsuad.gsu.edu@snowball two]$ ./vaccine
(N)ew, (R)etrieve, (Q)uit
n

First name: Brenda
Last name: Ackley
Date of birth (mm/dd/yyyy): 8/7/1998
(M)ale, (F)emale, (X) Do not wish to identify: f
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 1
Zip Code: 92562

Your unique code: BA22M562

(N)ew, (R)etrieve, (Q)uit
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Johnny
Last name: Alsup
Date of birth (mm/dd/yyyy): 6/5/1952
(M)ale, (F)emale, (X) Do not wish to identify: m
Dose number: 2
Date of last dose (mm/dd/yyyy): 3/15/2021
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 2
Zip Code: 24309

Your unique code: JA68J309
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Arie
Last name: Andres
Date of birth (mm/dd/yyyy): 7/4/1979
(M)ale, (F)emale, (X) Do not wish to identify: x
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 0
Zip Code: 40574

Your unique code: AA41P574
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Carla
Last name: Harris
Date of birth (mm/dd/yyyy): 7/18/1991
(M)ale, (F)emale, (X) Do not wish to identify: f
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 2
Zip Code: 62624

Your unique code: CH29J624
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Mary
Last name: Hiatt
Date of birth (mm/dd/yyyy): 8/5/1988
(M)ale, (F)emale, (X) Do not wish to identify: f
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 1
Zip Code: 19032

Your unique code: MH32M032
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Kelli
Last name: Holden
Date of birth (mm/dd/yyyy): 12/7/1983
(M)ale, (F)emale, (X) Do not wish to identify: x
Dose number: 2
Date of last dose (mm/dd/yyyy): 4/1/2021
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 1
Zip Code: 31153

Your unique code: KH37M153
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Oscar
Last name: Jack
Date of birth (mm/dd/yyyy): 9/17/1950
(M)ale, (F)emale, (X) Do not wish to identify: m
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 0
Zip Code: 78843

Your unique code: OJ70P843
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Joey
Last name: Johnson
Date of birth (mm/dd/yyyy): 3/5/1973
(M)ale, (F)emale, (X) Do not wish to identify: m
Dose number: 2
Date of last dose (mm/dd/yyyy): 1/1/2021
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 2
Zip Code: 74358

Your unique code: JJ48J358
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Roy
Last name: Lovett
Date of birth (mm/dd/yyyy): 8/25/1952
(M)ale, (F)emale, (X) Do not wish to identify: m
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 1
Zip Code: 40706

Your unique code: RL68M706
```

```
First name: Luis
Last name: Matsuda
Date of birth (mm/dd/yyyy): 8/28/1973
(M)ale, (F)emale, (X) Do not wish to identify: m
Dose number: 1
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 0
Zip Code: 36140

Your unique code: LM47P140
```

```
(N)ew, (R)etrieve, (Q)uit
n

First name: Elizabeth
Last name: Merritt
Date of birth (mm/dd/yyyy): 8/4/1984
(M)ale, (F)emale, (X) Do not wish to identify: f
Dose number: 2
Date of last dose (mm/dd/yyyy): 2/13/2021
0: Pfizer
1: Moderna
2: Johnson & Johnson
Type of vaccine: 2
Zip Code: 88756

Your unique code: EM36J756
```

```
(N)ew, (R)etrieve, (Q)uit
r

Enter your vaccine code:
CH29J624

Name: Carla Harris
Date of birth: 7/18/1991
Sex: Female
Dose number: 1
Type of vaccine: Johnson & Johnson
Zip Code: 62624

(N)ew, (R)etrieve, (Q)uit
r

Enter your vaccine code:
BA22M562

Name: Brenda Ackley
Date of birth: 8/7/1998
Sex: Female
Dose number: 1
Type of vaccine: Moderna
Zip Code: 92562

(N)ew, (R)etrieve, (Q)uit
r

Enter your vaccine code:
AA41P574

Name: Arie Andres
Date of birth: 7/4/1979
Sex: Does not wish to identify
Dose number: 1
Type of vaccine: Pfizer
Zip Code: 40574

(N)ew, (R)etrieve, (Q)uit
q

[manderson113@gsuad.gsu.edu@snowball two]$
```