Lab 6: Loops and Arrays...

Due date: Monday, November 2, 11:00am.

Lab Assignment

Assignment Preparation

Lab type. This is a **pair programming lab**. The pairs assignments will be made by the instructor and given out on Monday, October 26.

Purpose. The lab allows you to practice the use of loops and arrays.

Programming Style. All submitted C programs must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties. Also note the the Lab 2 requirement for the content of the header comment in each file you submit applies **to each assignment** (lab, programming assignment, homework) in this course.

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

Program Outputs must co-incide. Any deviation in the output is subject to penalties. **PLEASE, USE BINARY EXECUTABLES PROVIDED BY THE INSTRUCTOR!**. The exception is made in case of floating point computations leading to differences in the last few decimal digits. You can check whether or not a program produces correct output by running the `diff` command.

Please, make sure you test all your programs prior to submission! Feel free to test your programs on test cases you have invented on your own.

The Task

You will write a few programs drawing PPM images using two-dimensional arrays. You will also write a few programs extending your prior assignments.

Next lections program: elections.c

The final version of the elections program will compute the election results using loops and arrays.

Input. The input to the program shall be the same as in Lab 3: a sequence of 52 numbers. All numbers except for the 29th input are either 0 or 1, the 29th input is an integer in the range $0 \ldots 5$.

The first input is a mode indicator. It it is equal to 1, the program runs in the verbose mode, if it is 0, the program runs in silent mode.

Organization. Your program will now declare a an array

```
int electoralCollege[] = {AL, AK, AZ, AR, ..., WA, WV, WY}
```

(here we assume that symbolic constants `AL,..., WY` are `#define`d in the program).

The program, as before will read the first input, determine the mode (verbose or silent), and then read 51 numbers representing the results of the state vote. The program shall keep track of the Obama's and McCain's vote totals, and, if verbose mode is selected, will inform the user about the winner of each state (just as in Lab 3). Only now, this will be done using a loop.

Don't forget the special case, Nebraska (input number 29 of the program).

Output. The output of the program shall be **almost the same as in Lab 3**. The only extra thing you need to report is how many states voted for each candidate. The output format for the last two lines shall be:

```
Electoral College vote total for John  McCain (R) is <VOTES> (<STATES> states)
Electoral College vote total for Barack Obama (D) is <VOTES> (<STATES> states)
```

Here `<VOTES>` and `<STATES>` are the electoral college vote totals for respective candidates and the number of states that voted for them.

State name generation. To make your program a bit simpler, you need to be able to generate actual state names on the fly. Since your program will output everything in verbose mode from within its main loop, you can no longer hardcode state names in your `printf()` statements without ugly hacks.

Instead, use the following solution. Declare an array `char *stateName[]` using the declaration below:

```
char *stateName[] = {"Alabama", "Alaska", "Arizona", "Arkansas", "California",
      "Colorado", "Connecticut", "Delaware", "District of Columbia", "Florida",
      "Georgia",  "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
```

```
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsilvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"};
```

(Note, this is your first encounter with C strings. We will talk about them later during the quarter.)

To output the state name use the following printf statements:

```
printf("%s goes to Obama\n", stateName[i]);
```

and

```
printf("%s goes to McCain\n", stateName[i]);
```

Here `int i` is the variable indcating which state name needs to be reported.

Program Name. Name your program `elections.c`.

Elections histogram: elections-graph.c

This program will take as input the same files as the `elections.c` but instead of text output, it will produce a PPM file representing graphically the election results.

Input. The input is the same as for the `elections.c` program. The first number from the input shall be read by the program, but it will not be used (feel free to read it and ignore it). The remaining 51 input numbers will determine the histogram.

Output. The output of your program is a PPM file with the following properties:

- The dimensions of the image are 590×50.
- Each column of the image represents either one electoral vote for a given state or a separator between the states.
- There are 50 state separator lines inside the image. Additionally, columns 1 and 590 are used to frame the image and will also be referred to as "separators" below. The image width is thus 52 (separators) + 538 (electoral votes) = 590 columns.
- If a column is a separator, the color of all pixels in it is **black**.
- The top (row 1) and bottom (row 50) of the image are colored **black**. This will frame the actual histogram.
- Each state is represented by the number of columns equal to its number of Electoral College votes (e.g., Alabama is represented by 9 columns, Hawaii — by 3 columns, Maryland – by 10, etc.).

- Each column (row 2 — row 49) representing an electoral vote of a state will be painted either **red** if the state gave that vote to McCain or **blue** if the state gave the vote to Obama. For all states by Nebraska this means that all its columns will be painted either **red** or **blue**.

  For Nebraska, Obama votes are painted first, followed by McCain votes.

- Information about state votes is output in the alphabetical order by the state name - same order in which the input is organized.

Program organization. As in `elections.c` program, use an array (`int electoralCollege[]` to represent the Electoral College votes each state has. Additionally, you will need to declare one more array, e.g., `int results[51]`[1]. Your program shall work as follows:

1. Read the input. Ignore the first number, store all remaining numbers in the `results` array.
2. Construct the output image. Remember, the first and the last row of the image are **black**. Rows 2 through 49 are constructed as follows:
   - Output a **black** pixel.
   - Scan the `results` array. On each step `i`, determine who won the state and output `electoralCollege[i]` pixels of appropriate color. If the current step is 28 (Nebraska), determine the number of Obama's votes and output that number of **blue** pixels. Output the remaining pixels for Nebraska in **red**.
   - When the array scan is over, output a **black** pixel.

Notes. Name your program `elections-graph.c`.

Temperature conversion program: converter.c

Modify your temperature conversion program from Labs 2 and 3 as follows:

NNCR0. Your program run in a loop. First, the program shall ask the user to enter the temperature in degrees Fahrenheit. After a good temperature is entered, the program shall convert it to degrees Celcius and output the result, using the same output format as in Lab 3[2]. After that, the program will ask the use whether the user wants to convert another temperature and will read the user input. The user can select to quit or to enter another number. If the user selects to enter another number, the program returns to asking the user to enter the temperature. If the user selects to quit, the program shall print a goodbye message and terminate.

NNCR1. User inputs (both) must be guarded using a sentinel loop. Temperature in degrees Fahrenheit cannot be lower than **-459.67**. Your program shall ask for input until a proper value is entered.

NNCR2. At the end of each loop your program shall output

[1] You will actually need to `#define` the number of states as a symbolic constant.

[2] I have modified the spelling in the program to be correct and consistent. Recommend that you do the same.

```
Continue? (1 - more conversions; 0 - quit):
```

and read an `int` value until the user input is either 0 or 1. If the value entered is `1`, the program shall continue asking for temperature values for conversion. If the value entered is `0`, the program shall output

```
See you soon!
```

and terminate. The `Continue?...` prompt shall be repeated each time an incorrect value is entered.

Notes. Name your program `converter.c`

Yahtzee roll: yahtzee.c

This program will require the use of a random number generator functionality available to you in C. Please read Appendix A for more comments on the random number generation in C.

The game of Yatzee is played with five six-sided dice. The objective of the game is to roll the dice to obtain various combinations of numbers, from simple pairs and three-of-a-kind rolls to the "yahtzee" combination (all five dice show the same number).

Your program will generate successivly a series of dice rolls until it produces a **"Yahztee"**, i.e., a combination of five dice showing the same number. It will then report the roll and some simple statistics.

Your program shall support two modes silent and verbose. In silent mode, only the final roll and the statistics are reported. In the verbose mode, your program shall report each roll.

The functional requirements for the program are below.

YR1. Your program shall start by printing

```
Yahtzee Roll: choose your mode!
0 - silent
1 - verbose
What will it be?
```

and read an `int` value from the keyboard until either `0` or `1` is read.

YR2. The program shall declare an integer array `int dice[5]` to store information about each dice roll. Upon declaration, the program shall initialize it to set all its elements to 0.

R3. The program shall proceed by rolling five dice and recording the result in the `dice` array. A "roll" of a die is simulated by using C's random number generator (please remember to reset the seed at the beginning of the program).

R4. If the output mode is verbose, the program shall output the following information:

- the current number of the roll (first, second, etc. . .);
- the values of all dice.

The output shall be formatted as shown below on the example of a roll that resulted in 2,3,4,4,5:

```
Roll #1: 2  3  4  4  5
```

R5. Your program shall check if the current roll resulted in a **"Yahtzee"** combination (i.e., if all dice are of the same denomination). If a **"Yahtzee"** combination was reached,

- In a verbose mode, your program shall print `YAHTZEE!` after the roll output (see below).
- In a silent mode, your program shall output the entire roll followed by `YAHTZEE!`.

In both cases, the output for the final roll shall look the same:

```
Roll #17: 5  5  5  5  5   YAHTZEE!
```

After that, your program output

```

It took you <N> rolls to get a YAHTZEE roll
```

(notice an empty line in front of the text) and terminate.

R6. If a **Yahtzee** combination was not reached, your program shall continue onto the next roll (hint: use a loop).

Note. Name your program `yahtzee.c`.

Note. Your program has essentially only two inputs: `0` and `1`. However, if your seed your random number generator based on, e.g., current time, you will always get a different execution on each run. (If you seed your random number generator to the same seed each time, your execution will always be the same). This program will be tested interactively.

Using arrays for image production

Until now, we produced PPM images "on the fly" - for each pixel enumerated row-wise we computed its color and immediately outputed it. This is convenient and fast for simple images, however, images with more objects on them can be created in a much more convenient way by preparing an array of pixel colors first, and then, by printing it out in one fell swoop. One of the benefits of this approach is that a color initially assigned to a specific pixel can be later overwritten, i.e., replaced with a new color. This makes it easier to draw multiple overlapping objects: you establish the order of depth and paint objects one after another starting with the deepest object on the image.

Given symbolic constants `HEIGHT` and `WIDTH` representing the dimensions of the PPM image (and a sybolic constant `COLORS` defined as `#define COLORS 3`, a PPM image can be represented by the following array:

```
unsigned char image[WIDTH][HEIGHT][COLORS];
```

Here `image[j][i]` refers to the color of pixel in row `i` and column `j` of the PPM image if we assume that the first pixel has coordinates $(0, 0)$. The `Red` component of the color is `image[j][i][0]`, the `Green` component is `image[j][i][1]` and the `Blue` component is `image[j][i][2]`.

To output a pixel, you can use the following printf statement:

```
printf("%c%c%c",image[j][i][0],image[j][i][1],image[j][i][2]);
```

Using arrays to represent PPM images, you will create a few programs described below. Each program will consist of two stages. On stage 1, the program will fill the `image` array with pixel colors according to what the image should look like. On stage 2, the completed array is scanned row-wise and the pixel colors are output, forming a PPM file.

Rainbow Box image: rainbox.c

Write a program that outputs a 700×700 PPM image which constructs a *rainbow of boxes* starting with **red** color on the outside and ending with **violet** color on the inside. The full order of colors, the dimensions of the boxes and the RGB values to use in the program are specified in the table below. All boxes are concentric (it is your job to correctly identify where each square/box starts).

No.	Square size	Color	Red	Green	Blue
1.	700×700	Red	255	0	0
2.	600×600	Orange	255	128	0
3.	500×500	Yellow	255	255	0
4.	400×400	Green	0	255	0
5.	300×300	Light Blue	0	178	255
6.	200×200	Blue	0	0	255
7.	100×100	Violet	85	26	139

Your program **shall have no** *conditional statements* used to decide pixel colors. Instead it shall consecutively color the seven boxes (squares) defined above.

Program name. Name your program `rainbox.c` (yes, I know, this is a horrible pun.)

Chessboard image: chessboard.c

Create an 800×800 image that represents on empty chess board.

A chessboard consists of eight rows and eight columns of **white** and **black** squares. Each square on your image then will have size 100×100 pixels The columns are numbered `a, b, c, d, e, f, g` from left to right. The rows are numbered `1,2,3,4,5,6,7,8` from *bottom* to *top* (thus, pixel (0,0) is in the `a8` square).

Square `a1` is **black**. **Black** and **white** squares alternate both on rows and columns.

Your program shall use no *conditional statements* for determining the color of each pixel. (Note, however, that there is no need to overwrite any color to create this image. You are still required to create the array representing the image though. This will be useful for your next assignemnt.)

Program name. Name your program `chessboard.c`

Checkers image: checkers.c

This program extends your `chessboard.c` program[3]. It will read from input a list of chessboard squares, and output a chessboard with checkers positioned on the specified squares.

Input. The input to your program has the following format. First, your program shall read a single `int` number. This number will specify how many checker pieces are to be put on the board. After that, your program shall read pairs of values. The first value is a `char` character, the second value is an `int`. The number of shall be equal to the first input value of your program. The legal values for the `char` value are `'a','b','c','d','e','f','g',h'`. The legal values for the `int` values are 1—8. A sample input may look as follows:

```
4
b2
c3
a3
d6
```

The first value in the input indicates the number of checker pieces to be put on the chessboard. Each subsequent pair of values indicates the position of a checker on the board. `a1` is the bottow left corner of the chessboard, `h8` is the top right corner. `a` through `h` are known as *files* or *verticals*. 1 through 8 are known as *ranks* or *horizontals*.

Your program is responsible for determining the validity of each square description. If a square description is invalid (e.g., `a9` or `F4` or `w5`), this input shall be ignored, **but counted** as one of the specified number of points, i.e., the following input, which contains one incorrect point is correct:

```
4
a1
b3
q4
f5
```

After reading `q4` your program shall read only one other pair of inputs.

[3] International checkers are played on a 10×10 checkerboard. However, an 8×8 board is used for a version of checkers known as Russian checkers. For simplicity, this program will use the 8×8 chessboard that you have generated using `chessboard.c` program.

Checker pieces. A checker piece in this program shall be represented by a *filled* **white** circle of *radius 40* centered in the center point with *relative coordinates* (49,49) of the appropriate chessboard square.

Please note, that such pieces shall only be visible on **black squares** of the chessboard. This is by design (checkers are played only on black squares of the board). For simplicity, no black checker pieces are used in this program.

If the input contains a **white** square, your program is allowed to do nothing or to simply place a **white** checker piece on that square (which will yield the same image). This is expected behavior.

Program flow. Start your program by setting the `image` array to contain the chessboard image as constructed by you in the `chessboard.c` program.

Once the chessboard is prepared, start processing input. Read the total number of checker pieces. For each piece, read from input its location, determine if the location is valid, and, if valid, proceed to drawing the checker piece in the correct location.

Once all checker pieces are drawn, output the image.

Drawing a checker piece. You can draw a checker piece in the following way:

1. First, determine the top left corner of the chessboard square that should contain the piece. This requires some artihmetics, but notice, that your *rank* designator (`'a'` through `'h'`) can be turned into an integer with value of 1 through 8 using the following expression: `rank - 'a' + 1`. (assuming `char rank` is the variable used in your program).

   The rest of the arithmetics is left to you.

2. While the actual checker piece is smaller than the entire chessboard square, for simplicity you can redraw the entire chessboard square. When redrawing it, you are essentially trying to create an image of a (**black** under normal circumstances) 100×100 square with a filled circle in the middle of it. This can be done in a manner similar to the national flag of `Laos` (your `laos.c` program), except that you do not need to color the pixels outside of circle (since they already have been given a color).

   You are allowed to use *conditional statements* in this part of the program (but not in the part that draws the actual chessboard).

Program name. Name your program `checkers.c`.

Submission.

Files to submit. Each pair submits one set of files from one account. The following files are mandatory:

```
team.txt,
elections.c
elections-graph.c
converter.c
yahtzee.c
```

```
rainbox.c
chessboard.c,
checkers.c
```

You also must submit all your unit test C programs. Feel free to use any filenames for them.

`team.txt` file shall contain the name of the team and the names of all team members in each pair, and the Cal Poly IDs of each. E.g, if I were on the team with Dr. John Bellardo, my `team.txt` file would be

```
Go, Poly!
John Bellardo, bellardo
Alex Dekhtyar, dekhtyar
```

Submit `team.txt` as soon as you form your group.

Files can be submitted one-by-one, or all-at-once.

Submission procedure. You will be using `handin` program to submit your work. The procedure is as follows:

- `ssh` to `vogon` (`vogon.csc.calpoly.edu`).

```
> handin dekhtyar lab05  <your files go here>
```

`handin` is set to stop accepting submissions 24 hours after the due time.

Testing and Grading

Any submitted program that does not compile earns 0 points.

Appendix A: Random Number generation in C

C provides two functions from `stdlib.h` standard library to deal with generation of **pseudorandom numbers**.

Pseudorandom numbers. Often programs need to use a random number as part of their work. Random numbers are used in various game programs to allow computer to simulate various situations.

As such, computers cannot generate true random numbers. Instead, special algorithms are created that generate a sequence of numbers that look random, and on subsequent requests for "random" numbers return members of the sequence. This metohd is called **pseudorandom number generation**.

In C, two functions are responsible for generating pseudorandom numbers.

int rand() . This function returns a number from the pseudorandom number sequence. This number is in the range between 0 and `RAND_MAX`, a symbolic constant defined in `<stdlib.h>` library[4].

[4] For the `gcc` compiler used on CSL `RAND_MAX` is equal to 2147483647.

void srand(unsigned int seed) . This function is responsible for determining from which place in the pseudorandom number sequence the `int rand()` function will retrieve the numbers. The `unsigned int seed` parameter is used to determine the starting point in the sequence.

Random number generation. Usually, to generate random numbers in a program, one make a single call to `srand()` at the beginning of the program. This sets the random number sequence. Later in the program, whenever a random number is needed, a call to `rand()` function is made. The resulting pseudorandom number can then be processed to obtain a pseudorandom number in any desired interval.

Example. Here is a simple code fragment that seed the random number generator and then retrieves and prints two random numbers:

```
int r;

srand(2);
r = rand();
printf("First random number is %d\n",r);
r = rand();
printf("Second random number is %d\m",r);
```

Tying srand() to time. Seeding the random number generator using a constant value (like in the example above) will yield **the same sequence of pseudorandom numbers** used in the program on every run of the program (try it yourselves!). If you want to make sure that different runs of the program yield distrinctly different outcomes, you need to seed the random number generator using a value that is somewhat unique to the specific run of the program. Typically, current timestamp is used for that. A simple way of seeding the random number generator is to use the `time()` function from the `time.h` package.

The following example illustrates how this function can be used in combination with `srand()`.

```
#include  <stdlib.h>  /* rand() and srand() are here */
#include <time.h>     /* time() is defined here      */
#include <stdio.h>    /* printf()                    */

int main() {

 int r;

 srand(time(0));  /* time() is called with 0 as input,      */
                  /* and its return value is fed to srand() */

 r = rand();
 printf("First random number is %d\n",r);
 r = rand();
 printf("Second random number is %d\m",r);

 return 0;
}
```

Remember to use `srand()` exactly once at the beginning of your program.

Appendix B: PPM Format Explanation

Portable Pixel Map (`.ppm`) file format is a simple format for storing graphical images. Files in this format can easily be created using C programs.

Basics. An computer image is a two-dimensional grid of pixels. Each pixel represents the smallest undivisible part of the computer screen. An image file is an assignment of color to each pixel. Pixels are referred to by their Cartesian coordinates. The top left corner of an image has the coordinates (0,0). The bottom right corner has the coordinates (n, m) where n is the width of the image in pixels and m is the height of the image in pixels.

Colors. Portable pixel map files use RGB (Red, Green, Blue) color format to represent the color of each pixel. In RGB format, a color of a single pixel is separated into three components: the red component, the green component and the blue component. The final color of an RGB pixel is determined by combining the Red, Green and Blue components into a single color.

In our course, all individual RGB component intensities range from 0 (not visible) to 255 (highest intensity) and are represented as integer numbers. RGB color (0,0,0) is black, RGB color (255,255,255) is white. The table below contains the list of colors used in this lab and their RGB values.

Color	RGB Red	RGB Green	RGB Blue
black	0	0	0
white	255	255	255
red	255	0	0
green	0	255	0
blue	0	0	255
yellow	255	255	0
purple	255	0	255
orange	255	128	0
dark blue	0	0	80

File format. There are two PPM formats: a "raw" PPM file and a "plain" (ASCII) PPM file. ASCII PPMs are human-readable, but they take too much space. Raw PPMs are smaller in size, but cannot be read by a human. In this lab you will be generating raw PPM files.

From http://netpbm.sourceforge.net/doc/ppm.html (with some modifications):

Each PPM image consists of the following:

1. *A "magic number" for identifying the file type. A ppm image's magic number is the two characters* `"P6"`*.*
2. *Whitespace (blanks, TABs, CRs, LFs).*
3. *A width, formatted as ASCII characters in decimal.*
4. *Whitespace.*
5. *A height, again in ASCII decimal.*

6. *Whitespace.*
7. *The maximum color value (Maxval), again in ASCII decimal. Must be less than 65536 and more than zero.*
8. *A single whitespace character (usually a newline).*
9. *A raster of Height rows, in order from top to bottom. Each row consists of Width pixels, in order from left to right. Each pixel is a triplet of green, blue and red intensities, in that order*[5].

Representing Colors in C. Outputing raw PPM files is actually quite simple. The idea is to use `unsigned char` variables to store information about RGB intensities.

Variables of type `char` and `unsigned char` are treated by C both as a character and as a number in the range -128 – 127 or 0 — 255 respectively. The following code outputs an RGB triple to stdout.

```
unsigned char Rcolor, Bcolor, Gcolor;
Rcolor = 255;
Bcolor = 0;
Gcolor = 128;

printf("%c%c%c", Gcolor, Bcolor, Rcolor);
```

[5] This is what worked for me. If your colors don't look right, switch to RGB order.

Appendix C: Electoral College Information

No.	State	Abbr.	Population	Electoral College Votes
1.	Alabama	AL	4,500,752	**9**
2.	Alaska	AK	648,818	**3**
3.	Arizona	AZ	5,580,811	**10**
4.	Arkansas	AR	2,725,714	**6**
5.	California	CA	35,484,453	**55**
6.	Colorado	CO	4,550,688	**9**
7.	Connecticut	CT	3,483,372	**7**
8.	Delaware	DE	817,491	**3**
9.	District of Columbia	DC	563,384	**3**
10.	Florida	FL	17,019,068	**27**
11.	Georgia	GA	8,684,715	**15**
12.	Hawaii	HI	1,257,608	**4**
13.	Idaho	ID	1,366,332	**4**
14.	Illinois	IL	12,653,544	**21**
15.	Indiana	IN	6,195,643	**11**
16.	Iowa	IA	2,944,062	**7**
17.	Kansas	KS	2,723,507	**6**
18.	Kentucky	KY	4,117,827	**8**
19.	Louisiana	LA	4,496,334	**9**
20.	Maine	ME	1,305,728	**4**
21.	Maryland	MD	5,508,909	**10**
22.	Massachusetts	MA	6,433,422	**12**
23.	Michigan	MI	10,079,985	**17**
24.	Minnesota	MN	5,059,375	**10**
25.	Mississippi	MS	2,881,281	**6**
26.	Missouri	MO	5,704,484	**11**
27.	Montana	MT	917,621	**3**
28.	Nebraska	NE	1,739,291	**5**
29.	Nevada	NV	2,241,154	**5**
30.	New Hampshire	NH	1,287,687	**4**
31.	New Jersey	NJ	8,638,396	**15**
32.	New Mexico	NM	1,874,614	**5**
33.	New York	NY	19,190,115	**31**
34.	North Carolina	NC	8,407,248	**15**
35.	North Dakota	ND	633,837	**3**
36.	Ohio	OH	11,435,798	**20**
37.	Oklahoma	OK	3,511,532	**7**
38.	Oregon	OR	3,559,596	**7**
39.	Pennsylvania	PA	12,365,455	**21**
40.	Rhode Island	RI	1,076,164	**4**
41.	South Carolina	SC	4,147,152	**8**
42.	South Dakota	SD	764,309	**3**
43.	Tennessee	TN	5,841,748	**11**
44.	Texas	TX	22,118,509	**34**
45.	Utah	UT	2,351,467	**5**
46.	Vermont	VT	619,107	**3**
47.	Virginia	VA	7,386,330	**13**
48.	Washington	WA	6,131,445	**11**
49.	West Virginia	WV	1,810,354	**5**
50.	Wisconsin	WI	5,472,299	**10**
51.	Wyoming	WY	501,242	**3**