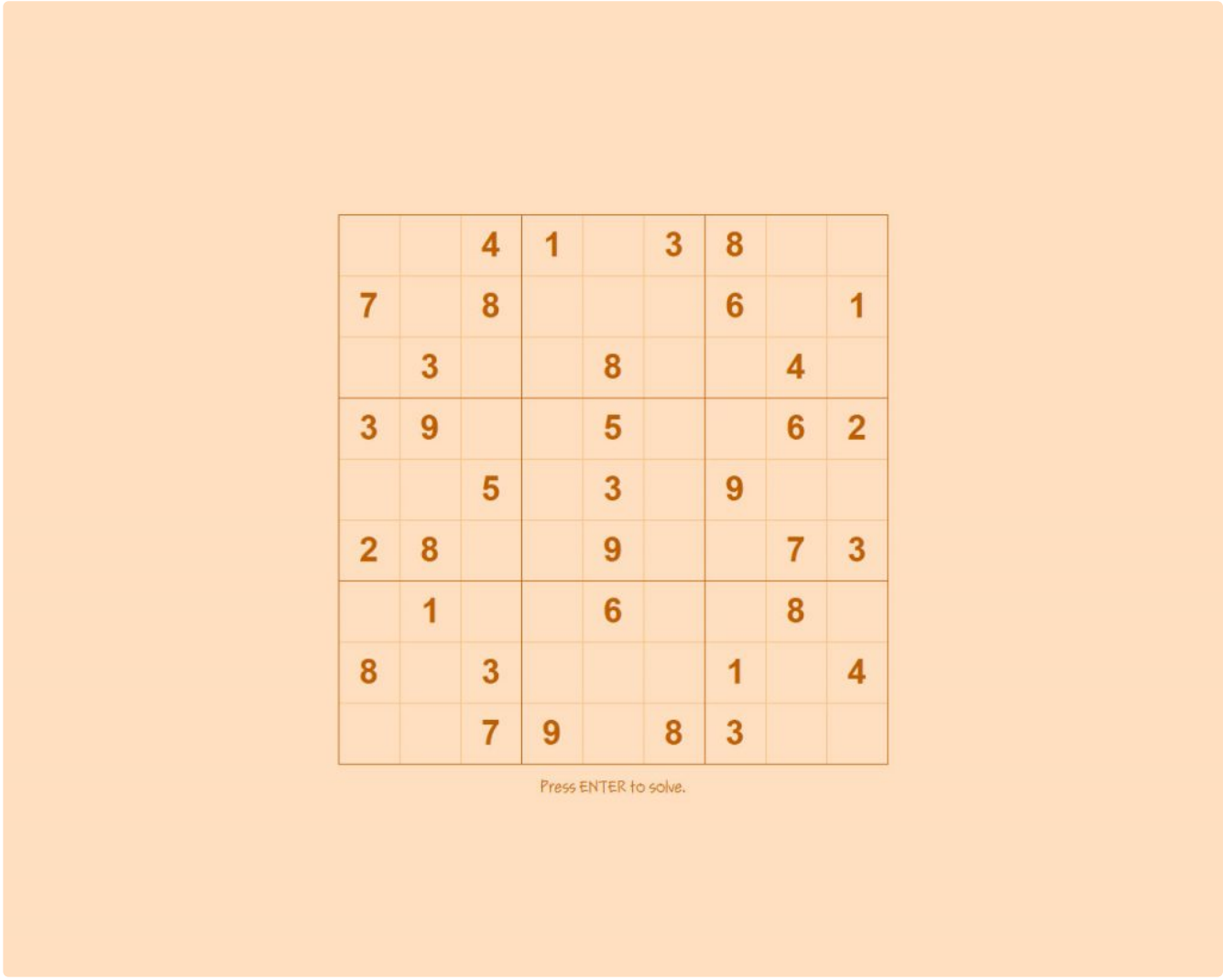
Sudoku Solver

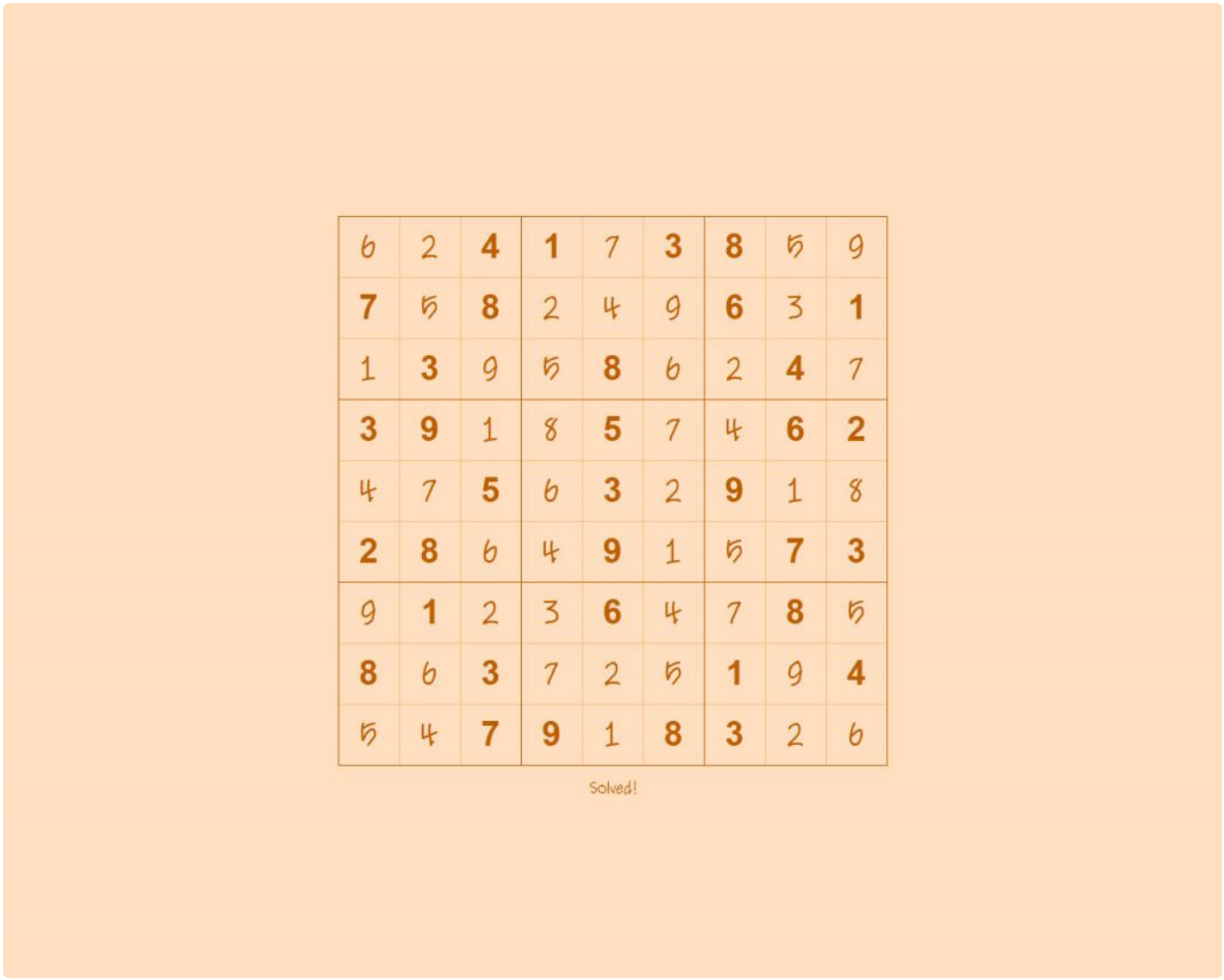
gerryrzeppa Uncategorized May 14, 2018 2 Minutes

This is a Sudoku puzzle:



The object is to get one, and only one, of each digit in each row, column, and 3-by-3 block.

This is what it looks like after my Plain English program solves it:



And this is the Plain English routine that does the deed:

```
To solve a sudoku puzzle:
Get a blank cell. If the blank cell is nil, exit.
Loop.
Add 1 to a number. If the number is greater than 9, exit.
If the number is not valid in the blank, repeat.
Put the number into the blank.
Solve the puzzle. If the sudoku puzzle is solved, break.
Erase the blank cell.
Repeat.
Display message “Solved!”.
```

Technically speaking, it’s a “recursive backtracking” routine. It tries a number in a blank cell, then calls itself to try another number in another blank cell. If everything works out, we’re done. If we get stuck, it works backward, erasing the mistakes and trying again with a different starting number. It’s fun to watch; you just draw the puzzle on the screen right before each recursive call.

I noticed that there are other articles on various sites that use a similar algorithm, and I thought a brief comparison might be instructive and edifying. Below is a C/C++ equivalent of the above Plain English routine that I found:

```
bool SolveSudoku( int grid[N][N])
{
     int row, col;
     // If there is no unassigned loc, we are done
     if (!FindUnassignedLocation(grid, row, col))
        return true ;  // success!
     // consider digits 1 to 9
     for ( int num = 1; num <= 9; num++)
     {
         // if looks promising
         if (isSafe(grid, row, col, num))
         {
             // make tentative assignment
             grid[row][col] = num;
             // return, if success, yay!
             if (SolveSudoku(grid))
                 return true ;
             // failure, unmake & try again
             grid[row][col] = UNASSIGNED;
         }
     }
     return false ;  // this triggers backtracking
}
```

I showed the two versions to my wife, and she said, “Kind of makes you wonder why some people say that Plain English is too verbose!” So we did some counting:

Lines of code: Plain English, 10; C/C++, 27
(more than double in the C/C++ version)

Characters: Plain English, 334; C/C++, 681
(more than double in the C/C++ version)

Punctuation Marks: Plain English, 19; C/C++, 70
(more than three times as many in the C/C++ version)

Conclusion? Plain English is easier to learn, easier to remember, easier to think about, easier to type, easier to read. And reasonably compact (*more* compact in some cases, like the one above).