Sudoku Programming

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <windows.h>

-> These are the header files we need to use to create our sudoku program.

const int Empty_value = 0;

-> This line creates a constant variable named Empty_value and assigns it the value of 0. The const keyword indicates that this variable cannot be modified later in the program. It will be used to get empty space we have to fill in sudoku.

const int Puzzle_size = 9;

-> This line creates another constant variable named Puzzle_size and assigns it the value of 9. Puzzle_size will be used in array as array[Puzzle_size][Puzzle_size];

To make a 9x9 grid.

int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

-> This line declares and initializes an array variable named array that can hold up to 9 integers. The integers 1 through 9 are provided as initial values for the elements of the array.

int numberOfSolution = 1;

-> This line declares and initializes an integer variable named numberOfSolution with a value of 1. The purpose of this variable is to set the no. of possible solutions of a generated sudoku to 1 only. It will be used in conditional statements as if numberOfSolution > 1 || < 1 sudoku have some problem.

typedef int bool;

#define true 1

#define false 0

-> This code defines a custom boolean data type and constants for true and false.

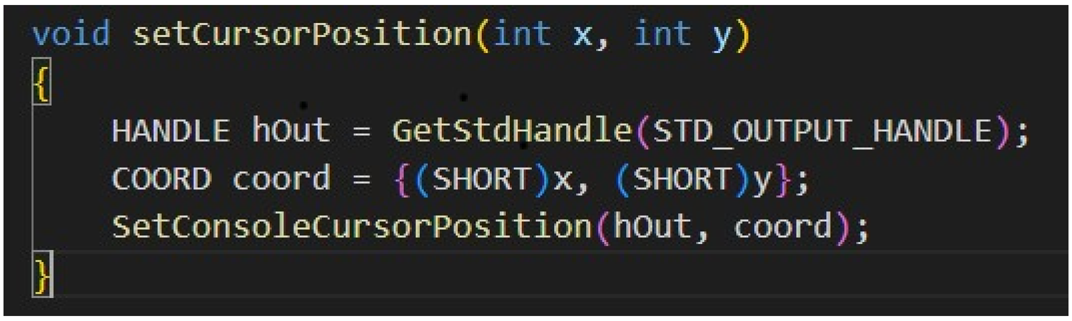
- **typedef int bool; :** This line creates a new data type called bool that is equivalent to the int data type. This is a common practice when working with boolean values in C, as the language does not have a built-in boolean data type like other programming languages.
- **#define true 1:** This line defines a constant true as having a value of 1. The #define preprocessor directive is used to define a macro that will replace all

instances of true in the code with the value 1 at compile time.

- **#define false 0:** This line defines a constant false as having a value of 0. Similarly, the #define preprocessor directive is used to define a macro that will replace all instances of false in the code with the value 0 at compile time.

These are used as;

bool clear = true;



```
void setCursorPosition(int x, int y)
{
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD coord = {(SHORT)x, (SHORT)y};
    SetConsoleCursorPosition(hOut, coord);
}
```

-> This code defines a function named setCursorPosition that sets the position of the console cursor in the console window to a specific x and y coordinate.

- **void setCursorPosition(int x, int y):** This line declares the function named setCursorPosition, which takes two integer arguments x and y representing the x and y coordinates of the desired cursor position, respectively. The void keyword indicates that this function does not return any value.
- **HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE); :** This line declares a handle hOut to the standard output device (i.e. the console window) using the GetStdHandle function from the Windows API. The

STD_OUTPUT_HANDLE constant specifies that the handle should be for the console's standard output stream.

- **COORD coord = {(SHORT)x, (SHORT)y}; :** This line creates a COORD structure named coord to hold the x and y coordinates for the cursor position. The SHORT data type is used to ensure that the coordinates fit within the range of valid console window coordinates.
- **SetConsoleCursorPosition(hOut, coord); :** This line calls the SetConsoleCursorPosition function from the Windows API, passing in the handle to the console's standard output device and the COORD structure containing the desired cursor position. This function moves the console cursor to the specified position in the console window.

In summary, this function is useful for programs that need to control the position of the console cursor, such as for creating console-based games or other interactive command-line applications. By setting the cursor position with this function, the program can output text or other content to specific locations on the console screen.

handle : In Windows operating system, the console window (also known as command prompt or terminal) is a special type of text-based user interface that allows users to interact with the operating system by running commands or executing programs. The standard output device is the default output stream for console applications, where output is directed to the console window.

In the given line of code, the GetStdHandle() function from the Windows API is used to obtain a handle to the standard output device (i.e. the console window) and store it in the variable hOut. The GetStdHandle() function takes an argument that specifies the type of standard handle to retrieve. In this case, the STD_OUTPUT_HANDLE constant is used to retrieve the handle for the console's standard output stream.

The handle returned by GetStdHandle() is an opaque value that represents the standard output device and can be used with other Windows API functions that manipulate the console window, such as SetConsoleCursorPosition() or WriteConsoleOutput(). In the given code, the handle hOut is used later in the SetConsoleCursorPosition() function to set the position of the console cursor in the console window.

COORD : COORD is a C structure type defined in the Windows API that represents a set of X and Y coordinates. It is used to represent the position of a character cell within a console screen buffer.

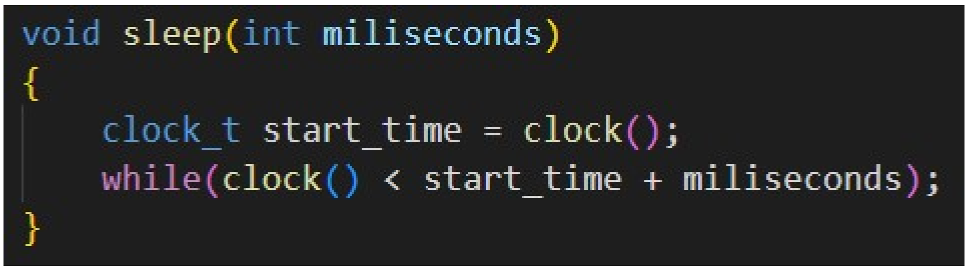
The COORD structure contains two members:

- X: an integer value that represents the horizontal coordinate of the character cell.
- Y: an integer value that represents the vertical coordinate of the character cell.

Both members of the COORD structure are defined as SHORT data type, which is a 16-bit signed integer.

COORD structures are commonly used with other Windows API functions that manipulate the console window or screen buffer, such as SetConsoleCursorPosition(), GetConsoleScreenBufferInfo(), or WriteConsoleOutput(). For example, the SetConsoleCursorPosition() function takes a COORD structure as input to set the position of the console cursor, while the WriteConsoleOutput() function takes an array of CHAR_INFO structures, each of which contains a COORD structure that specifies the position where the corresponding character should be written.

In summary, COORD is a useful structure type in Windows programming for representing positions on the console screen or other types of graphical user interfaces.



```
void sleep(int miliseconds)
{
    clock_t start_time = clock();
    while(clock() < start_time + miliseconds);
}
```

-> This code defines a function sleep() that pauses the execution of a program for a specified number of milliseconds. The function takes a single argument miliseconds, which is the duration of the pause in milliseconds.

The function works by using the clock() function from the <time.h> library to obtain the current processor time in clock ticks at the start of the function call, and storing it in the variable start_time. Then, the function enters a loop that continuously checks the current processor time using clock() and compares it to the start_time plus miliseconds to determine if the desired pause duration has been reached. The loop continues to execute until the pause duration has been reached.

Note that this implementation of sleep() is not very accurate, as it relies on the processor clock to keep track of time, which can vary in accuracy depending on the system and other running processes. A more accurate implementation would use operating system-specific sleep functions or timers.

Overall, this function provides a simple way to pause the execution of a program for a specified duration, but should be used with caution and verified for accuracy in critical applications.


```
void printPuzzle(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE],
bool clear)
{
   if (clear)
   {
      setCursorPosition(0, 0);
   }
   char text[100], separator[100], padding[100];

   for (int i = 0; i < PUZZLE_SIZE; i++)
   {
      sprintf(text, "|");
      sprintf(separator, " -");
      sprintf(padding, "|");
      for (int j = 0; j < PUZZLE_SIZE; j++)
      {
         char value[10];
         if (puzzle[i][j] == EMPTY_VALUE)
         {
```

```
      strcpy(value, " ");
   }
   else
   {
      sprintf(value, "%d", puzzle[i][j]);
   }
   sprintf(text, "%s  %s  |", text, value);
   sprintf(separator, "%s------", separator);
   sprintf(padding, "%s     |", padding);
   if (j % 3 == 2 && j != PUZZLE_SIZE - 1)
   {
      sprintf(text, "%s|", text);
      sprintf(padding, "%s|", padding);
   }
}
if (i != 0 && i % 3 == 0)
{
   for (int k = 0; k < strlen(separator); k++)
   {
      if (separator[k] == '-')
      {
         separator[k] = '=';
      }
   }
}
printf("%s\n%s\n%s\n%s\n", separator, padding, text,
padding);
   }
   printf("%s\n", separator);
}
```


Explaination of above function

1.

```
void printPuzzle(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE],
bool clear)
{
   if (clear)
   {
      setCursorPosition(0, 0);
   }
   char text[100], separator[100], padding[100];
```

- printPuzzle function is defined that takes two input parameters: a 2D array puzzle of size PUZZLE_SIZExPUZZLE_SIZE and a boolean clear.
- char text[100], separator[100], padding[100]; declares character arrays to hold text, separator, and padding characters.

2.

```
for (int i = 0; i < PUZZLE_SIZE; i++)
{
        sprintf(text, "|");
        sprintf(separator, " -");
        sprintf(padding, "|");
```

- for loop iterates through the rows of the puzzle.
- sprintf function is used to initialize the text, separator, and padding strings to |, -, and | respectively.

3.

```
        for (int j = 0; j < PUZZLE_SIZE; j++)
      {
         char value[10];
         if (puzzle[i][j] == EMPTY_VALUE)
         {
            strcpy(value, " ");
         }
         else
         {
            sprintf(value, "%d", puzzle[i][j]);
         }
         sprintf(text, "%s  %s  |", text, value);
         sprintf(separator, "%s------", separator);
         sprintf(padding, "%s     |", padding);
        if (j % 3 == 2 && j != PUZZLE_SIZE - 1)
        {
            sprintf(text, "%s|", text);
            sprintf(padding, "%s|", padding);
        }
    }
```

- for loop iterates through the columns of the puzzle.
- char value[10]; declares a character array to hold the value of the current cell in the puzzle.
- If the current cell is empty, value is set to a space character. Otherwise, it is set to the integer value of the current cell using sprintf function.
- The sprintf function is used to append the value string to the text string

sprintf : sprintf is a function in C that is used to format and print a string to a buffer. It has the following syntax:

int sprintf(char *str, const char *format, ...);

It takes a pointer to a string buffer (str) where the formatted string will be written, a format string (format) that specifies how the output should be formatted, and any number of additional arguments that correspond to the placeholders in the format string.

The placeholders in the format string are indicated by a % symbol, followed by one or more formatting characters that specify the type of value that should be inserted into the string. For example, %d is used to format an integer, %f is used to format a floating-point number, and %s is used to format a string.

sprintf returns the number of characters that were written to the string buffer, not including the terminating null character. If an error occurs, it returns a negative value.

In the context of the code provided, sprintf is used to format strings that are then printed to the console using printf.

```
int isValid(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE], int row,
int col, int value)

{

  int c, r, startRow, startCol;


  for (c = 0; c < PUZZLE_SIZE; c++)

  {

    if (puzzle[row][c] == value)

      return 0;

  }


  for (r = 0; r < PUZZLE_SIZE; r++)

  {

    if (puzzle[r][col] == value)

      return 0;

  }
```

```
    startRow = floor(row / 3) * 3;
    startCol = floor(col / 3) * 3;

    for (r = startRow; r < startRow + 3; r++)
    {
        for (c = startCol; c < startCol + 3; c++)
        {
            if (puzzle[r][c] == value)
                return 0;
        }
    }

    return 1;
}

int hasEmptyCell(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE])
{
    int r, c;

    for (r = 0; r < PUZZLE_SIZE; r++)
    {
        for (c = 0; c < PUZZLE_SIZE; c++)
        {
            if (puzzle[r][c] == EMPTY_VALUE)
                return 1;
```

```
      }
   }

   return 0;
}

void copyPuzzle(int origin[PUZZLE_SIZE][PUZZLE_SIZE], int
copy[PUZZLE_SIZE][PUZZLE_SIZE])
{
   int r, c;

   for (r = 0; r < PUZZLE_SIZE; r++)
   {
      for (c = 0; c < PUZZLE_SIZE; c++)
      {
         copy[r][c] = origin[r][c];
      }
   }
}
```


Explaination of above functions

1. isValid function:

a)

int isValid(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE], int row, int col, int value)

-> This function takes in a 2D array of integers puzzle, a row number row, a column number col, and a value value, and returns an integer (1 or 0) indicating whether the value can be placed in the cell at position (row, col) in the puzzle array according to the rules of Sudoku.

b)

```
int c, r, startRow, startCol;
```

-> This line declares four integers that will be used as loop counters and to store the starting row and column numbers for the 3x3 sub-grid that contains the cell at position (row, col).

c)

```
for (c = 0; c < PUZZLE_SIZE; c++)
{
   if (puzzle[row][c] == value)
      return 0;
}
```

-> This loop iterates through all the cells in the same row as the cell at position (row, col) and checks whether any of them already contain the value. If it finds such a cell, it immediately returns 0, indicating that the value cannot be placed in the cell at position (row, col).

d)

```
for (r = 0; r < PUZZLE_SIZE; r++)
{
   if (puzzle[r][col] == value)
      return 0;
}
```

-> This loop iterates through all the cells in the same column as the cell at position (row, col) and checks whether any of them already contain the value. If it finds such a cell, it immediately returns 0, indicating that the value cannot be placed in the cell at position (row, col).

e)

```
startRow = floor(row / 3) * 3;

startCol = floor(col / 3) * 3;
```

-> These two lines calculate the starting row and column numbers for the 3x3 sub-grid that contains the cell at position (row, col). The floor function is used to round down the row and column numbers to the nearest multiple of 3.

f)

```
for (r = startRow; r < startRow + 3; r++)
{
  for (c = startCol; c < startCol + 3; c++)
  {
    if (puzzle[r][c] == value)
      return 0;
  }
}
```

-> This nested loop iterates through all the cells in the 3x3 sub-grid that contains the cell at position (row, col) and checks whether any of them already contain the value. If it finds such a cell, it immediately returns 0, indicating that the value cannot be placed in the cell at position (row, col).

g)

```
return 1;
```

If none of the previous checks have caused the function to return 0, then it means that the value can be placed in the cell at position (row, col) according to the rules of Sudoku. Therefore, the function returns 1, indicating that the value is valid.

2. hasEmptyCell function:

-> The hasEmptyCell function is used to check whether the given Sudoku puzzle has any empty cells or not. Here's an explanation of each line:

a)

```
int hasEmptyCell(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE])
```

-> This line defines a function called hasEmptyCell that takes a 2D array puzzle of size PUZZLE_SIZE x PUZZLE_SIZE as input and returns an integer value.

b)

```
{
  int r, c;
```

-> This line declares two integer variables r and c.

c)

```
  for (r = 0; r < PUZZLE_SIZE; r++)
  {
    for (c = 0; c < PUZZLE_SIZE; c++)
    {
      if (puzzle[r][c] == EMPTY_VALUE)
        return 1;
    }
```

```
   }
```

-> These lines use a nested loop to iterate over each cell in the puzzle. If an empty cell is found (i.e., a cell with a value of EMPTY_VALUE), the function immediately returns 1 to indicate that the puzzle has at least one empty cell.

d)

```
   return 0;

}
```

-> If the function has not yet returned after iterating over all cells, it means that no empty cells were found, so the function returns 0 to indicate that the puzzle is complete (i.e., all cells have non-zero values).

3. copyPuzzle function:

The copyPuzzle function is used to create a copy of a two-dimensional integer array representing a Sudoku puzzle. The function takes two arguments: origin and copy, which are both two-dimensional integer arrays of size PUZZLE_SIZE x PUZZLE_SIZE.

Here is a line-by-line explanation of the copyPuzzle function:

a)

```
void copyPuzzle(int origin[PUZZLE_SIZE][PUZZLE_SIZE], int copy[PUZZLE_SIZE][PUZZLE_SIZE])
```

-> This line declares the function copyPuzzle with two parameters: origin and copy. Both are two-dimensional integer arrays of size PUZZLE_SIZE x PUZZLE_SIZE.

b)

```
{
  int r, c;
```

-> This line declares two integer variables r and c which will be used as indices to iterate through the two-dimensional array.

c)

```
  for (r = 0; r < PUZZLE_SIZE; r++)
  {
    for (c = 0; c < PUZZLE_SIZE; c++)
    {
      copy[r][c] = origin[r][c];
    }
  }
```

-> This is the main logic of the copyPuzzle function. It iterates through every cell in the two-dimensional origin array, and copies the value of that cell to the corresponding cell in the copy array.

At the end of the function, the copy array will contain an exact copy of the origin array.

The reason why we need this function in C is that arrays are passed to functions by reference, not by value. This means that any changes made to an array inside a function will affect the original array that was passed to the function. In the context of a Sudoku solver, we want to be able to try different values in different cells of the puzzle without modifying the original puzzle, so we need to make a copy of it. This is why the copyPuzzle function is necessary.

```
---
void shuffle(int arr[], int n)
{
   srand(time(NULL));
   for (int i = n - 1; i > 0; i--)
   {
      int j = rand() % (i + 1);
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
   }
}

bool fillPuzzle(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE])
{
   int row, col;
   for (int i = 0; i < PUZZLE_SIZE * PUZZLE_SIZE; i++)
   {
      row = floor(i / PUZZLE_SIZE);
      col = i % PUZZLE_SIZE;
      if (puzzle[row][col] == EMPTY_VALUE)
      {
         shuffle(values, PUZZLE_SIZE);
         for (int j = 0; j < PUZZLE_SIZE; j++)
         {
             if (isValid(puzzle, row, col, values[j]))
             {
                 puzzle[row][col] = values[j];
                 if (!hasEmptyCell(puzzle) || fillPuzzle(puzzle))
                 {
                     return true;
                 }
             }
         }
         break;
      }
   }
   puzzle[row][col] = EMPTY_VALUE;
   return false;
}
```

Explaination of above functions

1. shuffle function:

The shuffle function shuffles an array of integers randomly using the Fisher-Yates shuffle algorithm. Here's a line-by-line explanation of the function:

```
void shuffle(int arr[], int n)
{
  // Seed the random number generator using the current
time.
  srand(time(NULL));

  // Iterate over the array in reverse order.
  for (int i = n - 1; i > 0; i--)
  {
      // Generate a random index j between 0 and i (inclusive).
      int j = rand() % (i + 1);

      // Swap the elements at indices i and j.
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
  }
}
```

->

- **void shuffle(int arr[], int n) -** defines a function shuffle that takes an integer array arr and its length n as arguments, and does not return any value.
- **srand(time(NULL)); -** seeds the random number generator using the current time, which ensures that the generator produces a different sequence of random numbers each time the program runs.
- **for (int i = n - 1; i > 0; i--) -** iterates over the array in reverse order, starting from the last element and going backwards to the first element.
- **int j = rand() % (i + 1); -** generates a random integer j between 0 and i (inclusive) using the rand() function.

The % operator ensures that the result is between 0 and i by computing the remainder of the division of rand() by i + 1.

- **int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp; -** swaps the elements at indices i and j in the array arr by using a temporary variable temp to store the value at index i before overwriting it with the value at index j. The value at index j is then stored in index i, and the value in temp (which is the original value at index i) is stored in index j.

1. fillPuzzle function:

The fillPuzzle function takes a 2D array puzzle representing a Sudoku puzzle and fills in the empty cells to create a valid puzzle.

a)

```
bool fillPuzzle(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE])
```

-> The function returns a boolean value indicating whether a valid puzzle was created or not.

b)

```
{
   int row, col;
```

-> Declare variables to hold the current row and column values.

c)

```
   for (int i = 0; i < PUZZLE_SIZE * PUZZLE_SIZE; i++)
   {
      row = floor(i / PUZZLE_SIZE);
      col = i % PUZZLE_SIZE;
```

-> Loop over all cells of the puzzle by iterating over the numbers 0 to 80 (inclusive) and calculating the row and column values from the index.

d)

```
    if (puzzle[row][col] == EMPTY_VALUE)
    {
      shuffle(values, PUZZLE_SIZE);
```

-> If the current cell is empty, shuffle the array of possible values (1 to 9) using the shuffle function defined elsewhere in the code.

e)

```
      for (int j = 0; j < PUZZLE_SIZE; j++)
      {
        if (isValid(puzzle, row, col, values[j]))
        {
          puzzle[row][col] = values[j];
```

-> Loop over the shuffled values and try to fill the current cell with each value until a valid value is found using the isValid function defined elsewhere in the code. If a valid value is found, set it in the current cell.

f)

```
          if (!hasEmptyCell(puzzle) || fillPuzzle(puzzle))
          {
            return true;
          }
        }
      }
```

```
        break;
      }
    }
```

-> After attempting to fill the current cell with all possible values, if no valid value is found, back up to the previous cell and try again with a different value. If the entire puzzle has been filled without encountering any invalid cells, return true to indicate success.

g)

```
  puzzle[row][col] = EMPTY_VALUE;
  return false;
```

-> If no valid solution is found, reset the current cell to empty and return false to indicate failure.

```
bool solveSudoku(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE],
bool visualize)
{
  int row, col;
  for (int i = 0; i < PUZZLE_SIZE * PUZZLE_SIZE; i++)
  {
    row = floor(i / PUZZLE_SIZE);
    col = i % PUZZLE_SIZE;
    if (puzzle[row][col] == EMPTY_VALUE)
    {
      for (int value = 1; value <= PUZZLE_SIZE; value++)
```

```
{
  if (isValid(puzzle, row, col, value))
  {
    puzzle[row][col] = value;
    if (visualize)
    {
      sleep(100);
      printPuzzle(puzzle);
    }
    if (!hasEmptyCell(puzzle))
    {
      numberOfSolution++;
      if (visualize)
      {
        sleep(100);
        printPuzzle(puzzle);
        return true;
      }
      break;
    }
    else if (solveSudoku(puzzle, visualize))
    {
      return true;
    }
  }
```

```
         }
         break;
      }
   }
   puzzle[row][col] = EMPTY_VALUE;
   if (visualize)
   {
      sleep(100);
      printPuzzle(puzzle);
   }
   return false;
}
```

Explaination of above function

solveSudoku Function:

a)

bool solveSudoku(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE], bool visualize)

-> This is the function signature, which takes in a 2D integer array puzzle representing the Sudoku board and a boolean visualize which indicates whether or not to visualize the solving process. It returns a boolean value indicating whether or not a solution was found.

b)

int row, col;

-> Declare two integer variables to keep track of the current row and column being filled in the Sudoku board.

c)

for (int i = 0; i < PUZZLE_SIZE * PUZZLE_SIZE; i++)

-> Loop over all the cells in the Sudoku board. PUZZLE_SIZE is the size of the board (usually 9), so PUZZLE_SIZE * PUZZLE_SIZE gives us the total number of cells.

d)

row = floor(i / PUZZLE_SIZE);

col = i % PUZZLE_SIZE;

-> Calculate the current row and column based on the current index i.

e)

if (puzzle[row][col] == EMPTY_VALUE)

-> Check if the current cell is empty. EMPTY_VALUE is a constant representing an empty cell in the Sudoku board.

f)

for (int value = 1; value <= PUZZLE_SIZE; value++)

Loop over all possible values (1 to 9 for a standard Sudoku board).

g)

if (isValid(puzzle, row, col, value))

-> Check if the current value is valid for the current cell by calling the isValid function. This function checks if the value is valid for the row, column, and 3x3 sub-grid that the cell belongs to.

h)

```
puzzle[row][col] = value;
```

-> If the value is valid, set the current cell to the value.

i)

```
if (visualize)
{
   sleep(100);
   printPuzzle(puzzle);
}
```

-> If visualization is enabled, sleep for a short period of time (100 milliseconds) and print out the current state of the Sudoku board.

j)

```
if (!hasEmptyCell(puzzle))
```

-> Check if the Sudoku board is fully filled. If it is, a solution has been found.

k)

```
numberOfSolution++;
if (visualize)
{
   sleep(100);
   printPuzzle(puzzle);
   return true;
}
break;
```

-> Increment a global counter for the number of solutions found, and if visualization is enabled, print out the final state of the Sudoku board and return true. Otherwise, break out of the loop since we don't need to try any more values for this cell.

l)

```
else if (solveSudoku(puzzle, visualize))
{
   return true;
}
```

-> If the board is not fully filled and a valid value was found for the current cell, recursively call solveSudoku with the updated board. If a solution is found, return true.

m)

```
puzzle[row][col] = EMPTY_VALUE;
```

-> If no solution was found for the current value, set the cell back to empty.

n)

```
if (visualize)
{
   sleep(100);
   printPuzzle(puzzle);
}
```

-> If visualization is enabled, sleep for a short period of time and print out the current state of the Sudoku board.

o)

```
return false;
```

-> If no solution was found for the current cell, return false.

Visualize: In C, visualize is a boolean variable that can have one of two values: true or false. It is typically used as a flag to indicate whether or not to display visual output during the execution of a program.

In the context of the solveSudoku function provided, visualize is used to determine whether or not to print out the current state of the Sudoku board during the solving process. When visualize is true, the function will print out the board at certain points during the execution, allowing the user to see how the board is being filled in and potentially aiding in understanding the algorithm being used to solve the Sudoku puzzle. When visualize is false, no visual output is produced, and the function simply returns a boolean value indicating whether or not a solution was found.

In general, the use of visualization in programming can be very helpful for understanding the behavior of an algorithm or the data being processed. It can help to identify errors or inefficiencies, and can also make debugging easier by providing a clear view of the state of the program at various points during execution. However, it can also be resource-intensive and may slow down the program, so it is often used selectively or as a debugging tool rather than as a default behavior.

```
void generatePuzzle(int puzzle[PUZZLE_SIZE]
[PUZZLE_SIZE], int difficulty)

{

   int i, j, row, col, attempt, backupValue;

   for (i = 0; i < PUZZLE_SIZE; i++)
```

```
{
   for (j = 0; j < PUZZLE_SIZE; j++)
   {
      puzzle[i][j] = EMPTY_VALUE;
   }
}
fillPuzzle(puzzle);
srand((unsigned)time(NULL));
attempt = difficulty;
while (attempt > 0)
{
   row = floor(rand() % PUZZLE_SIZE);
   col = floor(rand() % PUZZLE_SIZE);
   while (puzzle[row][col] == EMPTY_VALUE)
   {
      row = floor(rand() % PUZZLE_SIZE);
      col = floor(rand() % PUZZLE_SIZE);
   }
   backupValue = puzzle[row][col];
   puzzle[row][col] = EMPTY_VALUE;
   numberOfSolution = 0;
   solveSudoku(puzzle, 0);
   if (numberOfSolution != 1)
   {
      puzzle[row][col] = backupValue;
```

```
            attempt--;
        }
    }
}
```


Explaination of above function

generatePuzzle Function:

a)

void generatePuzzle(int puzzle[PUZZLE_SIZE][PUZZLE_SIZE], int difficulty)

-> This line defines the generatePuzzle() function with two arguments: puzzle, which is a two-dimensional array representing the Sudoku puzzle, and difficulty, which is an integer value representing the difficulty level of the puzzle.

b)

int i, j, row, col, attempt, backupValue;

-> This line declares several integer variables that will be used later in the function.

c)

```
for (i = 0; i < PUZZLE_SIZE; i++)
{
    for (j = 0; j < PUZZLE_SIZE; j++)
    {
        puzzle[i][j] = EMPTY_VALUE;
```

```
  }
}
```

-> This loop initializes all elements of the puzzle array to the EMPTY_VALUE constant.

d)

```
fillPuzzle(puzzle);
```

-> This line calls the fillPuzzle() function, which fills the puzzle array with values to create a solvable Sudoku puzzle.

e)

```
srand((unsigned)time(NULL));
```

-> This line initializes the random number generator with the current time, so that the puzzle can be randomized each time the function is called.

f)

```
attempt = difficulty;
```

-> This line sets the attempt variable to the difficulty value passed to the function. This value determines how many times the function will attempt to remove a number from the puzzle to increase its difficulty.

g)

```
while (attempt > 0)
{
  row = floor(rand() % PUZZLE_SIZE);
  col = floor(rand() % PUZZLE_SIZE);
  while (puzzle[row][col] == EMPTY_VALUE)
  {
    row = floor(rand() % PUZZLE_SIZE);
```

```
        col = floor(rand() % PUZZLE_SIZE);
    }
    backupValue = puzzle[row][col];
    puzzle[row][col] = EMPTY_VALUE;
    numberOfSolution = 0;
    solveSudoku(puzzle, 0);
    if (numberOfSolution != 1)
    {
        puzzle[row][col] = backupValue;
        attempt--;
    }
}
```

-> This loop removes a number from the puzzle array difficulty times to increase the puzzle's difficulty. It does this by selecting a random row and column, checking that the selected element is not already empty, and then temporarily removing the value from the puzzle. It then calls the solveSudoku() function to check how many solutions the modified puzzle has. If the puzzle has more or less than one solution, the removed value is restored to its original location, and the attempt variable is decremented. If the puzzle has exactly one solution, the removed value remains empty, and the loop moves on to the next attempt.

Overall, this function generates a Sudoku puzzle with a specified difficulty level, and returns the puzzle as a two-dimensional array.

```
int main(int argc, char **argv) {
   int puzzle[PUZZLE_SIZE][PUZZLE_SIZE];
   int origin[PUZZLE_SIZE][PUZZLE_SIZE];
   generatePuzzle(puzzle, 1);
   copyPuzzle(puzzle, origin);
   system("clear");
   printPuzzle(puzzle, 1);
   char run[2];
   printf("solve puzzle (Y/n) ");
   scanf("%s", run);
   if (strcmp(run, "n") == 0 || strcmp(run, "N") == 0) {
      fflush(stdout);
      return 0;
   }
   solveSudoku(puzzle, 1);
   printPuzzle(origin, 0);
   fflush(stdout);
   return 0;
}
```


Explanation

a)

int main(int argc, char **argv) {

-> The main function is the entry point of the program. It takes two arguments, argc and argv, which are the number of command line arguments and an array of those arguments respectively.

b)

```
int puzzle[PUZZLE_SIZE][PUZZLE_SIZE];
int origin[PUZZLE_SIZE][PUZZLE_SIZE];
```

-> These lines declare two 2D arrays of integers, puzzle and origin, both of size PUZZLE_SIZE by PUZZLE_SIZE.

c)

```
generatePuzzle(puzzle, 1);
```

-> This line calls the generatePuzzle function and passes it the puzzle array and the integer 1 (means int difficulty = 1) as arguments. This generates a Sudoku puzzle of the specified difficulty and stores it in the puzzle array.

d)

```
copyPuzzle(puzzle, origin);
```

-> This line calls the copyPuzzle function and passes it the puzzle and origin arrays as arguments. This function copies the contents of the puzzle array into the origin array.

e)

```
system("clear");
```

-> This line clears the console screen.

f)

```
printPuzzle(puzzle, 1);
```

-> This line calls the printPuzzle function and passes it the puzzle array and the integer 1 which sets clear to true. This function prints the Sudoku puzzle to the console.

g)

```
char run[2];
printf("solve puzzle (Y/n) ");
scanf("%s", run);
```

-> These lines declare a character array run of size 2, prompt the user to enter whether they want to solve the puzzle or not, and read their input into the run array.

h)

```
if (strcmp(run, "n") == 0 || strcmp(run, "N") == 0) {
    fflush(stdout);
    return 0;
}
```

-> This if statement checks if the user entered 'n' or 'N' to indicate that they don't want to solve the puzzle. If they did, the main function returns 0 and terminates the program.

i)

```
solveSudoku(puzzle, 1);
```

-> This line calls the solveSudoku function and passes it the puzzle array and the integer 1 which set visualize to true. This function solves the Sudoku puzzle and modifies the puzzle array to contain the solved puzzle.

j)

```
printPuzzle(origin, 0);
```

-> This line calls the printPuzzle function and passes it the origin array and the integer 0 which sets clear to False. This function prints the original Sudoku puzzle to the console without the solution.

k)

```
fflush(stdout);
```

```
   return 0;
}
```

-> These lines flush the output buffer and return 0, indicating successful program termination.

THE END