**Sudoku Mania**

**Introduction**

**Purpose**

**Generating and solving a Sudoku of varying degree of toughness.**

**System Requirements-**

1. **The monitor must be set to resolution 1280x1024**
2. **The ezwindows library should be with the path homefolder/ezwcs101/ezwin/library**

**Rules-**

**The object of Sudoku is to fill all the blank spaces such that each row column and 3 x 3 grids (not all but those that divide that divide the Sudoku into 9 equal parts) should contain the numbers 1 to 9.**

**Click on the new game button and then select the level of hardness and start playing the game.**

**Generic Technology**

**C++, EZ Windows, Ubuntu, Adobe photoshop**

**Web references**

[**http://www.youtube.com/watch?v=p-gpaIGRCQI**](http://www.youtube.com/watch?v=p-gpaIGRCQI)

**Book references**

**An introduction to programming and object-oriented design by James P. Cohoon, Davidson**

**Work Distribution :**

Aditya Rajgopal took the task of programming the generating algorithm and created ‘Generate.h’ file.

Aditya Prakash Singh handled the designing part, graphics and created the ‘Sodoku\_windows.cpp’.

Akanksha Jain designed the bitmap images for graphics, tried coding of equal\_array. Also made SRS document.

Akeek Maitra tried to write no conflict function and handed to Aditya Rajgopal for verification and incorporation into ‘Generate.h’. Handled the documentation; wrote final stage report.

Aditya Kumar helped in SRS preparation, program testing, documentation.

Afsar Ali helped in overall project preparation.

**Changes:**

Instead of using 4 different windows 4 Bitmap images are used in a single window.

**Program:**

----------------------------------------------------**‘Generate.h’**----------------------------------------------

**Header files used:**

**Iostream**-contains the basic input output finctions

**Stdlib**-contains all the standard functions

**Ctime**-contains all the time related function

**Code and Explanation-**

*/\* This is the header file that contains the functionality to generate the Sudoku. To generate a Sudoku the user has to create two objects of class Sudoku and call the generate function passing the objects as parameters and also the number of numbers to hide. The program uses brute force method along with backtracking to solve Sudoku and generates by solving a Sudoku with 1 row with numbers 1 to 9.*

*\*/*

#ifndef GENERATE\_H\_INCLUDED

#define GENERATE\_H\_INCLUDED

#define UNASSIGNED 0

#include<iostream>

#include<stdlib.h>

#include<ctime>

using namespace std;

class Sudoku

{

public:

int puzzle[9][9]; *//this array contains the sudoku*

int srow; *//* *this contains the row from where to start searching //for number*

int scol; *//* *this contains the row from where to start searching //for number*

bool SolveSudoku(Sudoku &gird,Sudoku &grid2,int c,int &count);

int generate(Sudoku &grid1,Sudoku &grid2,int level);//

int print\_array(Sudoku &grid);//

int generate\_final(Sudoku &grid1,Sudoku &grid2,int level);

bool check\_equal(Sudoku &grid1,Sudoku &grid2);//

bool check\_unique(Sudoku &grid1,Sudoku &grid2);

int equal\_array(Sudoku &grid1,Sudoku &grid2);//

bool FindUnassignedPos(Sudoku &grid,int &r,int &c);//

bool NoConflict(Sudoku &grid,int r,int c,int num);//

};

*/\*The function FindUnassignedPos is a public member of the Sudoku class. It has a return type of bool it takes the object we are solving starts from the given starting point and finds the first unassigned location and returns true if found. As the row and column are passed by reference this function also sets the row and coloumn to that location.\*/*

bool Sudoku::FindUnassignedPos(Sudoku &grid,int &r,int &c)

{

int i,j;

int count=0;

int k=grid.scol;

for(i=grid.srow;i<9;i++)

{

if(count>0)

{

k=0;

}

for(j=k;j<9;j++)

{

if(grid.puzzle[i][j]==UNASSIGNED)

{

r=i;

c=j;

return true;

}

count++;

if(count==81)

{

return false;

}

}

if(i==8&&j==9)

{

i=-1;

j=0;

}

}

}

*/\*The function No-Conflict takes the object of Sudoku and the row column and checks if the given number complies with the given rules of Sudoku at that position.\*/*

bool Sudoku::NoConflict(Sudoku &grid,int r,int c,int num)

{

int i,j,k,l;

for(i=0;i<9;i++) *//Checks rows and columns*

{

if(grid.puzzle[r][i]==num||grid.puzzle[i][c]==num)

{

return false;

}

}

for(i=0;i<=6;i+=3) *//checks 3x3 grids*

{

for(j=0;j<=6;j+=3)

{

if(r<i+3&&r>=i&&c<j+3&&c>=j)

{

for(k=i;k<i+3;k++)

{

for(l=j;l<j+3;l++)

{

if(grid.puzzle[k][l]==num)

{

return false;

}

}

}

}

}

}

return true;

}

*/\*This is the function that is called to solve the Sudoku it*

*uses a backtracking algorithm to solve the Sudoku. If the parameter c is zero then the function checks numbers from 1 to 9 and if it is anything else it checks from 9 to 1. The Sudoku that is solved is the first parameter and if someone wants to check if the solution is equal to any other sudoku then they pass the sudoku they want to compare with as the second parameter. If the solution is equal to the sudoku then the variable count is set to 1.\*/*

bool Sudoku::SolveSudoku(Sudoku &grid,Sudoku &grid1,int c,int &count)

{

int row,col;

if(!grid.FindUnassignedPos(grid,row,col))

{

if(check\_equal(grid,grid1))

{

count=1;

}

return true;

}

if(c==0)

{

for(int i=1;i<=9;i++)

{

if(grid.NoConflict(grid,row,col,i))

{

grid.puzzle[row][col]=i;

//system("cls");

//grid.print\_array(grid);

if(grid.SolveSudoku(grid,grid1,c,count))

{

return true;

}

grid.puzzle[row][col]=UNASSIGNED;

}

}

}

else

{

for(int i=9;i>=1;i--)

{

if(grid.NoConflict(grid,row,col,i))

{

grid.puzzle[row][col]=i;

//system("cls");

//grid.print\_array(grid);

if(grid.SolveSudoku(grid,grid1,c,count))

{

return true;

}

grid.puzzle[row][col]=UNASSIGNED;

}

}

}

return false;

}

*/\*This is the main function that is called it requires two parameters of the Sudoku type the first one will later contain the complete solved sudoku and the second one will contain a sudoku with numbers removed equal to the third parameter.\*/*

int Sudoku::generate(Sudoku &grid1,Sudoku &grid2,int level)

{

int i,j,num;

int dummy;

for(i=0;i<9;i++)

{

for(j=0;j<9;j++)

{

grid1.puzzle[i][j]=UNASSIGNED;

grid2.puzzle[i][j]=UNASSIGNED;

}

}

i=0;

srand(time(0));

j=0;

while(i<9) *//this fills the first row with numbers randomly //form 1 to 9;*

{

num=rand()%9+1;

if(grid1.NoConflict(grid1,j,i,num))

{

grid1.puzzle[j][i]=num;

i++;

}

}

grid1.srow=rand()%3;

grid1.scol=rand()%9;

grid1.SolveSudoku(grid1,grid1,0,dummy);

grid2.generate\_final(grid1,grid2,level);

return 0;

}

*/\*The following function prints the object passed. This has no relevance to the actual generation but is used during testing and can be used to check it the sudokus are generating correctly without graphics.\*/*

int Sudoku::print\_array(Sudoku &grid)

{

for(int i=0;i<9;i++)

{

for(int j=0;j<9;j++)

{

cout<<grid.puzzle[i][j]<<" ";

}

cout<<endl;

}

return 0;

}

*/\*the following function takes two objects of the sudoku class and copies the array of the second oject to that of the first.\*/*

int Sudoku::equal\_array(Sudoku &grid1,Sudoku &grid2)

{

for(int i=0;i<9;i++)

{

for(int j=0;j<9;j++)

{

grid1.puzzle[i][j]=grid2.puzzle[i][j];

}

}

return 0;

}

*/\*The following function takes two objects of the type Sudoku and checks if their member arrays are equal to each other.\*/*

bool Sudoku::check\_equal(Sudoku &grid1,Sudoku &grid2)

{

for(int i=0;i<9;i++)

{

for(int j=0;j<9;j++)

{

if(grid1.puzzle[i][j]!=grid2.puzzle[i][j])

{

return false;

}

}

}

return true;

}

*/\*The following function takes the same input as the generate function but it is required that you pass a solved/completed sudoku as the first parameter and a sudoku in which you require "level" number of numbers form random positions.\*/*

int Sudoku::generate\_final(Sudoku &grid1,Sudoku &grid2,int level)

{

int Pos[2][81];

int i,j,k=0,temp;

srand(time(0));

for(i=0;i<9;i++)

{

for(j=i\*9;j<i\*9+9;j++)

{

Pos[0][j]=i;

Pos[1][j]=k++;

}

k=0;

}

for(i=0;i<80;i++)

{

int r=i+(rand()%(81-i));

int temp = Pos[0][i]; Pos[0][i] = Pos[0][r]; Pos[0][r] = temp;

temp = Pos[1][i]; Pos[1][i] = Pos[1][r]; Pos[1][r] = temp;

}

i=0;

k=0;

j=0;

equal\_array(grid2,grid1);

while(i<level)

{

if(k==81)

{

break;

}

temp=grid2.puzzle[Pos[0][k]][Pos[1][k]];

grid2.puzzle[Pos[0][k]][Pos[1][k]]=UNASSIGNED;

if(grid2.check\_unique(grid1,grid2))

{

i++;

k++;

continue;

}

grid2.puzzle[Pos[0][k]][Pos[1][k]]=temp;

k++;

}

return 0;

}

*/\*The following function takes a sudoku with numbers removed and checks if it has a unique solution. It solves it with the c parameter of the function SolveSudoku set to one and then to 0 if in both cases count is 1 then the sudolu has a unique solution.\*/*

bool Sudoku::check\_unique(Sudoku &grid1,Sudoku &grid2)

{

int i=0,k=0;

Sudoku grid;

int count1=0,count2=0;

grid.equal\_array(grid,grid2);

grid.scol=rand()%9;

grid.srow=rand()%9;

grid.SolveSudoku(grid,grid1,1,count1);

grid.equal\_array(grid,grid2);

grid.scol=rand()%9;

grid.srow=rand()%9;

grid.SolveSudoku(grid,grid1,1,count2);

if(count1==1&&count2==1)

{

return true;

}

return false;

}

#endif // GENERATE\_H\_INCLUDED

------------------------------------------**‘Sudoku\_windows.cpp’**---------------------------------------

**Header files used:**

**position.h**-contains the functions related to position manipulation.

**Ezwin.h**-makes the graphic functions accessible

**Bitmap.h**-contains the functions used for manipulating bitmaps

**assert.h**-the assert library provides a pre-processor macros that is useful for debugging

**generate.h**-used for generation of Sudoku.

**String**- contains functions used for string manipulation

**Code and Explanation-**

#include "position.h"

#include "ezwin.h"

#include "bitmap.h"

#include<assert.h>

#include "Generate.h"

#include<string>

SimpleWindow w1("Sudoku Mania",14.0,12.0,Position(5.0,1.0));

BitMap Bmp1(w1);

BitMap Bmp2(w1);

BitMap Bmp3(w1);

BitMap Bmp4(w1);

/*/Need for remembering the status of each bitmap*

enum BmpStatus{Drawn,Erased};

BmpStatus Bmp1Status=Erased;

BmpStatus Bmp2Status=Erased;

BmpStatus Bmp3Status=Erased;

BmpStatus Bmp4Status=Erased;

Position Centre;

Position BmpCorner;

Position Upperleft;

Position Lowerleft;

Position VarSqre;

Position CentreUpperleftSqre;

Position CentreLowerleftSqre;

double CentreX;

double CentreY;

double CursorX;

double CursorY;

int t;

int value;

int level;

int m;

int n;

string conversion;

stringstream ss;

Sudoku Solved,Grid1,Grid2;

Position ChangeVarSquare(int x,int y*)//REturns a position type after receiving //two parameters*

{ *//The position in the sudoku grid is calculated //according to the array indices passed*

Position v=CentreUpperleftSqre +Position(1\*x,1\*y);

Upperleft = v + Position(-0.4,-0.4);

Lowerleft = v + Position(0.4,0.4);

return v;

}

Position ChangeLowerVarSqre(int x*)//Returns a position type after receving a //parameter*

*//The position in the input box is calculated according to the indices passed*

{

Position v= CentreLowerleftSqre + Position(x\*0.65,0);

return v;

}

int DisplaySudoku() *//Displays the internally generated sudoku for playing in the window at the beginning of the game*

{

for(m=0;m<9;m++)

{

for(n=0;n<9;n++)

{

VarSqre =ChangeVarSquare(m,n);

if(Grid1.puzzle[m][n]!=0)

{

char c=Grid1.puzzle[m][n] + 48;

char C[2];

C[0]=c;

C[1]='\0';

w1.RenderText(Upperleft,Lowerleft,C,Magenta,White);

}

}

}

return 0;

}

*/\*The return type integer function which is the parameter of predefined function "SetMouseClickCallback" of simple window class*

*It returns 1 on a sucessful mouse click and zero otherwise.It notes the position of the mouse click and judges what is to be done according to position of the click with respect to the bitmap appearing in the window.We have taken some enum variable which remebers whether a bitmap is drawn in the window or it is erased and other is drawn.All the enum objects carrying the bitmap status are initially initialised to zero except the first bitmap which represents the main menu window.According to the click position it decides whether to call the next bitmap.In this case it initialises the first bitmapstatus to erased and erases it and sets the called bitmap status to drawn and draws it without changing the window in other words all the bitmaps are linked with the same window.\*/*

int Mouse(const Position &p)

{

CursorX=p.GetXDistance();

CursorY=p.GetYDistance();

if(Bmp1Status==Drawn)

{

if(CursorX>=(CentreX-2.1)&&CursorX<=(CentreX+2.5))

{

if(CursorY>=(CentreY-2.8)&&CursorY<=(CentreY-1.2))

{

Bmp2Status=Drawn;

Bmp1Status=Erased; *//The button for New Game*

Bmp2.SetPosition(BmpCorner);

Bmp1.Erase();

Bmp2.Draw();

return 1;

}

else if(CursorY>=(CentreY)&&CursorY<=(CentreY+1.5))

{

Bmp1Status=Erased;

Bmp1.Erase();

w1.Close(); //The button Quit

return 0;

}

}

}

if(Bmp2Status==Drawn)

{

if((CursorX>=CentreX-2.1) && (CursorX<=CentreX+2.5))

{

if ((CursorY>=CentreY-2.8) && (CursorY<=CentreY-1.2)) *// for easy button*

{

level=81-(23+rand()%2);

VarSqre=ChangeLowerVarSqre(2);

Upperleft = VarSqre + Position(-3,-1);

Lowerleft = VarSqre + Position(3,1);

Grid1.generate(Solved,Grid1,level);

Grid1.equal\_array(Grid2,Grid1);

Bmp2.Erase();

Bmp2Status=Erased;

Bmp3.SetPosition(BmpCorner);

Bmp3.Draw();

Bmp3Status=Drawn;

DisplaySudoku();

return 1;

}

if ((CursorY>=CentreY-0.8) && (CursorY<=CentreY+0.8)) *// for medium button*

{

level=81-(25+rand()%3);

Grid1.generate(Solved,Grid1,level);

Grid1.equal\_array(Grid2,Grid1);

Bmp2.Erase();

Bmp2Status=Erased;

Bmp3.SetPosition(BmpCorner);

Bmp3.Draw();

Bmp3Status=Drawn;

DisplaySudoku();

return 1;

}

if ((CursorY>=CentreY+1.2) && (CursorY<=CentreY+2.9)) *// For hard button*

{

level=81-(28+rand()%3);

Grid1.generate(Solved,Grid1,level);

Grid1.equal\_array(Grid2,Grid1);

Bmp2.Erase();

Bmp2Status=Erased;

Bmp3.SetPosition(BmpCorner);

Bmp3.Draw();

Bmp3Status=Drawn;

DisplaySudoku();

return 1;

}

return 1;

}

}

if(Bmp3Status==Drawn)

{

if((CursorX>=CentreX-4.5) && (CursorX<=CentreX-2.9))

{

if((CursorY>=CentreY-4) &&(CursorY<=CentreY-2.4))*// verify button*

{

if(Grid1.check\_equal(Solved,Grid2))

{

w1.Message("You solved Correctly");

}

else

{

w1.Message("You solved Incorrectly");

}

}

if((CursorY>=CentreY-0.8) &&(CursorY<=CentreY+0.8))*// solved button*

{

Bmp3.Erase();

Bmp3Status=Erased;

Bmp4.SetPosition(BmpCorner);

Bmp4.Draw();

Bmp4Status=Drawn;

for(m=0;m<9;m++)

{

for(n=0;n<9;n++)

{

VarSqre = ChangeVarSquare(m,n);

char c=Solved.puzzle[m][n]+48;

//ss<<c;

//ss>>conversion;

char C[2];

C[0]=c;

C[1]='\0';

if(Grid1.puzzle[m][n]!=0)

{

w1.RenderText(Upperleft,Lowerleft,C,Magenta,White);

}

else

{

w1.RenderText(Upperleft,Lowerleft,C,Black,White);

}

}

}

return 1;

}

if((CursorY>=CentreY+2.4) &&(CursorY<=CentreY+4)) *// Main Menu button*

{

Bmp3.Erase();

Bmp3Status=Erased;

Bmp1.SetPosition(BmpCorner);

Bmp1.Draw();

Bmp1Status=Drawn;

return 1;

}

}

for(t=0;t<9;t++)

{

VarSqre = ChangeLowerVarSqre(t);

if(CursorX>=(VarSqre.GetXDistance()-0.28)&&CursorX<=(VarSqre.GetXDistance() +0.28))

{

if(CursorY>=(VarSqre.GetYDistance()-0.27)&&CursorY<=(VarSqre.GetYDistance() +0.27))

{

value = t + 1;

return 1;

}

}

}

for(m=0;m<9;m++)

{

for(n=0;n<9;n++)

{

VarSqre =ChangeVarSquare(m,n);

if(CursorX>=(VarSqre.GetXDistance()-0.77)&&CursorX<=(VarSqre.GetXDistance()+0.77))

{

if(CursorY>=(VarSqre.GetYDistance()-0.77)&&CursorY<=(VarSqre.GetYDistance()+0.77))

{

if(Grid1.puzzle[m][n]!=0)

{

return 1;

}

Grid2.puzzle[m][n]=value;

char c=value + 48;

//ss<<c;

//s>>conversion;

char C[2];

C[0]=c;

C[1]='\0';

w1.RenderText(Upperleft,Lowerleft,C,Black,White);

return 1;

}

}

}

}

}

if(Bmp4Status==Drawn)

{

if(CursorX>=(CentreX - 4.5)&&CursorX<=(CentreX-2.9))

{

if(CursorY>=(CentreY+2.4)&&CursorY<=(CentreY+4))

{

Bmp4Status=Erased;

Bmp1.SetPosition(BmpCorner);

Bmp1Status=Drawn;

Bmp4.Erase();

Bmp1.Draw();

return 1;

}

}

}

return 1;

}

int ApiMain()

{

w1.Open();

//assert(w1.GetStatus() == WindowOpen);

Centre = w1.GetCenter() + Position(0,-0.5);

BmpCorner = Centre + Position(-5.0,-4.0);

CentreUpperleftSqre = Centre + Position(-2,-3.55);

CentreLowerleftSqre = Centre + Position(-2.1,5.7);

CentreX = Centre.GetXDistance();

CentreY = Centre.GetYDistance();

Bmp1.Load("1.xpm");

assert(Bmp1.GetStatus()==BitMapOkay);

Bmp2.Load("2.xpm");

//assert(Bmp2.GetStatus()==BitMapOkay);

Bmp3.Load("3.xpm");

//assert(Bmp3.GetStatus()==BitMapOkay);

Bmp4.Load("4.xpm");

//assert(Bmp4.GetStatus()==BitMapOkay);

Bmp1.SetPosition(BmpCorner);

Bmp1.Draw();

Bmp1Status=Drawn;

w1.SetMouseClickCallback(Mouse);

return 0;

}

**Scope:**

Further addition of time tracking of the game and scoring is also possible.

Instead of bitmaps we could use renderLine and renderRectangle functions to make the program resolution independent.

**Limitations:**

The screen freezes for few seconds while the Sudoku puzzle generates on the window where hardness is selected.

The program is resolution dependent; works well in 1280×1024 resolution of the screen display.