

fantasy19 / Derp

Issues

Pull requests

Actions

Projects

• • •

e681c03cbb

• • •

Derp / 280 / assignment03-Sudoku / Sudoku.cpp

fantasy19 No commit message



1 contributor

Blame



258 lines (214 sloc) | 7.12 KB

```

1  /*****
2  /*!
3  \file   Sudoku.cpp
4  \author Ang Cheng Yong
5  \par    email: a.chengyong\@digipen.edu
6  \par    DigiPen login: a.chengyong
7  \par    Course: CS280
8  \par    Programming Assignment #3
9  \date   25/10/2016
10 \brief
11 This file contains the implementation for BList.
12 */
13 /*****
14
15 #include "Sudoku.h"
16 #include <iostream>
17
18 /*****
19 /*!
20 \fn Sudoku::Sudoku(int basesize, SymbolType stype , CALLBACK callback )
21 \brief
22 constructor for sudoku and sets choices of board values
23 \return
24 None
25 */
26 /*****
27
28 Sudoku::Sudoku(int basesize, SymbolType stype , CALLBACK callback ) :board(0), cb(callback) {
29
30     // setting statistics

```

```

31     sStats.basesize = basesize;
32     sStats.backtracks =
33     sStats.moves =
34     sStats.placed = 0;
35
36     // setting values for sudoku puzzle
37     first = (stype) ? 'A' : '1';
38     width = basesize*basesize;
39
40     last = static_cast<char>(first + static_cast<char>( width-1 ));
41
42 }
43
44 /*****
45  *!
46  \fn Sudoku::~Sudoku()
47  \brief
48  destructor for sudoku, free up the board.
49  \return
50  None
51  */
52 /*****
53
54  Sudoku::~Sudoku() { delete[] board; }
55
56 /*****
57  *!
58  \fn Sudoku::SudokuStats Sudoku::GetStats() const
59  \brief
60  Gettor for sudoku statistics.
61  \return
62  data structure for statistics for sudoku board.
63  */
64 /*****
65
66  Sudoku::SudokuStats Sudoku::GetStats() const {
67      return sStats;
68  }
69
70 /*****
71  *!
72  \fn const char * Sudoku::GetBoard() const
73  \brief
74  Gettor for sudoku board
75  \return
76  array of values within the board.
77  */
78 /*****
79
80  const char * Sudoku::GetBoard() const {
81      return board;
82  }

```

```

83
84 /*****
85  */
86 \fn void Sudoku::SetupBoard(const char *values, size_t size)
87 \brief
88 Setting up the board with the specified board values and board size
89 \return
90 None
91 */
92 /*****
93
94 void Sudoku::SetupBoard(const char *values, size_t size) {
95
96     board = new char[size];
97
98     //setting board values
99     for (size_t i = 0; i < size; ++i) {
100         board[i] = (values[i] == '.') ? EMPTY_CHAR : values[i];
101     }
102 }
103
104 /*****
105  */
106 \fn bool Sudoku::Solve()
107 \brief
108 Attempts to solve the sudoku board
109 \return
110 True if solvable, false if not
111 */
112 /*****
113
114 bool Sudoku::Solve() {
115
116     cb(*this, board, MSG_STARTING, sStats.moves, sStats.basesize, 0, 0);
117
118     size_t init_val = 0;
119     if (place_value(init_val)) {
120         cb(*this, board, MSG_FINISHED_OK, sStats.moves, sStats.basesize, 0, 0);
121         return true;
122     }
123     else {
124         cb(*this, board, MSG_FINISHED_FAIL, sStats.moves, sStats.basesize, 0, 0);
125         return false;
126     }
127 }
128
129 /*****
130  */
131 \fn bool Sudoku::place_value(size_t place)
132 \brief
133 recursive completes the board by placing values cell after cell or
134 decides if the board is unsolvable

```

```

135 \param place
136 index in the array representing the board to put value in
137 \return
138 True if solvable, false if not
139 */
140 /*****
141
142 bool Sudoku::place_value(size_t place) {
143
144     if (place == (width*width))
145         return true;
146
147     if (board[place] != EMPTY_CHAR)
148         return place_value(place + 1);
149
150     for (char val = first; val <= last; ++val) {
151         bool abort = cb(*this, board, MSG_ABORT_CHECK, sStats.moves, sStats.basesize, s
152
153         if (abort)
154             return false;
155
156         //placing a value
157         board[place] = val;
158         ++moves_;
159         ++sStats.moves;
160         ++sStats.placed;
161
162         cb(*this, board, MSG_PLACING, sStats.moves, sStats.basesize, static_cast<unsig
163         // see whether value is valid
164         if (ConflictCheck(place, board[place])) {
165
166             if (place == (width*width) - 1) // stop checking if we're done
167                 return true;
168             else {
169                 if (place_value(place + 1)) // continue checking if available
170                     return true;
171
172                 if (abort)
173                     return false;
174             }
175
176         }
177     }
178     else { // replace value if value is invalid and there are still values availabl
179         board[place] = EMPTY_CHAR;
180         --sStats.placed;
181         cb(*this, board, MSG_REMOVING, sStats.moves, sStats.basesize, static_ca
182     }
183 }
184 // all values tried, going back to previous cell to change value to try again
185 board[place] = EMPTY_CHAR;
186 ++sStats.backtracks;

```

```
187         --sStats.placed;
188     --moves_;
189     return false;
190 }
191
192 /*****
193  *!
194  \fn bool Sudoku::ConflictCheck(size_t place, char val)
195  \brief
196  test a value if its valid by checking for conflict in row,
197  column and the box its in.
198  \param place
199  index in the array representing the board to put value in
200  \param val
201  value to be tested
202  \return
203  true if value is valid, if not false
204  */
205 /*****
206
207 bool Sudoku::ConflictCheck(size_t place, char val) {
208     bool maintain = true;
209
210     size_t startrow = place - place%width;
211     size_t endrow = startrow + width;
212
213     //row check
214     for (size_t i = startrow; i < endrow; ++i) {
215         if (board[i] == val && i != place) {
216             maintain = false;
217             break;
218         }
219     }
220
221     size_t startcol = place % width;
222     size_t endcol = startcol + (width - 1)*width;
223
224     //column check
225     for (size_t i = startcol; i <= endcol; i += width) {
226         if (board[i] == val && i != place) {
227             maintain = false;
228             break;
229         }
230     }
231
232     // setting up start and end of value for box check
233     // decrementing the row aspect to correct starting position
234     size_t boxstart = place - (place % sStats.basesize);
235     size_t rowoffset = place / width;
236
237     // decrementing the column aspect to correct starting position
238     boxstart -= (rowoffset % sStats.basesize) * width;
```

```
239
240     // end value is the start of last row of box to be checked
241     size_t boxend = boxstart + (sStats.basesize - 1) * width;
242
243     // box check
244     for (size_t i = boxstart; i <= boxend; i += width) {
245         bool brake = true;
246         for (size_t j = 0; j <= sStats.basesize - 1; ++j) {
247             if (board[i + j] == val && (i + j) != place) {
248                 brake =
249                     maintain = false;
250                 break;
251             }
252         }
253         if (!brake) break;
254     }
255
256     // final return value after passing through checks
257     return maintain;
258 }
```