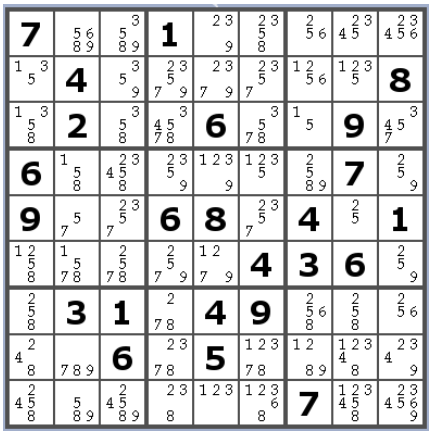
A6: Sudokus automatisch lösen



Bei einem Sudoku-Rätsel sucht man für jedes freie Feld eine Ziffer zwischen 1 und 9 so, dass folgende Regeln erfüllt sind: In jeder Zeile, in jeder Spalte und in jedem dick umrandeten Teilquadrat kommt jede der Ziffern von 1 bis 9 genau einmal vor. Wir wollen in dieser Aufgabe einen Sudoku-Löser (Klasse `SudokuSolver`) programmieren, der zum Lösen Hilfszahlen (Klasse `possibleDigits`) benutzt. Hilfszahlen werden in freie Felder eingetragen (siehe Beispiel oben) und zeigen an, welche Zahlen theoretisch in diesem Feld noch möglich sind, da sie weder in der jeweiligen Zeile, Spalte bzw. im jeweiligen Quadrat vorkommen. Zur Beschreibung der Lösungsstrategie studieren Sie bitte die Vorlesungsfolien aus Termin 13.

Aufgaben

1) Wir stellen Ihnen für diese Aufgabe zwei Dateien `sudoku.h` und `sudoku.cpp` zur Verfügung. Diese enthalten zum Einen die fertige Klasse `Sudoku` aus der letzten Aufgabe A5, sowie zum Anderen zwei neue Klassen `SudokuSolver` und `possibleDigits`, die im Folgenden von Ihnen mit Leben gefüllt werden sollen.

   Legen Sie ein neues Projekt in `Code::Blocks` an und kopieren Sie die beiden Dateien `sudoku.h` und `sudoku.cpp` in denselben Ordner, in dem auch das neue `main.cpp` liegt. Fügen Sie Ihrem neuen Projekt die beiden Dateien `sudoku.h/cpp` hinzu. Binden Sie in `main.cpp` den Header mit `#include "sudoku.h"` ein. Ersetzen Sie das Hauptprogramm `int main()` durch Ihr Hauptprogramm aus A5 und testen Sie, dass alles ordnungsgemäß kompiliert und funktioniert.

2) Wir beginnen mit der Klasse `possibleDigits` für die Hilfszahlen, die Sie in den Dateien `sudoku.h/cpp` finden. Als Attribut wird der Vektor

   ```
   vector<bool> possible;
   ```

   verwendet, der in seinen 9 Einträgen speichert, ob die jeweilige Ziffer möglich ist (`true`) oder nicht (`false`). Ergänzen Sie den Konstruktor sowie alle Methoden der Klassen.

3) Schreiben Sie den Logisch-Und-Operator

```
possibleDigits operator&& (const possibleDigits& a,
                           const possibleDigits& b)
```

der das logische Und `&&` eintragsweise durchführt und damit die Schnittmenge der möglichen Ziffern in `a` und `b` berechnet. Beachten Sie, dass diese Funktion als Friend vollen Zugriff auf die privaten Attribute von `possibleDigits` hat. Schreiben Sie zu Testzwecken ausserdem einen Ausgabeoperator für `possibleDigits`, der einfach die möglichen Ziffern ausgibt. Testen Sie die Klasse anschließend mit folgendem Code:

```
possibleDigits a(true), b(false);
a.disable(3); a.disable(4);
b.enable(1);  b.enable(3);
cout << "a:␣" << a << "\tb:␣" << b
     << "\na␣und␣b:␣" << (a && b) << endl;
```

4) Nun wenden wir uns der Klasse `SudokuSolver` zu. Diese enthält als private Attribute

```
vector<possibleDigits> pdRow, pdCol, pdSqr;
Sudoku&                Sudo;
```

Der Vektor `pdRow` mit 9 Einträgen speichert zu jeder Zeile die Hilfszahlen, d.h. welche Zahlen in der jeweiligen Zeile noch möglich sind. Entsprechendes gilt bei `pdCol` bzw. `pdSqr` für die Spalten bzw. Quadrate.

Beachten Sie, dass Sudo eine *Referenz* auf das im Konstruktor übergebene Sudoku ist. Das `SudokuSolver`-Objekt besitzt also keine eigene Kopie, sondern arbeitet direkt auf dem übergebenen Sudoku-Objekt.

Schreiben Sie den Konstruktor `SudokuSolver( Sudoku& S)`, der die Attribute wie oben beschrieben mithilfe des übergebenen Sudoku `s` initialisiert. Es macht Sinn, zur Anpassung der Hilfszahlen die (noch zu schreibende) Methode `setDigit` zu verwenden.

5) Schreiben Sie die Methode

```
void setDigit( int r, int c, int digit)
```

die die Ziffer `digit` in Zeile `r`, Spalte `c` sowie Quadrat `q` in den Hilfszahlen als nicht möglich markiert und außerdem die Ziffer in `Sudo` an der richtigen Stelle einträgt. Wie in der vorigen Aufgabe sind $r, c, q \in \{1, \ldots, 9\}$. Zur Berechnung des Quadrates q, das den Eintrag (r, c) enthält, stellen wir Ihnen die private, statische Methode[1] `static int getSqr( int r, int c)` zur Verfügung, die q zurückgibt.

6) Schreiben Sie die Methode

[1] Diese Methode ist privat, da sie nur für den internen Gebrauch bestimmt ist und nicht zur öffentlichen Schnittstelle gehören sollte, und statisch, da sie keines der Attribute benötigt.

```
void unsetDigit( int r, int c)
```

die die Wirkung von `setDigit( r, c, digit)` rückgängig macht und außerdem in `Sudo` an der richtigen Stelle eine 0 einträgt.

7) Falls das Feld (r, c) frei ist, soll die Methode

```
possibleDigits getPossible( int r, int c) const
```

die Hilfszahlen zu diesem Eintrag zurückgeben, also die Schnittmenge der Hilfszahlen zu Reihe r, Spalte c und zugehörigem Quadrat q. Ist das Feld nicht frei, so soll die leere Hilfszahlmenge zurückgegeben werden, da dann keine Zahlen möglich sind.

Lesen Sie zum Testen das Sudoku aus `Sudoku1.txt` ein und geben Sie für drei freie Felder sowie ein besetztes Feld die Hilfszahlen auf dem Bildschirm aus. Überprüfen Sie die Ergebnisse auf ihre Richtigkeit.

8) Implementieren Sie als nächstes die Methode

```
void getNextCell( int& r_min, int& c_min) const
```

die unter allen freien Feldern eines mit minimaler Anzahl möglicher Hilfszahlen findet und die gefundene Position in den Parametern $r_{\min}, c_{\min}$ zurückgibt. Hierbei ist natürlich die `possibleDigits`-Methode `getNumPossible` von unschätzbarem Wert.

9) Nun haben wir alle Bausteine beisammen, um die Methode

```
bool solve( int numEmpty)
```

zu schreiben. Schauen Sie sich noch einmal die Folien an, mit denen in der Vorlesung das Vorgehen beim Sudokulösen erläutert wurde:

1. Falls es kein freies Feld mehr gibt, brechen wir ab und melden Erfolg/Misserfolg zurück.
2. Wähle ein freies Feld (r, c) mit möglichst wenigen Hilfszahlen.
3. Für alle möglichen Ziffern d in Feld (r, c):
   a) Setze Ziffer d in Feld (r, c) und passe Hilfszahlen an.
   b) Wende (rekursiv) die Strategie `solve` auf das modifizierte Sudoku an. Wenn das erfolgreich war, sind wir *fertig* und melden den Erfolg zurück.
   c) Mache Schritt a) rückgängig.
4. Wenn wir hier landen, waren wir nicht erfolgreich. Melde den Misserfolg zurück.

Testen Sie Ihre Methode mit dem Sudoku aus `Sudoku1.txt`. Versuchen Sie danach auch die anderen zur Verfügung gestellten Sudokus zu lösen.

Erweiterungen

1*) Zählen Sie, wie oft `solve` gerufen wird, um ein Sudoku zu lösen.

2*) Ein n-Sudoku ($n \in \mathbb{N}$) ist ein quadratisches Gitter mit n^2 Zeilen, Spalten und Quadraten (also insgesamt n^4 Felder), in die die Zahlen 1 bis n^2 eingetragen werden sollen, so dass alle Zeilen/Spalten/Quadrate gültig sind, d.h. jede Zahl kommt genau einmal vor. Für $n = 3$ ergibt sich also das klassische Sudoku.

Verallgemeinern Sie Ihre Klasse `SudokuSolver` so, dass ein n-Sudoku[2] gelöst werden kann. Fügen Sie dazu eine `Sudoku`-Methode `int getDegree() const` hinzu, welche n zuruckgibt. Welche Methoden müssen wie angepasst werden?

3*) Schreiben Sie einen automatischen Sudoku-Generator.

Lernziele

- **Module:** größeres Projekt in mehrere Module aufspalten, für jedes Modul die Schnittstellen in der Header-Datei veröffentlichen, Implementierung in die Source-Datei auslagern
- **Friends:** von der Klasse auserwählte externe Funktionen oder andere Klassen, die Zugriff auf den privaten Bereich erhalten (Klasse `possibleDigits` erlaubt dies der Funktion `operator&&`)

[2] vgl. Zusatzaufgaben von A5.