Praktikumstermin Nr. 10, INF: Vererbung, Game of Life

(Pflicht-) Aufgabe INF-10.01: Vererbung, Klasse `MyFilledRectangle`

Kopieren Sie die Dateien `MyRectangle.h` und `MyRectangle.cpp` aus dem letzten Praktikum in ein neues Projekt.

Legen Sie außerdem neue Dateien `MyFilledRectangle.h` und `MyFilledRectangle.cpp` an und programmieren Sie dort eine Klasse `MyFilledRectangle`, die von `MyRectangle` erbt.

Die Klasse `MyFilledRectangle` besitze keine eigenen Attribute, sondern nutze die Attribute der Basisklasse `MyRectangle`. *Wie muss der Zugriffsschutz der Basisklassen-Attribute sein (bzw. gegenüber dem vorigen Praktikum geändert werden), damit diese Attribute in der abgeleiteten Klasse mitgenutzt werden können und trotzdem nicht von außen zugreifbar sind?*

Der Konstruktor von `MyFilledRectangle` habe die gleichen Parameter wie der Konstruktor der Basisklasse `MyRectangle` und initialisiere die Attribute durch Aufruf des Basisklassen-Konstruktors mit den Parametern des eigenen Aufrufs.

Die `draw()` Methode von `MyFilledRectangle` soll erst einmal mit Hilfe der `draw()` Methode von `MyRectangle` ein blaues Rechteck zeichnen.
Dann sollen mit Hilfe der in `CImgGIP05.h` realisierten Funktion …

```
gip_draw_line(unsigned int x1, unsigned int y1,
              unsigned int x2, unsigned int y2,
              gip_color color);
```

… lauter horizontale rote (`red`) Linien gezeichnet werden, von „ganz oben im Kästchen“ (linker bis rechter Rand) bis „ganz unten im Kästchen“ (linker bis rechter Rand), allerdings mit jeweils einem Abstand von 2 Pixeln zum Rand. D.h. das Rechteck wird durch lauter horizontale rote Linien ausgefüllt.
Ihr Code soll prüfen, ob das `MyFilledRectangle` auch wirklich hoch und breit genug ist, um einen Abstand von 2 Pixeln zum Rand zu erlauben. Falls nicht, so soll das „Füllen mit Linien“ nicht gemacht werden (und nur das `MyRectangle` gezeichnet werden).

Schreiben Sie ein Hauptprogramm analog zum Hauptprogramm des vorherigen Praktikums, welches alle 4 Sekunden zwei neue

`MyFilledRectangle` an zufälligen Positionen zeichnet und die Kollisionserkennung durchführt.

(Pflicht-) Aufgabe INF-10.02: Conway's *Game of Life*

Schreiben Sie unter Benutzung der *`CImg`* Library (mittels der Headerdatei `CImgGIP05.h`, die Sie schon aus einem vorherigen Praktikumsversuch kennen) ein C++ Programm, welches das Spiel *Game of Life* realisiert. Dieses stellt die zeitliche Entwicklung einer „Zellkolonie" dar.

https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Die Größe des Feldes sei als Konstante `grid_size` vorgegeben. Benutzen Sie ein zweidimensionales statisches Array zur Realisierung des Spielfeldes.

Das Spiel startet je nach Benutzereingabe entweder mit einer zufälligen Belegung des Feldes oder mit einer vorgegebenen Belegung. Benutzen Sie für die zufällige Belegung den Funktionsaufruf `gip_random(0,1);` um jeweils eine Zufallszahl „entweder 0 oder 1" zu erzielen.

Orientieren Sie sich an folgendem Programmgerüst (welches auch als Datei in Ilias vorliegt) für Ihre Lösung:

```
#include <iostream>
```

```
#define CIMGGIP_MAIN
#include "CImgGIP05.h"
using namespace std;
using namespace cimg_library;

const int grid_size = 10; // Anzahl an Kaestchen in x- und y-Richtung
const int box_size = 30;  // size der einzelnen Kaestchen (in Pixel)
const int border = 20;   // Rand links und oben bis zu den ersten Kaestchen (in Pixel)

// Prototyp der Funktionen zum Vorbelegen des Grids ...
void grid_init(bool grid[][grid_size]);

int main()
{
    bool grid[grid_size][grid_size] = { 0 };
    bool next_grid[grid_size][grid_size] = { 0 };

    // Erstes Grid vorbelegen ...
    grid_init(grid);

    while (gip_window_not_closed())
    {
        // Spielfeld anzeigen ...
        // gip_stop_updates(); // ... schaltet das Neuzeichnen nach
                               //     jeder Bildschirmänderung aus

        // TO DO

        // gip_start_updates(); // ... alle Bildschirmänderungen (auch die
                                //     nach dem gip_stop_updates() ) wieder anzeigen
        gip_sleep(3);

        // Berechne das naechste Spielfeld ...
        // Achtung; Für die Zelle (x,y) darf die Position (x,y) selbst *nicht*
        // mit in die Betrachtungen einbezogen werden.
        // Ausserdem darf bei zellen am rand nicht über den Rand hinausgegriffen
        // werden (diese Zellen haben entsprechend weniger Nachbarn) ...

        // TO DO


        // Kopiere das naechste Spielfeld in das aktuelle Spielfeld ...

        // TO DO

    }
    return 0;
}

void grid_init(bool grid[][grid_size])
{
    int eingabe = -1;
    do {
        cout << "Bitte waehlen Sie die Vorbelegung des Grids aus:" << endl
            << "0 - Zufall" << endl
            << "1 - Statisch" << endl
            << "2 - Blinker" << endl
            << "3 - Oktagon" << endl
```

```
        << "4 - Gleiter" << endl
        << "5 - Segler 1 (Light-Weight Spaceship)" << endl
        << "6 - Segler 2 (Middle-Weight Spaceship)" << endl
        << "? ";
    cin >> eingabe;
    cin.clear();
    cin.ignore(1000, '\n');
} while (eingabe < 0 || eingabe > 6);

if (eingabe == 0)
{
    // Erstes Grid vorbelegen (per Zufallszahlen) ...

    // TO DO

}
else if (eingabe == 1)
{
    const int pattern_size = 3;
    char pattern[pattern_size][pattern_size] =
    {
        { '.', '*', '.' },
        { '*', '.', '*' },
        { '.', '*', '.' },
    };
    for (int y = 0; y < pattern_size; y++)
        for (int x = 0; x < pattern_size; x++)
            if (pattern[y][x] == '*')
                grid[x][y+3] = true;
}
else if (eingabe == 2)
{
    const int pattern_size = 3;
    char pattern[pattern_size][pattern_size] =
    {
        { '.', '*', '.' },
        { '.', '*', '.' },
        { '.', '*', '.' },
    };
    for (int y = 0; y < pattern_size; y++)
        for (int x = 0; x < pattern_size; x++)
            if (pattern[y][x] == '*')
                grid[x][y+3] = true;
}
else if (eingabe == 3)
{
    const int pattern_size = 8;
    char pattern[pattern_size][pattern_size] =
    {
        { '.', '.', '.', '*', '*', '.', '.', '.' },
        { '.', '.', '*', '.', '.', '*', '.', '.' },
        { '.', '*', '.', '.', '.', '.', '*', '.' },
        { '*', '.', '.', '.', '.', '.', '.', '*' },
        { '*', '.', '.', '.', '.', '.', '.', '*' },
        { '.', '*', '.', '.', '.', '.', '*', '.' },
        { '.', '.', '*', '.', '.', '*', '.', '.' },
        { '.', '.', '.', '*', '*', '.', '.', '.' },
    };
```

```
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+1] = true;
    }
    else if (eingabe == 4)
    {
        const int pattern_size = 3;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '*', '.' },
            { '.', '.', '*' },
            { '*', '*', '*' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
    else if (eingabe == 5)
    {
        const int pattern_size = 5;
        char pattern[pattern_size][pattern_size] =
        {
            { '*', '.', '.', '*', '.' },
            { '.', '.', '.', '.', '*' },
            { '*', '.', '.', '.', '*' },
            { '.', '*', '*', '*', '*' },
            { '.', '.', '.', '.', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
    else if (eingabe == 6)
    {
        const int pattern_size = 6;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '*', '*', '*', '*', '*' },
            { '*', '.', '.', '.', '.', '*' },
            { '.', '.', '.', '.', '.', '*' },
            { '*', '.', '.', '.', '*', '.' },
            { '.', '.', '*', '.', '.', '.' },
            { '.', '.', '.', '.', '.', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
}
```

Eine Zelle wird in einem leeren Feld neu geboren, wenn im vorigen Grid drei der umgebenden Nachbarzellen belebt waren.

Nachbarzellen: Eine Zelle, die nicht am Rand des Spielfelds liegt, hat 8 Nachbarn: oben, unten, links, rechts und 4x diagonal. Die Zelle selbst zählt nicht zu ihren eigenen Nachbarn. Zellen am Rand des Spielfelds haben weniger Nachbarn.

Eine Zelle bleibt in einem Feld am Leben, wenn im vorigen Grid zwei oder drei der umgebenden Nachbarzellen belebt waren.

In allen anderen Fällen wird keine Zelle geboren bzw. eine dort lebende Zelle stirbt wegen zu vieler oder zu weniger Nachbarn, d.h. das Feld wird im nächsten Spielfeld unbewohnt sein.

Stellen Sie das Spielfeld über die Zellgenerationen hinweg graphisch dar.

Hinweis: Sie werden sehen, dass die Aktualisierungen des "Spielfelds" recht langsam "Kästchen für Kästchen" passieren … Das ist eine Konsequenz der GIP-spezifischen Änderungen der CImg Library (http://cimg.eu/), die ich vorgenommen habe: Ich "zwinge die Library", jede Graphik-Operation auch sofort auf dem Bildschirm sichtbar zu machen. Das bremst die Library sehr stark, hat aber den Vorteil, dass Sie im Debugger jeden Schritt des Programms einzeln nachverfolgen können und wirklich auf dem Bildschirm sehen, was ihre einzelnen Graphik-Operationen wie `gip_draw_rectangle()` *tun. Normalerweise wäre das nicht der Fall …*
Wenn Sie dann sicher sind, dass ihr Code wohl funktioniert, dann fügen Sie die im Programmrahmen noch auskommentierten Funktionsaufrufe `gip_stop_updates()` *und* `gip_start_updates()` *hinzu.*
`gip_stop_updates()` *"schaltet CImg in seinen üblichen Modus um": Graphik-Änderungen ("Updates") werden* ***nicht*** *mehr direkt angezeigt, sondern nur noch intern durchgeführt und gespeichert.*
Erst mit dem Aufruf von `gip_start_updates()` *wird die Library wieder in den "GIP Modus geschaltet": Alle "in der Zwischenzeit" intern gespeicherten Änderungen werden dann mit einem Schlag sichtbar gemacht und alle danach stattfindenden Änderungen werden dann auch wieder sofort sichtbar gemacht …*

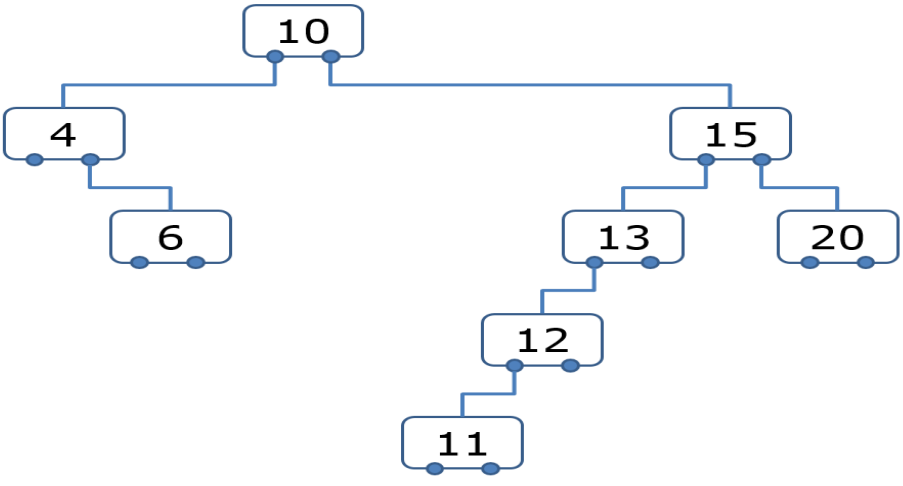
Testläufe: *(Animation der vorgegebenen Muster siehe Wikipedia Seite)*

```
Bitte waehlen Sie die Vorbelegung des Grids aus:
0 - Zufall
1 - Statisch
2 - Blinker
3 - Oktagon
4 - Gleiter
5 - Segler 1 (Light-Weight Spaceship)
6 - Segler 2 (Middle-Weight Spaceship)
?
```

Für das Aufgaben-Tutorium am Freitag 22.1.2021, als freiwillige Aufgabe:

(Freiwillige) Aufgabe INF-10.03: Dynamische Datenstruktur: Binärer Suchbaum (duplikatfrei) über `int` Werten

Ein *Binärer Suchbaum (ohne Duplikate) über `int` Werten* ist eine Datenstruktur, in der `int` Werte in den Knoten der Datenstruktur nach den im folgenden beschriebenen Regeln gespeichert werden.



Jeder Knoten der Datenstruktur speichert genau einen `int` Wert und besitzt genau einen *Elternknoten* (Ausnahme: *Wurzelknoten* des Baums, der keinen Elternknoten besitzt) und höchstens zwei *Kindknoten*.

Der *erste* in den Baum einzufügende `int` Wert wird im neu zu erzeugenden Wurzelknoten des Baums abgelegt.

Jeder weitere einzufügende `int` Wert wird nach folgendem Prinzip in den Baum eingefügt: Ausgehend vom Wurzelknoten wird der neue Wert mit dem im jeweiligen Knoten gespeicherten Wert verglichen.

1. Ist der neue Wert *gleich* dem Wert im Knoten, so wird der neue Wert nicht erneut in den Baum eingefügt (*duplikatfreier* Baum).

2. Ist der neue Wert *kleiner* dem Wert im Knoten und besitzt der Knoten *keinen* linken Kindknoten, so wird der neue Wert in einen neu zu erzeugenden linken Kindknoten eingefügt.

3. Ist der neue Wert *kleiner* dem Wert im Knoten und besitzt der Knoten einen linken Kindknoten, so wird die Prüfung ab Fall 1. für den linken Kindknoten erneut vorgenommen.

   Fälle 4. und 5. sind analog zu 2. und 3.:

4. Ist der neue Wert *größer* dem Wert im Knoten und besitzt der Knoten *keinen* rechten Kindknoten, so wird der neue Wert in einen neu zu erzeugenden rechten Kindknoten eingefügt.
5. Ist der neue Wert *größer* dem Wert im Knoten und besitzt der Knoten einen rechten Kindknoten, so wird die Prüfung ab Fall 1. für den rechten Kindknoten erneut vorgenommen.

Programmieren Sie eine geeignete Datenstruktur `BaumKnoten` sowie Funktionen `einfuegen()` und `ausgeben()`, um einen duplikatfreien Binärbaum über `int` Werten gemäß den Testläufen zu realisieren.

Welche Komponenten die Datenstruktur `BaumKnoten` enthalten muss, sollen Sie anhand der Notwendigkeiten der Funktionen `einfuegen()` und `ausgeben()` entscheiden.

Verwenden Sie keine globalen oder `static` Variablen.

Deklarieren Sie die Datenstruktur `BaumKnoten` sowie die Funktions-prototypen für `einfuegen()` und `ausgeben()` in einer Headerdatei `binaerer_suchbaum.h`, und innerhalb eines Namespaces `suchbaum`.

Implementieren Sie die Funktionen `einfuegen()` und `ausgeben()` in einer Datei `binaerer_suchbaum.cpp`. Sie können gerne zusätzliche Hilfsfunktionen definieren, dann aber innerhalb des Namespaces `suchbaum`.
Die Ausgabefunktion `ausgeben()` rückt die Knotenwerte entsprechend ihrer Tiefe im Baum (d.h. Abstand vom Wurzelknoten) ein, mit zwei Pluszeichen pro Tiefenstufe. Der Baum ist bei der textuellen Ausgabe „um 90 Grad gegen den Uhrzeigersinn gedreht" im Vergleich zur Diagrammdarstellung.
D.h. zu einem Baumknoten wird erst der rechte Teilbaum ausgegeben, dann der Wert des Knotens selbst, dann der linke Teilbaum.
Wegen der Selbstähnlichkeit (Teilbaum sieht von der Struktur aus wie der gesamte Baum): Realisieren Sie die Ausgabe über eine rekursive Funktion

```
void suchbaum::knoten_ausgeben(BaumKnoten* knoten, int tiefe);
```

… die aus der Funktion `ausgeben()` aufgerufen wird und genau das obige Ausgabeprinzip umsetzt (lassen Sie sich von der „Türme von Hanoi" Funktion inspirieren, falls nötig…).

Implementieren Sie die `main()` Funktion in einer Datei `suchbaum_main.cpp`.

Testläufe (Benutzereingaben sind unterstrichen):

```
Leerer Baum.
Naechster Wert (99 beendet): ? 10
Naechster Wert (99 beendet): ? 4
Naechster Wert (99 beendet): ? 6
Naechster Wert (99 beendet): ? 15
Naechster Wert (99 beendet): ? 13
Naechster Wert (99 beendet): ? 12
Naechster Wert (99 beendet): ? 15
Naechster Wert (99 beendet): ? 20
Naechster Wert (99 beendet): ? 11
Naechster Wert (99 beendet): ? 15
Naechster Wert (99 beendet): ? 99
++++20
++15
++++13
++++++12
++++++++11
10
++++6
++4
Drücken Sie eine beliebige Taste . . .
```

```
Leerer Baum.
Naechster Wert (99 beendet): ? 3
Naechster Wert (99 beendet): ? 3
Naechster Wert (99 beendet): ? 3
Naechster Wert (99 beendet): ? 2
Naechster Wert (99 beendet): ? 99
3
++2
Drücken Sie eine beliebige Taste . . .
```

```
Leerer Baum.
Naechster Wert (99 beendet): ? 99
Leerer Baum.
```

```
Drücken Sie eine beliebige Taste . . .
```