Programmeertechnieken 2017
Opdracht 2: Spreadsheet

Deadline deel A: Vrijdag 24 maart, voor het einde van de dag.
Deadline deel B: Vrijdag 14 april, voor het einde van de dag.

1 Introductie

De opdracht laat zich eenvoudig omschrijven: schrijf een spreadsheetprogramma met een tekstgebaseerde user interface, welke werkt in een terminal op UNIX-achtige systeem (Linux, macOS). Een voorbeeld van het programma is te zien in Figuur 1. Om te "tekenen" in de terminal zullen we gebruik maken van de library *ncurses*. Het is de bedoeling dat de spreadsheet wordt ontwikkeld in de taal C++ en dat er uitgebreid gebruik wordt gemaakt van klassen.

Het project heeft verschillende leerdoelen: het belangrijkste leerdoel is het leren opbouwen van grotere programma's en het uitgebreid gebruik maken van principes van het object geörienteerd programmeren. Omdat er in teams van drie personen mag worden gewerkt, is een ander belangrijk aspect het leren samenwerken aan een "code base" en het verdelen van de taken. Tenslotte komen ook de vaardigheden omtrent Makefiles en het gebruiken van andere bibliotheken (in dit geval *ncurses*) weer aan bod.

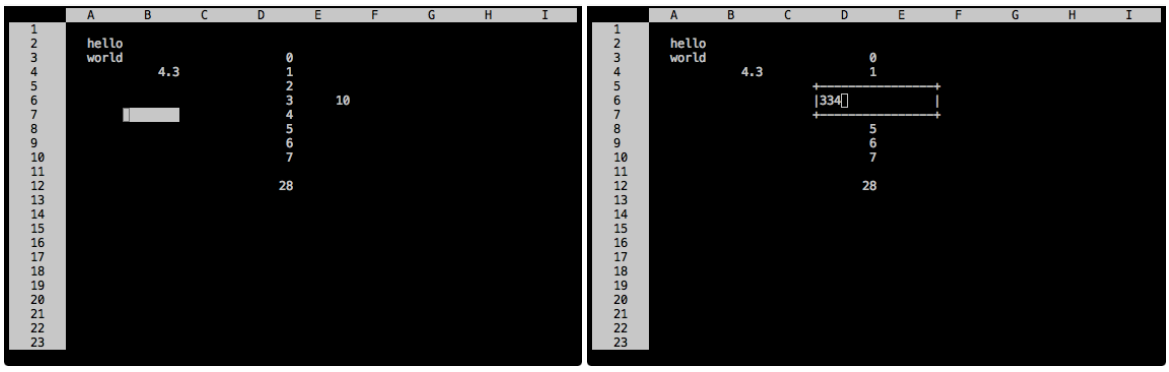
Dit is een behoorlijk groot programmeerproject. Er mag daarom worden gewerkt in teams van drie personen. Het is belangrijk om het werk te verdelen: spreek onderling goed af wie welk component voor zijn/haar rekening neemt! Overleg regelmatig over de interfaces (klassen en methoden) tussen de verschillende componenten, zodat de componenten goed op elkaar worden afgestemd. Als je twijfelt over hoe een interface goed te definiëren, vraag gerust de assistenten om advies! Om jullie op weg te helpen geven we verderop in deze omschrijving uitgebreide adviezen over hoe het project aan te pakken en een schets van een mogelijk ontwerp.

Deze opdracht heeft een A en een B deadline. Voor beide deadlines dient het project te worden ingeleverd: voor deadline A alleen een eerste deel (zie ook hieronder), voor deadline B het volledige project. Alleen het werk dat is ingediend voor de B deadline zal worden becijferd. Het is ons doel om al het werk dat *voor of op* deadline A is ingeleverd binnen een week te voorzien van kort commentaar. Deze beknopte feedback kan dan worden gebruikt om het eindproject te verbeteren voordat het wordt ingeleverd voor deadline B.

2 Functionaliteiten

Wat wordt er verwacht van jullie spreadsheetprogramma? We willen graag de volgende functionaliteiten terugzien in het eindproduct dat wordt ingeleverd voor deadline B:

- Een goed georganiseerde "Sheet" datastructuur.
  - De datastructuur moet worden geïnitialiseerd met een bepaalde grootte (aantel rijen, kolommen), welke als argumenten aan de constructor worden meegegeven.



Figuur 1: Twee screenshots van het spreadsheetprogramma.

  - Daarnaast moet het mogelijk zijn om het aantal rijen en kolommen op een later tijdstip te vergroten, bijvoorbeeld via een methode `ensureSize(n_rows, n_cols)`.
  - De mogelijkheid om cellen te maken van verschillende typen: integer, floating-point, string en formule. Dit moet zijn geïmplementeerd met behulp van templates.
  - We verwachten een zelfgeschreven iterator klasse voor het aflopen van reeksen van cellen.

- De mogelijkheid om gebruik te maken van geaggregeerde formules zoals som, gemiddelde en tellen. Dus we verwachten dat `=SUM(A1:A5)` werkt, maar `=A1+B4` niet. Formules worden geïmplementeerd door gebruik te maken van de zelfgeschreven iterator om de cellen in een reeks af te lopen (zie ook Sectie 6.6).

- In de formules moet het mogelijk zijn een reeks van cellen aan te geven met de dubbele-punt-notatie. De reeks mag zowel een deelrij, deelkolom of een blok van cellen zijn.

- Een tekst-gebaseerde user interface met de volgende mogelijkheden:
  - Vaste grootte van 24 regels en 80 kolommen. Scrollen hoeft niet te worden geïmplementeerd.
  - Duidelijke rij- en kolomheaders.
  - Een opgelichte "cursor" die met de pijltjestoetsen kan worden verplaatst.
  - Als er op "enter" wordt gedrukt zal de cel onder de cursor worden bewerkt. Hiervoor wordt een klein "popup window" gemaakt, waarin de huidige waarde van de cel kan worden aangepast. Met "enter" wordt de nieuwe waarde geaccepteerd.
  - Als er op "backspace" wordt gedrukt zal de waarde van de cel onder de cursor worden gewist.
  - Als er op "q" wordt gedrukt zal het programma worden afgesloten.

- Een verandering van de waarde van een cel zal leiden tot de herberekening van formules. Wanneer dit goed wordt ingericht, zullen ook "geneste" formules goed functioneren.

- Uiteraard is de source code voorzien van een Makefile en worden klassen zoveel mogelijk geïmplementeerd in afzonderlijke bestanden (.cpp/.cc en .h-bestanden).

Voor een uitwerking van de basisfunctionaliteiten kan maximaal het cijfer 9 worden behaald. We verwachten voor een 10 ietsje meer. Als het basisdeel voldoende is, kan het laatste punt worden gehaald door één van de volgende twee functionaliteiten te implementeren:

- De mogelijkheid om de spreadsheet op te slaan naar een bestand en uit een bestand in te lezen. In Sectie 6.7 geven we een aantal hints hoe het opslaan en lezen van de sheet mooi kan worden ingericht. Wat betreft de user interface: houd het simpel! Bijvoorbeeld: als er een bestandsnaam is opgegeven bij het opstarten van het programma, probeer dan de spreadsheet uit dit bestand in te lezen. In het geval het bestand nog niet bestaat, geef geen error en onthoud de bestandsnaam. Als er in het programma op de toets "s" wordt gedrukt, ga dan na of er een bestandsnaam was opgegeven. Zo ja, schrijf de huidige spreadsheet naar dit bestand.

- Volledige ondersteuning voor niet-geaggregeerde formules zoals `=A1+B3+B4`, `=(A4*2)+4`, `=(A4*2)+(B3*3)`, `=SUM(A1:B1)/10`, enzovoort. Hier is dus een uitgebreide parser benodigd!

Een referentie-implementatie van de basisfunctionaliteiten en het lezen/schrijven van bestanden kan worden getest binnen het "pt2017" environment met de commando's:

```
source /vol/share/groups/liacs/scratch/pt2017/pt2017.env
/vol/share/groups/liacs/scratch/pt2017/bin/tulip123
```

3 Inleveren

Er mag worden gewerkt in teams van drie personen. Verdeel onderling de taken en overleg regelmatig! De samenstelling van het team dat deel B inlevert moet hetzelfde zijn als het team dat deel A heeft ingeleverd (met andere woorden: gedurende opdracht 2 mag er niet van team worden gewisseld). De wijze van inleveren is hetzelfde voor het A als B deel en we verwachten het volgende wordt ingeleverd:

- C++ source code.
- Makefile.
- README bestand met daarin instructies voor het compileren van het programma en waarin is vermeld met welke compilers het programma is getest.

Zorg ervoor dat alle bestanden die worden ingeleverd zijn voorzien van namen en studentnummers! Plaats alle bestanden om in te leveren in een aparte directory (bijv., `opdracht2`) en maak een "gzipped tar" bestand:

```
tar -czvf opdracht2-sXXXXXXX-sYYYYYYY-sZZZZZZ.tar.gz opdracht2/
```

Vul op de plek van `XXXXXXX`, `YYYYYYY` en `ZZZZZZZ` de bijbehorende studentnummers in. De inzendingen kunnen worden verzonden per e-mail naar `pt2017 (at) handin.liacs.nl` met als onderwerp "PT Opdracht 2A" voor deel A, 'PT Opdracht 2B" voor deel B.

Belangrijk: Test je programma in ieder geval op de Linux machines in de computerzalen! Dit is voornamelijk belangrijk voor studenten die werken op een macOS computer, zorg ervoor dat je programma ook compileert op een computer in de computerzaal.

Voor deel A verwachten we dat de datastructuur, celadresparsing, reeks-iterator en formule parsing zijn geïmplementeerd en dat er code aanwezig is om deze functionaliteiten te testen.

De **normering** voor deel B is als volgt: kwaliteit code/commentaar/modulariteit (2 punten), werking datastructuur & reeksen (ranges) (2.5 punten), werking formules (2 punten), werking user interface (2.5 punten), extra functionaliteit (1 punt – alleen als de basisopdracht voldoende is). Dus zonder de extra functionaliteit kan voor een correct werkend programma dat de hierboven omschreven basisfunctionaliteiten bevat een 9 worden gehaald.

4 Compiler

Gebruik een zo recent mogelijke C++ compiler, die ook de nieuwe standaard C++11 (en eventueel C++14) ondersteunt. Op de Linux machines in de computerzalen is g++ 5.3.0 beschikbaar. We hebben ervoor gezorgd dat het "pt2017" environment standaard deze compileer selecteert. Draai dus altijd het commando

```
source /vol/share/groups/liacs/scratch/pt2017/pt2017.env
```

alvorens te beginnen. Voeg de optie `-std=c++14` toe aan het compilercommando zodat ondersteuning voor de nieuwe C++ standaard wordt aangezet.

Belangrijk: als je werkt op een macOS computer, dan maak je gebruik van de 'Clang' compiler in plaats van gcc. Zorg er in ieder geval voor dat je ook je programma probeert te compileren op de Linux systemen op de universiteit voordat je de opdracht inlevert! Code die door 'Clang' wordt geaccepteerd, wordt soms niet door gcc geaccepteerd (en vice versa).

De *ncurses* bibliotheek is in principe standaard beschikbaar op alle Linux en macOS systemen. Gebruik `-lcurses` bij het linken. Zie ook Sectie A.2.

Het is toegestaan om gebruik te maken van STL en Boost[1]. Gebruik bijvoorbeeld het unit testing framework van Boost om unit tests te schrijven. In het "pt2017" environment staat een recente versie van Boost klaar. Voeg het volgende toe aan het compilercommando zodat de Boost headerfiles worden gevonden:

```
-I/vol/share/groups/liacs/scratch/pt2017/include
```

5 Aanpak

Je staat nu aan het begin van een groot programmeerproject. Stel daarom eerst een plan van aanpak op en verdeel de taken. Neem de onderstaande schets van het ontwerp door en teken vervolgens voor jezelf de interface van de verschillende componenten uit op een blaadje. Zodra het helder is hoe alles in elkaar steekt, begin dan door klassen en methoden uit te schrijven in headerfiles. Hierna kunnen in parallel (!) de implementaties voor deze klassen worden ingevuld. Blijf regelmatig in overleg over de interfaces, deze kunnen wijzigen! Merk op dat naargelang het project groeit het makkelijker wordt om er in parallel aan te werken, maak hier gebruik van.

Het is ook een goed idee om een aantal "fasen" af te spreken. Voordat aan een volgende fase wordt begonnen moet een bepaalde "mijlpaal" zijn bereikt. Hoe weten we wanneer we een mijlpaal hebben bereikt? Dat moeten we nagaan met een testprogramma of zelfs nette units tests! Dus op z'n *allerminst* met een `test.cpp` die de functionaliteiten in de betreffende klassen beknopt test.

Voor fasen en deadlines zouden wij het volgende voorstellen:

1. *Basis datastructuur.* Er kunnen herhaaldelijk waarden van verschillende typen worden geschreven naar de cellen en deze cellen kunnen ook worden uitgelezen. Het is mogelijk te itereren over kolommen en cellen binnen een kolom.

2. *Celadres parsen.* Het is mogelijk om een adres (`A4`) te interpreteren (parsen) en hiermee een cel uit te lezen. Ook is het mogelijk om een reeks (bijvoorbeeld `B3:B10`) te interpreteren en met een reeks-iterator alle cellen/waarden binnen deze reeks te bezoeken.

3. *Formule parsen.* Er kan al een begin worden gemaakt met de functionaliteiten voor formules. Zo zal het nodig zijn om een formule te parsen: we moeten controleren of de formule in de juiste vorm staat. Welke functie wordt er in de formule gebruikt? En op welke reeks wordt de formule toegepast?

Bovenstaande onderdelen wil je in orde hebben voor deadline A.

4. *UI basis.* De user interface kan de gehele sheet op het scherm tekenen. Ook wordt een cursor getekend welke met de pijltjestoetsen kan worden verplaatst.

5. *UI edit.* De user interface kan de cel onder de cursor bewerken middels een popup-window. Na het bewerken wordt de inhoud van de cel aangepast.

6. *Formules.* Schrijf een klasse `CellFormula` voor het implementeren van de formule-functionaliteit. De parser voor formules heb je al geschreven en gaat deel uitmaken van deze klasse. Schrijf functies om de verschillende formules te kunnen berekenen, maak gebruik van je reeks-iterator!

7. *Formules automatisch herberekenen.* Zorg ervoor dat formules automatisch worden herberekend wanneer een cel in de spreadsheet wordt aangepast. Er is een infrastructuur nodig zodat er een "callback" of "observer" methode kan worden aangeroepen in het geval een cel in de spreadsheet wordt veranderd.

8. *Extra's.* Werk aan extra functionaliteiten, indien van toepassing.

Programma compleet: Deadline B.

[1] Versie 1.60.0, documentatie is hier te vinden: `http://www.boost.org/doc/libs/1_60_0/`.

6 Schets van het ontwerp

Als startpunt geven we een schets van een mogelijk ontwerp mee. In de nog komende programmeeropdrachten zal het echter de bedoeling zijn dat je zelf een ontwerp voor het programma maakt. Bestudeer de schets goed en *maak tekeningen om het ontwerp voor jezelf inzichtelijk te maken.*

6.1 Cellen en celwaarden

De spreadsheet is opgebouwd uit cellen. Een cel duidt een plek aan waar een waarde kan worden opgeslagen. Dit wordt gemodelleerd met de klasse `Cell`. Verschillende cellen kunnen waarden bevatten van verschillende typen. Een celwaarde wordt gemodelleerd met een basisklasse `CellValueBase`. Een cel is dus alleen een vakje (placeholder), maar slaat zelf geen waarde op. Een cel wordt pas gerelateerd, of verbonden, met een waarde wanneer er een subklasse `CellValueBase` mee wordt verbonden. De klasse `Cell` heeft daarom de volgende eigenschappen:

- Het bevat een pointer naar een `CellValueBase`. Elke cel heeft een eigen, unieke pointer naar een `CellValueBase`, gebruik daarom een `unique_ptr`, bijvoorbeeld: `std::unique_ptr<CellValueBase> value`. Initialiseer de pointer met `nullptr` in de constructor (wat aangeeft dat de cel geen waarde bevat).
- De klasse heeft methoden om de waarde te zetten, uit te lezen en te legen.

De klasse `CellValueBase` heeft de volgende virtuele methoden:

- Virtuele destructor.
- Een methode om de waarde terug te geven als string, te gebruiken bij het tekenen van de spreadsheet op het scherm (tip: gebruik bijvoorbeeld `std::stringstream` of `boost::lexical_cast`).
- Een methode om de waarde terug te geven als string, te gebruiken bij het bewerken van deze waarde. (Denk aan formules: je wilt bij het bewerken van de cel de formule zelf zien, maar in andere gevallen de uitkomst van het berekenen van de formule).
- Een methode om de waarde terug te geven als getal, bijvoorbeeld als `float` (voor het doorrekenen van formules).

Vervolgens zorg je voor subklassen om verschillende typen waarden op te slaan: `int`, `float`, `std::string`. Schrijven we al deze subklassen zelf uit? Neen, er kan gebruik worden gemaakt van templates:

```
template<typename T>
class CellValue final : public CellValueBase
{
  private:
    T value;

  public:
    CellValue(T initial_value)
      : CellValueBase(), value(initial_value)
    { }
...
```

en zorg ervoor dat er hier een implementatie komt van alle virtuele methoden van `CellValueBase`. Het is zeer belangrijk dat de implementatie van deze methoden samen met de definitie van de template klasse *in de headerfile* worden geplaatst! De compiler heeft deze informatie nodig om in verschillende .cpp-bestanden waar het template wordt gebruikt (via include) de juiste codes te kunnen genereren. Nadat je het template hebt geïmplementeerd kun je verschillende celwaarden maken met `CellValue<int>`, `CellValue<std::string>`, enzovoort.

6.2 Spreadsheet datastructuur

Bouw de spreadsheet bijvoorbeeld kolomgewijs op. De spreadsheet bevat kolommen en de kolommen bevatten cellen. Zorg voor een klasse `Column` met de volgende eigenschappen:

- De kolom bevat een in de constructor gespecificeerd aantal cellen.
- Sla de cellen op een in STL container, bijvoorbeeld `std::vector`.
- Zorg voor een methode `getCell` om een referentie te krijgen naar een cel op een bepaalde rij.
- Zorg voor `begin` en `end` methoden welke een begin en eind-iterator van je container retourneren, zodat je gemakkelijk over de rijen in de kolom kunt itereren.

Vervolgens bouw je de spreadsheet, klasse `Sheet`, op met deze kolommen. Sla de kolommen op in een STL container, zorg voor een methode `getCell` en ook weer `begin` en `end` methoden.

6.3 Celadressering

Een cel in de spreadsheet wordt geadresseerd met een kolom- en rijnummer. Het is handig om een klasse `CellAddress` te schrijven waarin een kolom- en rijnummer kunnen worden opgeslagen. Gebruikers van de spreadsheet gebruiken echter adressen van de vorm `B4` (laten we deze celreferenties noemen), dus een kolomnaam gevolgd door een rijnummer. Ten eerste heb je functies nodig die kunnen converteren tussen kolomnaam en kolomnummer, en tussen rijnummer in een referentie (welke beginnen bij 1) en rijnummer in de spreadsheet (welke beginnen bij 0). Ten tweede is het handig om een extra constructor voor `CellAddress` te schrijven die een kolomnaam en rijnummer uit een referentie als parameters heeft. Tenslotte, zorg voor een functie, stel `createFromReference`, die gegeven een string met een celreferentie een `CellAddress` instantieert. Deze functie mag een exceptie gooien wanneer de string niet in het juiste formaat is en dus geen celreferentie bevat.

6.4 Reeksen en reeksiteratie

Een reeks (klasse `Range`) duidt een reeks van cellen in een spreadsheet aan en bestaat uit een start- en eindadres. Schrijf een klasse `Range` waarin een start- en eindadres (type `CellAddress`) en een referentie naar de `Sheet` zijn opgeslagen. Vervolgens willen we graag de mogelijkheid hebben om alle cellen in een reeks te bezoeken. Dit doen we door onze eigen iterator te schrijven, een `RangeIterator`. In de Appendix kun je een kort voorbeeld van een iterator vinden.

Zorg er verder voor dat de klasse `Range` `begin()` en `end()` methoden heeft die een iterator van het type `RangeIterator` retourneren. Schrijf ook hulpfuncties om een `Range*` te maken gegeven een sheet, start- en eindadres, en een functie die hetzelfde doet gegeven een sheet en een string bevattende een reeks in de vorm `A3:D6`. Maak bij het "parsen" van de tekstuele reeks gebruik van de `createFromReference` functie die je al hebt geschreven. Vang eventuele excepties van `createFromReference` op en retourneer een `nullptr` in zo'n geval.

6.5 User Interface

Zodra de sheet datastructuur goed werkt, kun je een begin maken met de user interface. De user interface richten we in volgens het zogenaamde Model-View-Controller (MVC) principe. In het Model wordt alle data opgeslagen, dat is dus de klasse `Sheet` die al is geschreven. De View is verantwoordelijk voor het afbeelden van het model, hiervoor schrijven we een klasse `SheetView`. Tenslotte is de controller verantwoordelijk voor het verwerken van de invoer van de gebruiker en het aansturen van `SheetView` op basis van deze invoer (bijvoorbeeld verplaatsen cursor). Hiervoor schrijven we een klasse `SheetController`.

Voor de `SheetView` mag je er altijd vanuit gaan dat het terminal 24 rijen en 80 kolommen heeft. De `SheetView` heeft verschillende verantwoordelijkheden:

- Initialiseren en deinitialiseren curses library (zie ook de "introductie tot *ncurses*" in de appendix). Het aanmaken van een curses "window".
- Het tekenen van de rij- en kolomheaders.
- Het tekenen van de inhoud van alle cellen. Loop door alle kolommen in de spreadsheet en alle rijen in een kolom, en roep `getString` op elke cel aan. Teken dit volgens naar het scherm, kap af na 8 tekens.
- Het tekenen van de cursor. Houd bijvoorbeeld de cursorlocatie bij in `SheetView` met een `CellAddress`.
- Methoden om de huidige cursorlocatie uit te lezen (get) en in te stellen (set).
- Zorg voor een methode `getChar` die door `SheetController` kan worden aangeroepen om invoer te lezen uit het curses "window".
- Methoden om een "edit popup window" te maken, te verwijderen en om karakters uit het "edit window" te lezen. Dit kan overigens ook op een nette manier in aparte klassen `EditWindow` en `EditController` worden geïmplementeerd.

En de verantwoordelijkheden voor `SheetController`:

- Uitlezen van karakters uit `SheetView` en deze verder verwerken. De juiste acties behorende bij het karakter aanroepen. Schrijf een methode `run` die als "main loop" van de applicatie dient, waarin met behulp van een `while` loop steeds wordt gewacht op invoer.
- Implementeer als eerste het "quit commando" (letter "q") zodat je de main loop kunt afsluiten.
- Verplaatsen van de cursor: vraag `SheetView` om de huidige cursorlocatie, pas de cursorlocatie aan en stuur de nieuwe cursorlocatie terug naar `SheetView`.
- Het starten van een celbewerking en een aparte "main loop" die wordt gedraaid zolang het "cell popup window" zichtbaar is.
- Als de celbewerking klaar is, doe dan het volgende:
  - Detecteer de inhoud van de invoer, is het een int, float, string of formule? Maak op basis hiervan een nieuwe `CellValueBase*` aan. Schrijf bijvoorbeeld een "factory functie" `cellValueFactory` en voeg deze toe als statische functie aan de klasse `CellValueBase`.
  - Zet de nieuwe "cell value" op de juiste cel.
  - Zorg ervoor dat de `SheetView` opnieuw wordt getekend (of nog beter: alleen de cell die is veranderd).

6.6 Formules

Een formule is een speciale "cell value". Maak `CellFormula` een subklasse van `CellValueBase`. Implementeer de verplichte methoden van `CellValueBase`. Daarnaast moet deze klasse een string die een formule voorstelt kunnen parsen en doorrekenen. Maak gebruik van de functie om een reeks te parsen die je eerder hebt geschreven en uiteraard van de `RangeIterator` om de cellen in een reeks af te lopen. Als de formule niet in het correcte formaat is, of de reeks opgegeven in de formule klopt niet (je ving hiervoor dan een exceptie op), onthoud dit en laat `getString()` in zo'n geval de string "ERR" retourneren zodat de gebruiker te zien krijgt dat er iets mis is.

De klasse hoeft alleen maar formules van de vorm `=NAAM(A3:D6)` te kunnen verwerken waarbij op de plek van `NAAM` staat: `SUM`, `AVG` of `COUNT` (deze functies moeten in ieder geval zijn ingebouwd).

Tenslotte wil je nog graag dat de formule zich kan herberekenen wanneer een cel in de reeks wordt aangepast. Je kunt hier als volgt voor zorgen:

- Maak een klasse `SheetObserver`, met hierin 1 virtuele methode `void cellChanged(const Cell &cell)`.
- `Sheet` houdt een lijst bij van pointers naar `SheetObserver` objecten.
- Wanneer een object een seintje wil ontvangen in geval van wijzigingen, moet het de `SheetObserver` klasse erven en de methode implementeren. Daarnaast dient het object zich met de `Sheet` te registeren met bijvoorbeeld een `Sheet::addObserver` methode. Dit is het geval voor `CellFormula`. (Maar het is ook handig voor bijvoorbeeld `SheetView` om het hertekenen te optimaliseren).
- `Cell::setValue` zou een seintje naar `Sheet` kunnen sturen wanneer een cel wordt aangepast.
- Als een reactie daarop loopt `Sheet` de lijst van observers af en roept de methode `cellChanged` aan.
- Een `CellFormula` krijgt nu een seintje in geval van een wijziging.

6.7 Bestanden inlezen/opslaan

Gegeven een goede `Sheet` datastructuur is het implementeren van save/load routines eenvoudig. Het opslaan van een objectstructuur naar een bestand wordt ook wel "serialization" genoemd. Maak een klasse `Serializable` met twee methoden:

```
virtual void serialize(std::ostream &output) = 0;
virtual void deserialize(std::istream &input) = 0;
```

De klassen `Sheet`, `Column` en `Cell` moeten deze klasse erven en de methoden implementeren. In de implementatie voor `Sheet`, schrijf je bijvoorbeeld eerst naar `output` hoeveel rijen en kolommen de spreadsheet bevat. Vervolgens loop je alle kolommen af en roep je daarop de `serialize` methode aan. Een kolom loopt alle rijen af en roept `serialize` aan op elke cel. De cel schrijft de data (bijvoorbeeld `editString`) naar het bestand.

Een bestand kan nu worden opgeslagen door een `std::ofstream` te maken, `serialize` aan te roepen op de `Sheet` en de stream te sluiten. Implementeer dit bijvoorbeeld in `SheetController` als reactie op het drukken op de "s" toets. Het exacte bestandsformaat wordt niet door ons gespecificeerd en mag naar eigen inzicht worden ingericht. Zowel ASCII als binair behoren tot de mogelijkheden.

Deserialization is het omgekeerde proces, je leest het bestand en maakt binnen elke cel de juiste nieuwe `CellValue` weer aan.

A Appendix: Korte voorbeelden

De hieronder gegeven codes zijn ook online beschikbaar in cut-n-paste vriendelijk formaat: `http://liacs.leidenuniv.nl/~rietveldkfd/courses/pt2017/voorbeeldcodes/`.

A.1 Het schrijven van een iterator

Hieronder volgt een kort (kunstmatig) voorbeeld van de definitie van een eigen container klasse en eigen iterator. Onze eigen container heeft `begin` en `end` methoden welke een iterator opleveren. Hierdoor kan de container worden gebruikt in een "range-based for loop". De iterator is van het type "input iterator" wat inhoudt dat de iterator elementen uit een container kan lezen en een enkele passage door de container kan maken.

```
/* To compile: c++ -std=c++14 -Wall -o example example.cc */

#include <iostream>
#include <vector>
```

```
#include <iterator>

/* Forward declaration */
class MijnIterator;

class MijnContainer
{
  private:
    std::vector<int> members;

  public:
    MijnContainer(void)
      : members{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 }
    { }

    int &getValue(const size_t offset)
    {
      return members[offset];
    }

    /* Dit voegen we te omdat het de conventie is om MijnContainer::iterator
     * te kunnen schrijven.
     */
    typedef MijnIterator iterator;

    MijnIterator begin(void);
    MijnIterator end(void);
};

class MijnIterator : public std::iterator<std::input_iterator_tag, int>
{
  private:
    MijnContainer &container;
    size_t offset;

  public:
    MijnIterator(MijnContainer &container, size_t offset)
      : container(container), offset(offset)
    { }

    /* Zijn de iteratoren gelijk aan elkaar? */
    bool operator==(const MijnIterator &iter) const
    {
      return &iter.container == &container && iter.offset == offset;
    }

    bool operator!=(const MijnIterator &iter) const
    {
      return !operator==(iter);
    }

    /* Element op de huidige plek van de iterator uitlezen */
    int &operator*() const
    {
      return container.getValue(offset);
    }

    int *operator->() const
    {
      return &container.getValue(offset);
    }

    /* Implementeer "++iter": verplaats de iterator een plek */
    MijnIterator &operator++()
    {
      ++offset;
```

```
      return *this;
    }
};

/* Implementatie methoden uit MijnContainer */
MijnIterator MijnContainer::begin(void)
{
  return MijnIterator(*this, 0);
}

MijnIterator MijnContainer::end(void)
{
  return MijnIterator(*this, members.size());
}


int main (void)
{
  MijnContainer container;

  /* Mogelijk omdat MijnContainer begin() en end() methoden heeft. */
  for (auto member : container)
    std::cout << member << " ";
  std::cout << std::endl;

  return 0;
}
```

A.2 Introductie tot *ncurses*

ncurses is een library die wordt gebruikt voor het programmeren van terminaluitvoer. Zo is het mogelijk de cursor te verplaatsen en op een bepaalde plaats tekst naar het terminal te schrijven. Op de meeste UNIX systemen is de library standaard geïnstalleerd. Het volgende korte voorbeeld demonstreert het gebruik van de *ncurses* library. Het initialiseert de library en tekent twee strings naar het scherm. De tweede string krijgt een opgelichte achtergrond (om te gebruiken bij bijvoorbeeld de kolomheaders). In een loop wordt er gewacht tot er op enter wordt gedrukt en daarna wordt *ncurses* gedeïnitialiseerd.

```
/* Compile: c++ -std=c++14 -Wall -o ncursesdemo ncursesdemo.cc -lcurses */

#include <curses.h>

static const int lines(24);
static const int cols(80);

int main(void)
{
  /* Initialiseren */
  initscr();
  noecho();

  /* Maak een venster, grootte lines x cols */
  WINDOW *win = newwin(lines, cols, 0, 0);
  keypad(win, TRUE); /* Enable keypad input */

  /* Verplaats cursor rij 10, kolom 20 */
  wmove(win, 10, 20);
  /* Plaats een string */
  waddstr(win, "HELLO WORLD!!");

  /* Nogmaals, maar nu op een achtergrond */
  attr_t old_attr; /* Huidige settings onthouden */
  short old_pair;
  wattr_get(win, &old_attr, &old_pair, NULL);
```

```
    wattron(win, A_STANDOUT);
    wmove(win, 12, 20);
    waddstr(win, "HELLO WORLD!!!");
    wattr_set(win, old_attr, old_pair, NULL); /* Oude settings terugzetten */

    /* Wacht tot er op enter wordt gedrukt */
    int ch;
    while ((ch = wgetch(win)) != '\n');

    delwin(win);  /* Dealloceer venster */
    endwin();  /* Curses stoppen */

    return 0;
}
```

Enige tips voor het gebruik van *ncurses*:

- Het programma dient `initscr` en `endwin` maar één keer aan te roepen.
- Als aanpassingen niet op het scherm verschijnen: kijk dan eens naar de functie `wrefresh`.
- Er zijn "defines" voor pijltjestoetsen (`KEY_DOWN`, enz.) en speciale toetsen (`KEY_BACKSPACE`, `KEY_ENTER`) die worden gebruikt samen met `wgetch`.
- Een popup-window maken gaat het makkelijkst door een nieuw window te maken van de benodigde grootte. Je kunt automatisch een rand (border) tekenen met de functie `wborder`.
- Voor de Enter toets wil je de volgende keycodes testen: `KEY_ENTER`, `'\n'` en `'\r'`.
- Voor de backspace en delete toetsen (om cellen te wissen) wil je de volgende keycodes testen: `KEY_BACKSPACE_`, `KEY_CLEAR`, `KEY_DL`, `KEY_DC`, `0x7f`.

Er kan ook met kleur worden gewerkt in *ncurses*, maar voor de opdracht is dat niet verplicht. Meer informatie kan onder andere worden gevonden op:

- Ncurses manual pages in HMTL formaat: `http://invisible-island.net/ncurses/man/ncurses.3x.html`, scroll naar beneden naar "Routine Name Index" voor een lijst links naar subpagina's die de verschillende functies beschrijven.
- Ncurses Programming Guide:
  `http://hughm.cs.ukzn.ac.za/~murrellh/os/notes/ncurses.html`
- NCURSES Programming HOWTO:
  `http://invisible-island.net/ncurses/NCURSES-Programming-HOWTO.html`