TDDD86 – Laboration #1

24 augusti 2022

Syftet med den här laborationen är att öva grundläggande C++, som funktioner, strängar och I/O-strömmar. Du kommer också att öva på att bryta ned ett större problem i mindre deluppgifter och att representera dessa deluppgifter som väldesignade funktioner. Filerna du behöver för att komma igång finns som `labb1.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 1 redovisning''` och en `git push`. Informera sedan din assistant genom att maila honom/henne. Se till att sista versionen av dessa filer är med:

- **life.cpp**, C++-koden för Game of Life-simuleringen, och
- **mycolony.txt**, din egen unika indatafil för Game of Life med en bakteriekolonis ursprungstillstånd,

Game of Life

Game of Life är en simulering påhittad av den brittiske matematikern J. H. Conway 1970 och populariserad av Martin Gardner. Spelet är en simulering som modellerar livscykeln hos bakterier med hjälp av ett tvådimensionellt rutnät av celler. Givet ett initialt mönster simulerar spelet födelse och död hos framtida generationer av celler med en uppsättning enkla regler. I den här uppgiften kommer du att implementera en förenklad version av Conways simulering och ett enkelt textbaserat användargränssnitt för att titta på när bakterierna växer över tid.

Ditt Game of Life-program ska börja med att be användaren om ett filnamn och använda den filens innehåll för att bestämma ursprungstillståndet för ditt rutnät av bakterier. Sedan ska det låta användaren få kolonin att fortskrida genom generationer av tillväxt. Användaren kan mata in `t` för att "ticka" fram bakteriesimuleringen en generation, eller `a` för att starta en animationsloop som tickar fram simuleringen flera generationer, en var hundrade millisekund, eller `q` för att avsluta programmet. Alla kommandon är med små bokstäver och alla andra tangenter kan ignoreras.

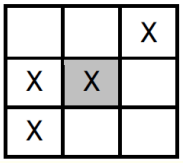
Här är en interaktionslogg mellan ditt program och en användare (med användarens indata understruken):

```
Welcome to the TDDD86 Game of Life,
a simulation of the lifecycle of a bacteria colony.
Cells (X) live and die by the following rules:
 - A cell with 1 or fewer neighbours dies.
 - Locations with 2 neighbours remain stable.
 - Locations with 3 neighbours will create life.
 - A cell with 4 or more neighbours dies.

Grid input file name? simple.txt
---------
---------
---XXX---
---------
---------
a)nimate, t)ick, q)uit? t
---------
----X----
----X----
----X----
---------
a)nimate, t)ick, q)uit? t
---------
---------
---XXX---
---------
---------
a)nimate, t)ick, q)uit? q
Have a nice Life!
```

Regler för Game of Life-simuleringen:

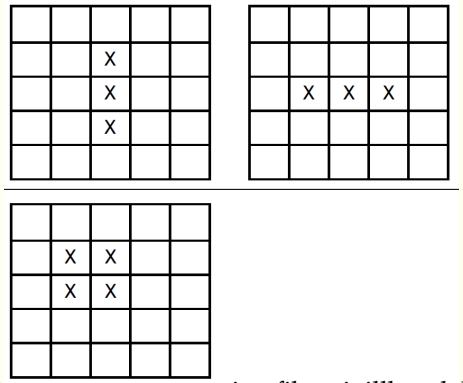
Varje plats i rutnätet är antingen tom eller så finns där en levande cell `X`. Grannarna till en plats är de eventuella celler som finns i de omgivande åtta positionerna. I exemplet till höger har den skuggade mittenplatesen tre grannar som innehåller levande celler. En ruta på kanten av rutnätet har färre än åtta grannar. Till exempel har rutan med `X` längst upp till höger bara tre närliggande rutor och eftersom endast en av dem innehåller en levande cell, så den har bara en levande granne.



Simuleringen startar med ett initialt mönster av celler på rutnätet och beräknar efterföljande generationer av celler enligt följande regler:

- En position som har noll eller en granne kommer att vara tom i nästa generation. Om en cell fanns där så dör den.
- En position med två grannar är stabil. Om den hade en cell innehåller den fortfarande en cell. Om den var tom fortsätter den att vara tom.
- En position med tre grannar kommer att innehålla en cell i nästa generation. Om positionen var tom föds en ny cell. Om den för närvarande innehåller en cell, kvarstår cellen.
- En position med fyra eller fler grannar kommer att vara tom i nästa generation. Om det finns en cell i den positionen dör den av överbefolkning.

Födslarna och dödsfallen som transformerar en generation till nästa äger alla rum samtidigt. När du beräknar nästa generation påverkar inte nya födslar/dödsfall i den generationen andra celler i den generationen. Eventuella förändringar (födslar eller dödsfall) i en given generation k börjar påverka sin omgivning först i generation $k + 1$.



Kontrollera din förståelse av spelreglerna genom att titta på exemplen till höger. De två mönstren uppe till höger bör alternera för evigt. Mönstret underst till höger ändrar sig inte mellan iterationerna eftersom varje cell har exakt tre levande grannar. Detta kallas för en stabil partition.

Indatafiler:
Rutnätet av bakterier i ditt program får sitt ursprungstillstånd från en av en uppsättning filer vi tillhandahåller. När ditt program läser rutnätsfilen får du anta att filen existerar och att dess innehåll är giltigt. Du behöver inte skriva kod för att kontrollera om filerna finns, inte heller kod för att hantera en fil i fel format. Beteendet hos ditt program i sådana fall är inte definierat i det här dokumentet; det kan krascha, avslutas, etc. Du får också anta att filnamnet användaren skriver in inte innehåller några mellanslag och att all annan indata från användaren är giltig.

I varje indatafil innehåller de två första raderna heltal r och c, vilka representerar antalet rader respektive antalet kolumner i rutnätet. Följande rader innehåller själva rutnätet, en mängd tecken av storlek $r \times c$ med en radbrytning (`\n`) efter varje rad. Varje tecken i rutnätet är antingen ett `'-'` (minustecken) för en tom död cell, eller ett `'X'` (stort X) för en levande cell. Indatafilen kan ha ytterligare rader med information efter rutnätsraderna, som till exempel kommentarer; sådant innehåll ska ignoreras av ditt program.

När Qt Creator exekverar ditt program kommer indatafilerna att finnas i samma katalog som ditt program. Till exempel skulle följande text kunna vara innehållet i en fil `simple.txt`, ett 5×9-rutnät med 3 initialt levande celler:

```
5
9
---------
---------
---XXX---
---------
---------
```

← antal rader hög
← antal kolumner bred

← rutnät av rader och kolumner
- är en död cell, X är en levande cell

Implementationsdetaljer:
Rutnätet av bakterieceller ska lagras i en 2-dimensionell array/fält, men arrayer/fält i C++ saknar vissa egenskaper och är i allmänhet krångliga för nybörjare i C++ att använda. De känner inte till sin egen längd, de orsakar konstiga buggar om du indexerar utanför arrayen och de kräver att man behärskar saker som peka-

re och minnesallokering. Så istället för att använda en array för att representera ditt rutnät kommer du att använda ett objekt av typen `Grid`, en klass vi tillhandahåller.

Ett `Grid`-objekt ger en renare abstraktion av en 2-dimensionell datamängd, med flera användbara metoder och egenskaper. Här är några användbara medlemmar i `Grid`-klassen:

Medlem	**Beskrivning**
`Grid<T>()`	Konstruerar ett tomt rutnät.
`Grid<T>(rows, columns)`	Konstruerar ett rutnät av given storlek, där varje cell lagrar ett värde av typen `T`.
`grid[row][column]`	Enskilda celler i rutnätet kan kommas åt och modifierar med hakparenteser.
`get(row, column)`	Returnerar innehållet i en enskild cell.
`inBounds(row, column)`	Returnerar `true` om det givna rad/kolumn-indexet är inom gränserna för rutnätet eller `false` om detta inte är fallet.
`numCols()`	Returnerar antalet kolumner i rutnätet.
`numRows()`	Returnerar antalet rader i rutnätet.
`resize(nRows, nCols)`	Ändrar rutnätets storlek till de givna dimensionerna. Eventuellt innehåll i rutnätet kastas bort.
`set(row, column, value)`	Sätter innehållet i en enskild cell till det angivna värdet.
`toString()`	Returnerar en strängrepresentation av rutnätet.

Du kan också använda tilldelningsoperatorn = för att kopiera tillståndet i ett `Grid`-objekt till ett annat.

Din `main`-funktion ska skapa din `Grid` och föra över den till övriga funktioner. Eftersom det är dyrt att göra kopior av en `Grid` ska du *alltid* se till att använda en referens om du för över den som parameter från en funktion till en annan (`Grid&`, inte `Grid`). Eftersom du inte känner till storleken på rutnätet förrän du läser indatafilen får du anropa `Grid`-objektets `resize`-metod när du tagit reda på den rätta storleken.

Ditt program har ett konsollbaserat användargränssnitt. Du kan skriva till konsollen med `std::cout` och läsa indata från den med `std::cin`. För att hjälpa till med animeringen finns följande funktioner i `lifeutil.h`:

Medlem	**Beskrivning**
`pause(ms)`	Får programmet att stanna i angivet antal millisekunder.
`clearConsole()`	Raderar den synliga texten i konsollen.

Lämplig arbetsgång:
Det är frestande att skriva hela programmet på en gång och sedan försöka kompilera och köra det. Vi rekommenderar inte att du gör så. Utveckla istället ditt program inkrementellt: Skriv en liten del av funktionaliteten, testa/debugga och gå sedan vidare till en annan liten del. Här är en lista över möjliga steg att ta för att komma fram till en lösning:

1. Få igång projektet och skriv kod för att skriva ut programmets välkomstmeddelande.
2. Skriv kod som ber om ett filnamn och öppnar och skriver ut den filens innehåll till konsollen. När detta fungerar, försök läsa enskilda celler i rutnätet och omvandla dem till ett `Grid`-objekt. Skriv ut `Grid`-objektets tillstånd på konsollen med `toString` för att bekräfta att det har rätt data i sig. Använd ett enkelt testfall, som `simple.txt`.
3. Skriv kod för att skriva ut rutnätets nuvarande tillstånd, utan att modifiera tillståndet.
4. Skriv kod för att avancera rutnätet från en generation till nästa.
5. Implementera programmets huvudmeny och animeringen.

När du försöker avancera bakterierna från en generation till nästa kan du inte göra detta "in place" genom att modifiera rutnätet samtidigt som du loopar över det. Om du gör det ändrar du cellerna och deras grannar och förstör då antalet levande grannar för närliggande celler. Du kommer alltså behöva skapa ett andra rutnät temporärt. Ditt existerande rutnät representerar den nuvarande generationen bakterier och du kan skapa ett andra, temporärt, rutnät som låter dig beräkna och lagra nästa generation utan att ändra i den nuvarande. När du väl har fyllt det andra rutnätet med information om nästa generations celler kan du kopiera tillbaka innehållet i originalrutnätet med tilldelningsoperatorn =.

En knepig bit med den här uppgiften är att kanterna av rutnätet måste hanteras försiktigt eftersom du inte vill att din kod försöker komma åt celler utanför rutnätets gränser. Din algoritm för att undersöka celler och räkna grannar bör hantera kanter och inre celler elegant och utan redundans så långt det är möjligt. Vissa studenter försöker uppnå detta genom att skapa ett extra lager runtom rutnätet, så till exempel en indatafil av storlek 5×9 lagras i en `Grid` av storlek 7×11. Vi vill inte att du löser problemet på det sättet av flera skäl; det undviker

flera av svårigheterna med uppgiften, det slösar med minne och det introducerar omotiverade hack för att se till att indexeringen fungerar korrekt.