

**i Institutt for datateknologi og informatikk****Eksamen TDT4102 - Prosedyre- og objektorientert programmering**

**Eksamensdato** : Torsdag 27. mai 2021.

**Eksamenstid (fra-til)** : 0900-1300 + 30 min til filopplasting

**Hjelpemiddelkode/Tillatte hjelpemiddel**: A / Alle skriftlige/trykte hjelpemiddel tillatt

**Faglig kontakt under eksamen** : Rune Sætre, Truls Asheim og Lasse A. Eggen

**Tlf** : 452 18 103 (Rune), +45 2282 5830 (Truls), 913 69 975 (Lasse)

**Email**: tdt4102-fagans@idi.ntnu.no

**Teknisk hjelp under eksamen**: NTNU Orakel. Tlf : 73 59 16 00

**ANNEN INFORMASJON**

Gjør dine egne antagelser og presiser i besvarelsen hvilke forutsetninger du har lagt til grunn i tolking/avgrensing av oppgaven. Faglig kontaktperson skal kun kontaktes dersom det er direkte feil eller mangler i oppgavesettet.

På flervalgsspørsmål får du positive poeng for riktige svar og negative poeng for feil svar. Summen vil aldri være mindre enn null poeng for et spørsmål, selv om alle svarene dine på det spørsmålet er feil.

**Juks/plagiat:**

Eksamen skal være et individuelt, selvstendig arbeid. Det er tillatt å bruke hjelpemidler, men vær obs på at du må følge eventuelle anvisninger om kildehenvisninger under. Under eksamen er det ikke tillatt å kommunisere med andre personer om oppgaven eller å distribuere utkast til svar. Slik kommunikasjon er å anse som juks. Alle besvarelser blir kontrollert for plagiat. Du kan lese mer om juks og plagiering på eksamen her: <https://innsida.ntnu.no/wiki/-/wiki/Norsk/Juks+påeksamen>

**Kildehenvisninger:**

Selv om "Alle hjelpemiddel er tillatt", er det ikke tillatt å kopiere andres kode og levere den som din egen. Du kan se på andre åpent tilgjengelige ressurser, og deretter skrive din egen versjon av det du så, i henhold til copyright-forskrifter.

**Varslinger:**

Hvis det oppstår behov for å gi beskjeder til kandidatene underveis i eksamen (for eksempel ved feil i oppgavesettet), vil dette bli gjort via varslinger i Inspira. Et varsel vil dukke opp som en dialogboks på skjermen i Inspira. Du kan finne igjen varselet ved å klikke på bjella øverst i høyre hjørne på skjermen. Det vil i tillegg bli sendt SMS til alle kandidater for å sikre at ingen går glipp av viktig informasjon. Ha mobiltelefonen din tilgjengelig.

**Vekting av oppgavene:**

Del 1 teller ca. 20% av totalen, Del 2 teller ca. 20% av totalen, og Del 3 teller ca. 60% av totalen på denne eksamen. Du vil få F på eksamen hvis du svarer blankt på en av de tre delene.

**Slik svarer du på oppgavene:**

Alle oppgaver som ikke er av typen filopplasting, skal besvares direkte i Inspira. I Inspira lagres svarene dine automatisk hvert 15. sekund.

NB! Klipp og lim fra andre programmer frarådes, da dette kan medføre at formatering og elementer (bilder, tabeller etc.) vil kunne gå tapt.

## Filoplasting:

Når du jobber i andre programmer fordi hele eller deler av besvarelsen din skal leveres som filvedlegg – husk å lagre besvarelsen din med jevne mellomrom.

Merk at alle filer må være lastet opp i besvarelsen før eksamenstida går ut.

Det framgår av filopplastingsoppgaven hvilket filformat som er tillatt (**zip**).

Det er lagt til **30 minutter** til ordinær eksamenstid for eventuell digitalisering av håndtegninger og opplasting av filer. Tilleggstida er forbeholdt innlevering og inngår i gjenstående eksamenstid som vises øverst til venstre på skjermen.

NB! Det er ditt eget ansvar å påse at du laster opp riktige og intakte filer. Kontroller zip-filen du har lastet opp ved å klikke “Last ned” når du står i filopplastingsoppgaven. Alle filer kan fjernes og byttes ut så lenge prøven er åpen.

Pass på at det ikke finnes noe forfatterinformasjon i filen(e) du skal levere.

I siste del av eksamen skal du bruke samme program som du satte opp i øving 0. Du må også vite hvordan du laster ned, pakker ut, og setter opp mapper fra en zip-fil som VS Code (eller tilsvarende) C++-prosjekter på maskinen din. Til slutt må du være i stand til pakke alle slike mapper sammen i en zip-fil igjen for å levere det du har kodet, innen tidsfristen, for å kunne bestå denne eksamen.

## Eksamen - step by step

Denne listen vil lede deg trinn for trinn igjennom hva du skal gjøre på denne eksamen.

1. Les nøye igjennom disse introduksjons-sidene
2. **Last ned** medfølgende zip-fil med en gang eksamen starter. I zip-filen finner du .cpp- og .h-filer samt oppgaveteksten til del 3.
3. Les **forklarlingen** på problemet gitt i begynnelsen av hver seksjon.
4. Du kan bruke VS Code (eller et hvilket som helst annet utviklingsmiljø du foretrekker) til å åpne og jobbe med den oppgitte koden (som i Øving 0). All kode skal være C++.
5. Du kan bruke boken eller andre online/offline ressurser, men du kan **IKKE** samarbeide med andre på noen måte, eller direkte kopiere og lime inn online kode som om det er din egen.
6. Etter å ha fullført hver enkelt kodeoppgave bør du huske å lagre.
7. Send inn koden din selv om den ikke kan kompileres og / eller ikke fungerer riktig.  
Fungerende kode er **IKKE** et krav for at du skal stå, men det er en fordel.
  - Last opp all den komplette koden som en .zip-fil. Ikke endre den opprinnelige mappestrukturen. For å få bestått på denne eksamen er det HELT AVGJØRENDE AT DU LASTER OPP ZIP-FILEN. Etter eksamensslutt (13:00) har du 30 minutter til rådighet til dette. Vi anbefaler likevel at du laster opp zip-filen minst en gang underveis i eksamenstiden.
  - Det er mulig å oppdatere både enkelt-svarene og filopplastningen flere ganger, i tilfelle du retter på noe etter første innlevering.
  - Prøv å last opp filen en gang midt i eksamenstiden, for å se at du klarer det, og hvor lang tid du bruker på det.
8. Husk at funksjonene du lager i en deloppgave ofte er ment å skulle brukes i andre deloppgaver. Selv om du står helt fast på en deloppgave bør du likevel prøve å løse alle eller noen av de etterfølgende oppgavene ved å anta at funksjoner fra tidligere deloppgave er riktig implementert.
9. Før manuell sensureringen av eksamen din, vil vi foreta automatisk testing og plagiatkontroll av all koden du har levert. Basert på resultatene kan det hende vi ber deg om en forklaring, og dette kan påvirke eksamensresultatet ditt. Du må huske å legge til kommentarlinjer i koden din, siden det også kan hjelpe oss å forstå koden din bedre.

## Nedlastning av start-fil (zip)

- Last ned og lagre .zip-filen (lenken er på Inspira). Gjør dette med en gang eksamen starter... i tilfelle noe er galt, eller internett detter ut av og til underveis.
- Husk å lagre den på et sted på datamaskinen du husker og kan finne igjen.
- Pakk ut (unzip) filen.

- Begynn å jobbe med oppgavene i VS Code (eller et hvilket som helst annet utviklingsmiljø du foretrekker). Vi forventer at du vet hvordan du sammenstiller og kjører kode, slik det ble forklart i øving 0 på begynnelsen av semesteret.
- Filene vi leverer ut kompilerer til et kjørende program, men du må selv skrive resten av koden for alle oppgavene og prøve å få hver programmet til å kjøre som et eget prosjekt. Det er ikke et krav at all den innleverte koden kan kjøres, men det er en fordel.
- Etter at du er ferdig med kodingen av alle del-spørsmål, må du laste opp alle kodefilene dine, etter at de på nytt er pakket sammen til en lignende .zip-fil (**ikke 7z, rar eller andre**) som den du begynte med. Ikke endre noe på de opprinnelige mappe/fil-navnene før du zipper filen og laster den opp til Inspira igjen. Last gjerne opp på nytt hver time!

### **Automatisk innlevering**

Besvarelsen din leveres automatisk når eksamenstida er ute og prøven stenger, forutsatt at minst én oppgave er besvart. Dette skjer selv om du ikke har klikket «Lever og gå tilbake til Dashboard» på siste side i oppgavesettet. Du kan gjenåpne og redigere besvarelsen din så lenge prøven er åpen. Dersom ingen oppgaver er besvart ved prøveslutt, blir ikke besvarelsen din levert. Dette vil anses som "ikke møtt" til eksamen.

### **Trekk/avbrutt eksamen**

Blir du syk under eksamen, eller av andre grunner ønsker å levere blankt/avbryte eksamen, gå til "hamburgermenyen" i øvre høyre hjørne og velg «Lever blankt». Dette kan ikke angres selv om prøven fremdeles er åpen.

### **Tilgang til besvarelse**

Du finner besvarelsen din i Arkiv etter at sluttida for eksamen er passert.

**☑ REGLER OG SAMTYKKER**

Dette er en **individuell** øving. Du har ikke lov til å kommunisere (gjennom web-forum, chat, telefon, hverken i skriftlig, muntlig eller annen form), ei heller samarbeide med noen andre under eksamen.

**Før du kan fortsette til selve øvingen må du forstå og SAMTYKKE i følgende:**

**Under øvingen:**

**Jeg skal IKKE motta hjelp fra andre.**

☐ Aksepter

**Jeg skal IKKE hjelpe andre eller dele løsningen min med noen.**

☐ Aksepter

**Jeg skal IKKE copy-paste noe kode fra noen eksisterende online/offline kilder. (Du kan se, og deretter skrive din EGEN versjon av koden).**

☐ Aksepter

**Jeg er klar over at øvingen kan bli underkjent uavhengig av hvor korrekt svarene mine er, hvis jeg ikke følger reglene og/eller IKKE aksepterer disse utsagnene.**

☐ Aksepter

- 1.1 Hver av følgende kodebiter inneholder en feil. Kategoriser feilen som en av A. Syntaksfeil, B. Type-feil, C. Kjøretidsfeil eller D. Logisk feil. Hvis en kodesnutt inneholder flere feil, velger du den første fra listen i forrige setning. Vi er interessert i feil som rapporteres/oppstår når koden kompileres og/eller kjøres.

a)

```
vector<int>* a;  
a->push_back(1);
```

Velg ett alternativ:

- ☐ Syntaksfeil
- ☐ Type-feil
- ☐ Kjøretidsfeil
- ☐ Logisk feil

b)

```
float pi()  
    return 3.14;
```

Velg ett alternativ

- ☐ Syntaksfeil
- ☐ Type-feil
- ☐ Kjøretidsfeil
- ☐ Logisk feil

c)

```
bool is_month(int month) {  
    return month > 1 || month < 12;  
}
```

**Velg ett alternativ**

- ☐ Syntaksfeil
- ☐ Type-feil
- ☐ Kjøretidsfeil
- ☐ Logisk feil

d)

```
vector<int> v;  
v.push_back("Hi");
```

**Velg ett alternativ**

- ☐ Syntaksfeil
- ☐ Type-feil
- ☐ Kjøretidsfeil
- ☐ Logisk feil

---

Maks poeng: 10

- 1.2** Følgende kode beregner statistikk fra flere kast med mynt (mynt eller krone kalles heads or tails på engelsk). Programmet teller antall tap, antall seire og den lengste seiersrekken og skriver det ut.

```

1  struct Game {
2      int count;
3      string bet;
4      bool we_won;
5  };
6
7  void print_stats(vector<Game>& games) {
8      int wins = 0;
9      int losses = 0;
10     int streak = 0;
11
12     int current_streak = 0;
13     for ( auto game : games ) {
14         if (game.we_won) {
15             current_streak++;
16             wins++;
17         } else {
18             losses++;
19             current_streak = 0;
20         }
21         streak = max(streak, current_streak);
22     }
23     cout << "Game stats: " << "\n" <<
24         " Lost: " << losses << endl <<
25         " Won: " << wins << endl <<
26         " Longest streak: " << streak << endl;
27 }
28
29 int main() {
30     vector<Game> outcomes {
31         {1, "heads", false},
32         {2, "heads", true},
33         {3, "tails", true},
34         {4, "heads", true}
35     };
36     print_stats(outcomes);
37     return 0;
38 }

```

Hvis koden ovenfor fungerte riktig, skulle den gi følgende utskrift.



Game stats:

Lost: 1

Won: 3

Longest streak: 3

For øyeblikket er det imidlertid noen feil som hindrer koden i å gjøre jobben sin. Hvilke av de følgende linjene inneholder feil? Velg en eller flere.

**Velg ett eller flere alternativer**

☐ 14

☐ 16

☐ 15

☐ 24

☐ 19

☐ 12

☐ 13

☐ 21

☐ 18

☐ 23

---

Maks poeng: 10

### 1.3 I denne oppgaven ser vi på egenskapene til constexpr funksjoner.

a) Hvilke av følgende definisjoner av funksjonen **my\_log2** er gyldige? Dvs. velg alle som kompilerer og kjører uten feil.

**Velg ett eller flere alternativer**

☐ `constexpr unsigned int my_log2(unsigned int n) {  
 unsigned int* res = new unsigned int;  
 *res = std::log2(n);  
 return *res;  
}`

☐ `constexpr unsigned int my_log2(unsigned int n) {  
 return ( (n<2) ? 1 : 1+my_log2(n/2));  
}`

☐ `#include <cmath>  
constexpr unsigned int my_log2(unsigned int n) {  
 unsigned int res = std::log2(n);  
 cout << "Computed log2 result " << res << endl;  
 return res;  
}`

☐ `constexpr unsigned int my_log2(unsigned int n) {  
 return std::log2(n);  
}`

b) Hvilke (null eller flere) av følgende utsagn er korrekte?

**Velg ett eller flere alternativer**

- ☐ **constexpr**-funksjoner kan bare kalle andre funksjoner hvis de også er **constexpr**.
- ☐ Bruk av **constexpr** kan åpne for verdi-avhengige kode-optimaliseringer
- ☐ Programytelsen kan noen ganger forbedres ved å beregne verdier på kompileringstidspunktet ved hjelp av **constexpr**
- ☐ Typesikkerheten til programmet vil bli bedre siden resultatet av **constexpr** funksjonen er kjent på kompileringstidspunktet

---

Maks poeng: 10

**1.4** Hvilke av følgende utsagn er sanne?

2.5 poeng per korrekt svar, -2.5 poeng for feil svar, 0 for vet ikke

a) En **int** kan inneholde et større tall enn en **unsigned int**.

**Velg ett alternativ**

- ☐ Sant
- ☐ Usant
- ☐ Vet ikke

b) En **double** gir høyere presisjon enn en **float**.

**Velg ett alternativ**

- ☐ Sant
- ☐ Usant
- ☐ Vet ikke

c) Aritmetiske operasjoner med **double** og **float** er alltid nøyaktige

**Velg ett alternativ**

- ☐ Sant
- ☐ Usant
- ☐ Vet ikke

d) I **double x = floor((double) a / (double) b); double y = a/b**, får x og y identisk verdi for alle heltall a og b, hvis  $b \neq a$ .

**Velg ett alternativ:**

- ☐ Sant
- ☐ Usant
- ☐ Vet ikke

---

Maks poeng: 10



- 1.5 Følgende kode skal legge sammen to matriser, men den fungerer ikke som den skal. Her er først et eksempel på hvordan elementvis addisjon av to matriser egentlig skal utføres.

$$\begin{pmatrix} 3 & 6 \\ 2 & 9 \\ 8 & 4 \end{pmatrix} + \begin{pmatrix} 7 & 2 \\ 1 & 1 \\ 5 & 0 \end{pmatrix} = \begin{pmatrix} 3+7 & 6+2 \\ 2+1 & 9+1 \\ 8+5 & 4+0 \end{pmatrix}$$

```

1  struct Matrix {
2      const size_t rows;
3      const size_t cols;
4      const vector<vector<int>> data;
5
6      Matrix(const vector<vector<int>> data)
7          : rows{data.size()}, cols{data.begin()->size()}, data{data} {}
8
9      const vector<int> & operator[](int i) const {
10         return data[i];
11     }
12
13     Matrix operator+(const Matrix &m) {
14         vector<int> rows;
15         for (unsigned int i = 0; i < data.length(); i++) {
16             vector<int> col;
17             for (unsigned int j = 0; j < data[i].length(); j++) {
18                 col.push_back(data[j][i] + m[j][i]);
19             }
20             rows.push_back(col);
21         }
22         return Matrix(rows);
23     }
24 };

```

Hvilke av kodelinjene inneholder feil? Velg null eller flere her. Mulige typer feil inkluderer logiske feil, type-feil og udefinerte funksjoner.

**Velg ett eller flere alternativer**

☐ 17

☐ 18

☐ 13

☐ 10

☐ 20

☐ 9

☐ 15

☐ 22

☐ 16

☐ 14

---

Maks poeng: 10

**1.6** Hvilke av følgende utsagn er sanne?

a) Å sette inn et element i midten av en **list** er raskere enn å sette inn et element i midten av en **vector**.

**Velg ett alternativ**

- ☐ Sant
- ☐ Usant
- ☐ Vet ikke

b) I stedet for å bruke et **map**, kan man like godt bruke en **vector** av **tupler**.

**Velg ett alternativ**

- ☐ Sant
- ☐ Vet ikke
- ☐ Usant

c) Alle elementene i et **set** er unike.

**Velg ett alternativ**

- ☐ Sant
- ☐ Usant
- ☐ Vet ikke

d) For tilfeldig tilgang (random access) er en **vector** raskere enn en **array**.

**Velg ett alternativ**

- ☐ Vet ikke
- ☐ Usant
- ☐ Sant

e) Å bruke en **array** er utrygt i tilfelle utilsiktede tilganger utenfor array-området skjer.



**Velg ett alternativ:**

- ☐ Vet ikke
- ☐ Sant
- ☐ Usant

---

Maks poeng: 10

- 2.1** Bruk **template** til å implementere en funksjon **print\_container** som skriver ut elementene i en iterabel container med et linjeskift mellom hvert element.

```
1  int main() {  
2      set<string> test_set{"a", "b", "c"};  
3      vector<string> test_vector{"d", "e", "f"};  
4      print_container(test_set);  
5      print_container(test_vector);  
6  }
```

Funksjonen skal oppføre seg slik at koden ovenfor skriver ut:

a  
b  
c  
d  
e  
f

Implementere funksjonen **print\_container**. Du kan utelate include, etc.

**Skriv ditt svar her**

1	
---	--

Maks poeng: 10

**2.2** Les igjennom følgende kode.

```
1  void f(const string g) {
2      stringstream a(g);
3      string b;
4      vector<string> c;
5      char d = ',';
6
7      while (getline(a, b, d)) {
8          c.push_back(b);
9      }
10
11     for(auto h = c.begin(); h != c.end(); h++) {
12         cout << *h << endl;
13     }
14 }
```

a) Variabelnavnene til denne koden er endret slik at koden nå er vanskelig å forstå. Kan du forklare hva funksjonen **f** gjør?

**Skriv ditt svar her**

b) Hva blir skrevet ut når følgende main-funksjon kjøres?

```
1  int main(){
2      f("is,this,hard?");
3  }
```

**Skriv ditt svar her**

---

Maks poeng: 10

## 2.3 Følgende kode flytter en robot til venstre, høyre, opp eller ned i henhold til brukerkommandoer.

```

1  class Robot {
2  private:
3      int x; int y;
4  public:
5      Robot(int initial_x, int initial_y): x(initial_x), y(initial_y) {}
6      void move(int direction) {
7          switch (direction) {
8              case 0: x--; break;
9              case 1: x++; break;
10             case 2: y++; break;
11             case 3: y--; break;
12             default: throw runtime_error("Invalid direction");
13         }
14     }
15     void show() {
16         cout << "Robot is at (" << x << ", " << y << ")" << endl;
17     }
18 };
19
20 int main() {
21     Robot r(0, 0);
22     while (true) {
23         int direction;
24         string input;
25         cout << "Where should the robot go?" << endl;
26         cin >> input;
27         if (input == "left") direction = 0;
28         else if (input == "right") direction = 1;
29         else if (input == "up") direction = 2;
30         else if (input == "down") direction = 3;
31         else throw runtime_error("Invalid direction");
32         r.move(direction);
33         r.show();
34     }
35     return 0;
36 }

```

Programmet er ganske uleselig på grunn av alle de “magiske tallene” det bruker, f.eks. kalles **move(1)** for å flytte roboten til høyre.

a) Hvilket C++ konsept bør brukes til å omskrive programmet på en mer lesbar måte? Spesielt spør vi etter en måte å spesifisere robotens retninger som symbolske konstanter.

**Skriv ditt svar her**

b) Skriv litt kode (kanskje en definisjon) som er første steg mot implementering av løsningen du foreslo:

**Skriv ditt svar her**

---

Maks poeng: 10

## 2.4 Vi ser på første utkast til en klasse som skal inneholde en matrise

```
1  template<typename T> class Matrix {
2  private:
3      int rows;
4      int cols;
5      T* data;
6  public:
7      Matrix(int rows, int cols) : rows{rows}, cols{cols} {
8          data = new T[rows*cols];
9      }
10 };
11
12 int main() {
13     for (int i = 0; i < 100; i++){
14         Matrix<int> m{100,100};
15     }
16     return 0;
17 }
```

a) Kan du identifisere problemet med koden over? Hint: Tenk på hva som skjer hvis vi øker antall ganger for-løkken på linje 13 kjøres

**Skriv ditt svar her**

b) Hvordan vil du fikse koden? Skriv koden du vil legge til i klassedefinisjonen over, her:

**Skriv ditt svar her**

---

Maks poeng: 10

**2.5** Klassen **StaticLengthVector** nedenfor implementerer en innpakking (wrapper) rundt standard vector-typen, og begrenser mengden av elementer den kan inneholde.

```

1  template <typename T>
2  class StaticLengthVector {
3      size_t size;
4      vector<T> vec;
5
6  public:
7      StaticLengthVector(size_t size) :
8          size{size} {}
9
10     void add_element(T el) {
11         if (vec.size() >= size) {
12             throw runtime_error("No more space in StaticLengthVector");
13         } else {
14             vec.push_back(el);
15         }
16     }
17 };

```

Funksjonelt er denne koden riktig, men siden vi vet størrelsen på en **StaticLengthVector** på forhånd kan vi unngå at **push\_back** i **add\_element** gjør mer minneomfordeling enn nødvendig for stadig å sikre nok plass til den voksende underliggende vector.

a) Hva ville du gjort for å unngå unødvendige minneoverføringer i den underliggende **vector**?

**Skriv ditt svar her**

b) Skriv koden du vil legge til i klassedeklarasjonen til **StaticLengthVector** for å implementere forslaget ditt.

**Skriv ditt svar her**

---

Maks poeng: 10

**2.6** Se på følgende kode

```
1  int main() {  
2      vector<string> names {"Bjarne", "Bell", "Ford"};  
3      for (string name : names) {  
4          name += "_cool";  
5          cout << name << '\n';  
6      }  
7  }
```

a) Hva blir utskriften fra koden over?

**Skriv ditt svar her**

b) Hva er resultatet hvis vi endrer linje 3 til

```
for (const string& name : names) {
```

**Skriv ditt svar her**

c) Hva er resultatet fra koden ovenfor hvis vi endrer linje 3 til

```
for (string& name : names) {
```

**Skriv ditt svar her**

---

Maks poeng: 10



**i VIKTIG:**

Zip-filen du skal laste ned inneholder kompilerbare (og kjørbare) .cpp- og .h-filer med forhåndskodede deler og en full beskrivelse av oppgavene i del 3 som en PDF-fil. Etter å ha lastet ned zip-filen står du fritt til å bruke et utviklingsmiljø etter eget valg (for eksempel VS Code) for å jobbe med oppgavene.

For å få bestått på denne eksamen er det **HELT AVGJØRENDE AT DU LASTER OPP ZIP-FILEN**. Etter eksamensslutt (13:00) har du 30 minutter til rådighet til dette

De korte svarene i Inspira (del 1 og 2) lagres automatisk hvert 15. sekund, helt til eksamenstiden er slutt. Og zip-filen (i del 3) kan også endres / lastes opp flere ganger. Så hvis du vil gjøre noen endringer etter at du har lastet opp filen, kan du bare endre koden, zippe den sammen på nytt, og laste opp filen igjen. Når prøvetiden er over, vil siste versjon av alt du har skrevet eller lastet opp i Inspira automatisk bli sendt inn som ditt gjeldende svar, så **sørg for at du laster opp minst en gang halvveis, og en gang før tiden går ut.**

På neste side kan du laste ned .zip-filen, og senere laste opp den nye .zip-filen med din egen kode inkludert. Husk at PDF-dokumentet med alle oppgavene er inkludert i zip-filen du laster ned.

### 3.1 LAST NED

[Trykk her for å laste ned utdelt kode](#)

### LAST OPP

Last opp all den komplette koden som en .zip-fil. Ikke endre den opprinnelige mappestrukturen. For å få bestå prøven er det **HELT AVGJØRENDE AT DU LASTER OPP DEN NYE ZIP-FILEN DIN KORREKT**, minst en gang i løpet av de 4 timene til rådighet.

Det er mulig å oppdatere både enkeltsvarene og filopplastningen **flere ganger**, i tilfelle du retter på noe etter første innlevering.


**Prøv å last opp en oppdatert zip-fil minst en gang ekstra, midt i prøvetiden**, for å se at du klarer det, og for å finne ut hvor lang tid du bruker på det.

Last opp zip-filen med besvarelsen din her. Alt i én zip-fil.



**Last opp filen her. Maks én fil.**

Alle filtyper er tillatt. Maksimal filstørrelse er **50 GB**.

 Velg fil for opplasting

---

Maks poeng: 200

## Question 25

Attached



## Part III: Conway's Game of Life

Maksimal score for del 3 er 200 poeng.

### Introduksjon til Conway's Game of Life

Conway's Game of Life er en cellulær automaton. Det er en simulering av et 2D-univers med rektangulære celler som deterministisk utvikler sin tilstand over tide. Hvis du kjører en simulering med de samme initialbetingelsene vil tilstanden i et gitt tidssteg være det samme mellom distinkte simuleringer. Det er et antall forskjellige livsformer, mønster, som kan representeres i Game of life. Noen mønster forflytter seg i universet med tiden, de kalles romskip. I den utdelte zip-filen vil du finne filer med filendelse `.cgo1` som inneholder et utvalg romskip.

Din oppgave er å implementere logikken som setter liv til universet. Celler er den primære byggeblokken i universet og du vil lage et uendelig rutenettsunivers av celler. For hvert tidssteg som går skal tilstanden i universet reflektere et sett regler som er bestemt for Game of life. Hva som bestemmer utfallet av reglene er antall levende og døde naboceller og den nåværende tilstanden til cellen regelen anvendes på.

Vi har inkludert en kort demo-video av hvordan programmet ser ut når det er ferdig implementert. Videoen ligger i zip-filen og heter `game-of-life_demo.mp4`. I videoen demonstreres handlingene som kan utføres og hvordan de utarter seg i det endelige programmet (klippet er uten lyd).

### Hvordan besvare del 3?

Alle oppgavene i del 3 er satt opp slik at de skal besvares i filen `Gameoflife.cpp`. Hver oppgave har en tilhørende unik kode for å gjøre det lettere å finne frem til hvor i filen du skal skrive svaret. Koden er på formatet `<tegn><siffer>` (TS), eksempelvis C1, C2 og G1. I `Gameoflife.cpp` vil du for hver oppgave finne to kommentarer som definerer henholdsvis begynnelsen og slutten av koden du skal føre inn. Kommentarene er på formatet:

```
// BEGIN: TS og //END: TS.
```

**Det er veldig viktig at alle svarene dine er skrevet mellom slike kommentar-par**, for å støtte sensurmekanikken vår. Hvis det allerede er skrevet noen kode mellom BEGIN- og END-kommentarene i filene du har fått utdelt, så kan, og ofte bør, du erstatte den koden med din egen implementasjon.

For eksempel, for oppgave C1 ser du følgende kode i utdelte `Gameoflife.cpp`

```
1  int Cell::get_value() const {
2      // BEGIN: C1
3      return 0;
4      // END: C1
5  }
```

Etter at du har implementert din løsning, bør du ende opp med følgende istedenfor

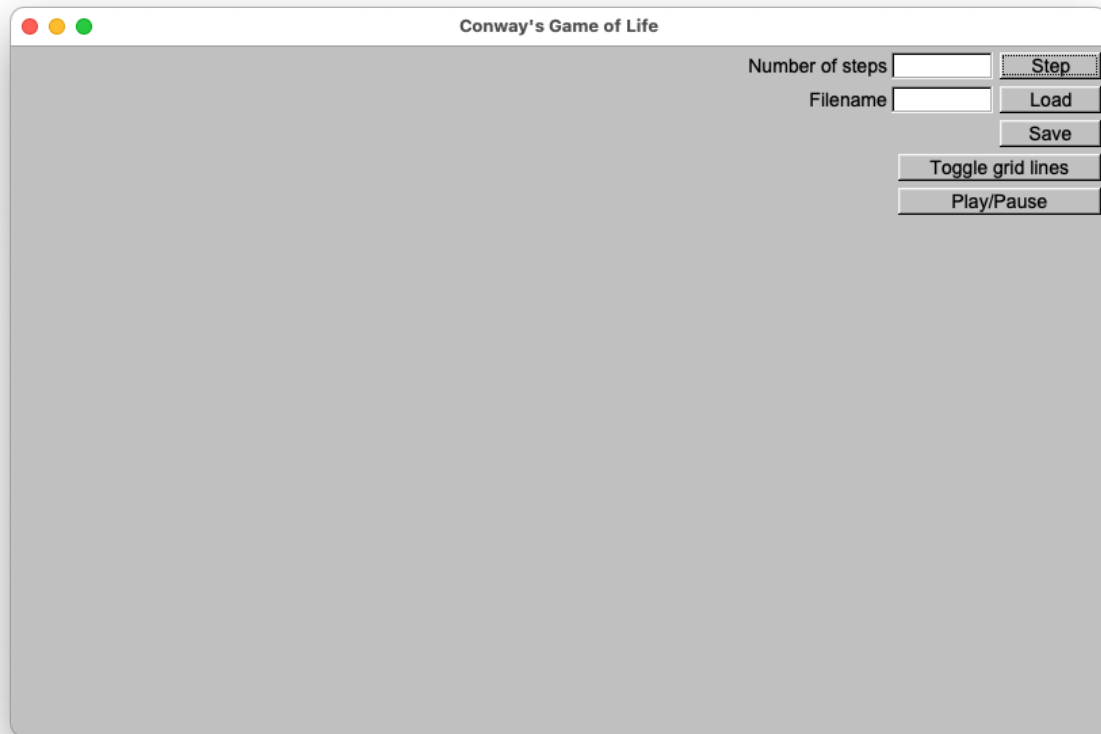
```
1  int Cell::get_value() const {
2      // BEGIN: C1
3
4      /* Din kode her */
5
6      // END: C1
7  }
```

Merk at BEGIN- og END-kommentarene **IKKE skal fjernes**.

Til slutt, hvis du synes noen av oppgavene er uklare, oppgi hvordan du tolker dem og de antagelsene du må gjøre som kommentarer i den koden du sender inn.

I zip-filen finner du bl.a. `Gameoflife.h` og `Gameoflife.cpp`. Det er kun `Gameoflife.cpp` som skal redigeres for å komme i mål med oppgavene. Headerfilen inneholder klassedefinisjoner og konstanter du bør ta en titt på før du starter å svare på spørsmålene i denne delen.

Før du starter må du sjekke at den (umodifiserte) utdelte koden kjører uten problemer. Du skal se det samme vinduet som i figur 1. Når du har sjekket at alt fungerer som det skal er du klar til å starte programmering av svarene dine.



**Figur 1:** Utlevert kode uten endringer.

## Klassen Cell (80 poeng)

Vi deler ut en ikke komplett versjon av klassen `Cell` som representerer en celle i Game of life-universet. Din oppgave er å gjøre cellen synlig på skjermen, endre dens tilstand (vekke den til live eller ta livet av den) og oppdatere den grafiske representasjonen.

En `Cell` har noen attributter du bør kjenne til. `enum class State` er en *scoped enum* som enumererer en celles tilstand, om den er i live eller død. Medlemsvariabelen `State state` holder orden på cellens nåværende tilstand. `shared_ptr<Rectangle> rect` er den grafiske representasjonen av en celle på skjermen.

Du kan bruke en `shared_ptr` på samme måte som du bruker en `unique_ptr` eller rå peker, f.eks. derefereringsoperatoren, `*`. Vi bruker en `shared_ptr` i denne eksamenen for å automatisk rydde opp `Rectangle`-instansen som er delt mellom kopier av et `Cell`-objekt.

På et tidspunkt vil du lese og skrive fra en fil, og kanskje terminalen for å debugge. Til det formålet har vi definert en tilstand til tegn-tabell i tabell 1. Vi har delt ut et sett tabeller og avbildninger(`map`) du bør vurdere å bruke for å oversette mellom tilstand og tegn-representasjonen. De er definert i header-filen `Gameoflife.h` og heter `Cell::chars` og `Cell::char_to_state`.

Tilstand	Tegn
Live	'#' (hash)
Dead	'.' (punktum)

**Tabell 1:** Tilstand til tegn-mapping.

1. (10 points) **C1: Implementer medlemsfunksjonen** `Cell::get_value()`.

Denne medlemsfunksjonen returnerer heltallsrepresentasjonen av tilstandsvariabelen.

(Hint: scope enum-en `State` representerer sine enumeratorer med heltallsverdier.)

Verdiene som skal returneres fra denne funksjonen er 0 for døde celler og 1 for levende celler.

2. (10 points) **C2: Implementer** `Cell::update()`.

Medlemsfunksjonens oppgave er å oppdatere den grafiske tilstanden til cellen, altså sette fyllfargen av medlemsvariabelen `rect`. En død celle er *svart* og en levende celle er *hvit*. Vi har delt ut et array som heter `colors` i klassedefinisjonen til `Cell` i filen `Gameoflife.h` og oppfordrer til bruk av dette for å løse oppgaven.

Husk å kall på denne funksjonen når du endrer tilstanden til en celle.

3. (10 points) **C3: Implementer** `Cell::kill()`.

Sett cellens tilstand til `State::Dead`.

Husk også å oppdatere cellens grafiske tilstand.

4. (10 points) **C4: Implementer** `Cell::resurrect()`.

Funksjonen komplementerer `Cell::kill()` og skal gjøre det motsatte av den, nemlig bringe cellen tilbake til live.

5. (10 points) **C5: Implementer** `Cell::set_state(char c)`.

Programmen vil snart lese tegn fra en fil og du må konvertere tegnrepresentasjonen lest fra filen til en tilstand, `State`. Oversettingen er listet i tabell 1 og vi har delt ut et *constant* map navngitt

`Cell::char_to_state` som inneholder denne mappingen.

Input-argumentet `c` er tegnet som representerer tilstanden cellen skal ha blitt tilegnet når funksjonen har returnert. Du kan anta at `c` alltid er gyldig og er enten `'.'` eller `'#'`.

6. (10 points) **C6: Implementer** `operator>>(istream& is, Cell& cell)`.

Denne operatoroverlastingen leser et tegn fra input-strømmen og oppdaterer tilstanden til cellen basert på input-verdien.

7. (10 points) **C7: Implementer** `Cell::is_alive()`.

Medlemsfunksjonen returnerer `true` hvis cellen er i live, `false` hvis ikke.

8. (10 points) **C8: Implementer** `Cell::as_char()`.

Denne medlemsfunksjonen returnerer tegn-representasjonen av cellens tilstand.

Bruk den samme tilstand til tegn-tabellen som før, fra tabell 1. Vi har delt ut et array som heter `Cell::chars` som passer fint til denne oppgaven.

## Universet (90 poeng)

Nå skal vi ta fatt på universet og legge til regler så vi kan få et innblikk i hvordan Game of life fungerer.

Klassen `Gameoflife` er definert i `Gameoflife.h` og de aller fleste definisjonene av medlemsfunksjonene befinner seg i `Gameoflife.cpp`. Du bør ta en titt i headerfilen for å få et overblikk over medlemmene til klassen `Gameoflife`.

Legg merke til linjen `using Grid = std::vector<std::vector<Cell>>`. Denne linjen lager et alias med navn `Grid` vi kan bruke til erstatning for en 2D-vector som inneholder `Cell`-objekter. Klassen lagrer to `Grid` i medlemsvariabelen `std::array<Grid, 2> grid`. Til enhver tid vil et av `grid`-ene (rutenettene) være det nåværende og det er det nåværende `grid`-et som skal vises på skjermen, samtidig vil det andre `grid`-et være midlertidig og du kan bruke det som mål for neste nåværende tilstand. Det er nyttig når universets tilstand skal endre seg med tiden, siden vi trenger kjennskap til den umodifiserte nåværende tilstanden helt til den neste tilstanden er helt ferdig beregnet. For å få tak i den nåværende tilstanden kan du kalle på medlemsfunksjonen `Gameoflife::get_current_grid()` og motsatt `Gameoflife::get_scratch_grid()` for å få tilgang til den midlertidige.

For å holde styr på hvilket `grid` som for øyeblikket lagrer verdiene for nåværende og midlertidig tilstand har vi opprettet to heltallsvariable: `current_grid` og `scratch_grid`. Til enhver tid skal en av dem inneholde heltallsverdien 0 og den andre inneholde 1. `Gameoflife::get_current_grid()` og `Gameoflife::get_scratch_grid()` returnerer nåværende og midlertidig `grid` basert på verdiene i disse variablene. *Du må endre verdiene til variablene `current_grid` og `scratch_grid` for å bytte på hvilket `grid` som er nåværende og midlertidig når det er en overgang som krever to separate tilstander.*

Vi har også definert noen konstanter i `Gameoflife.h` for å hjelpe deg å lage et rutenett av celler som passer vinduet. `int margin` holder en verdi som gjør at GUI-elementer i vinduet holder avstand til hverandre. `int cell_size` holder cellens grafiske størrelse i x- og y-aksen - alle celler er representert av kvadratiske rektangler på skjermen.

Du kan anta en konstant størrelse for Game of life-grid i alle oppgavene. Alle utdelte universtilstander er formatert som rutenett med størrelse 50x50. Det er den samme størrelsen som er lagret i konstantene `x_cells` og `y_cells`.

9. (30 points) **G1: Implementer konstruktøren til** `Gameoflife`.

Din oppgave er å initialisere de to 2D-rutenettene som inneholder `Cell`-objekter. Når du har gjort denne oppgaven skal vinduet se ut som det i figur 2.

I `Gameoflife.h` finner du konstantene `cell_size` og `margin` som du bør bruke for å plassere cellerekteklene i et rutenett inne i vinduet. Det grafiske rutenettet skal plasseres `margin` punkter fra topp- og venstrekanterne til vinduet. Antall celler på x- og y-aksen er lagret i hhv. variablene `x_cells` og `y_cells`.

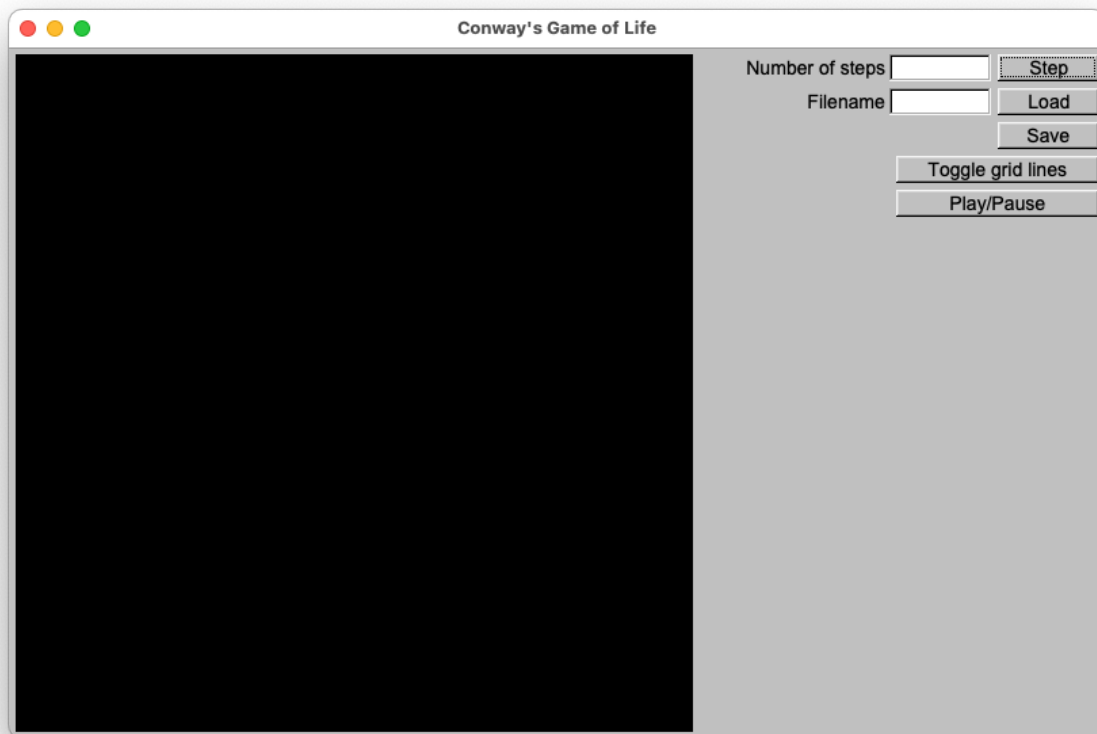
**Vi anbefaler at du først fyller ut ett av rutenettene og kaller på cellenes medlemsfunksjon `attach_to()` i det, før du deretter gjøre en kopi-tilordning (tilordningsoperatoren) av hele rutenettet**

til det andre (tomme) rutenettet. Dette vil fungere siden kopiering av en *shared\_ptr* kun vil kopiere pekeren, ikke instansen det pekes til.

Vi har delt ut medlemsfunksjonen `Cell::attach_to(Window& win)` som kobler den grafiske representasjonen til et vindu. Se til at denne funksjonen kalles for kun en kopi av et `Cell`-objekt.

Hvordan legger du til rader til en 2D-vector, hvordan legger du til et element i en 2D-vector og hvordan kobler du et element til vinduet? Vi viser tre kodelinjer i figur 3 som viser hvordan du kan lage et rutenett med størrelsen  $1 \times 1$  med en `Cell` som har størrelse 40 plassert i vinduets posisjon  $(0, 0)$ .

(Du vil se at deler av konstruktøren allerede er definert. Disse delene tar seg av å koble GUI-elementer til vinduet og sjekker at input-argumentene er positive. Den gjør også at vinduet og dets elementer holder en konstant størrelse når programmet kjører.)



**Figur 2:** Game of life med kun døde celler (etter oppgave 9. G1).

```
// Add a new row to the grid
get_current_grid().push_back({});
// Add a Cell to that row
get_current_grid().back().push_back(Cell{Point{0,0}, 40});
// Attach that Cell to the window
get_current_grid().back().back().attach_to(*this);
```

**Figur 3:** Eksempelkode som setter opp et  $1 \times 1$ -rutenett med en celle med størrelse 40 plassert i vinduets  $(0, 0)$ .



10. (10 points) **G2: Implementer** `operator>>(istream& is, Gameoflife& gameoflife)`.

Les fra input-strømmen inn i den samsvarende cellen i den nåværende tilstanden til ditt Game of life.

Input-formatet er `y_cells` antall linjer med `x_cells` tegn per linje. Se for øvrig i en av de utleverte filene med filendelse `.cgo1` hvordan en fullstendig initialtilstand ser ut. Vi har listet et eksempel-input med størrelse 5x5 i figur 4.

```
.....  
..#..  
...#.  
.###.  
.....
```

**Figur 4:** Eksempel-input med størrelse 5x5 som inneholder romskipet kalt "glider".

11. (10 points) **G3: Implementer** `Gameoflife::load(const std::string& filename)`.

Denne medlemsfunksjonen laster inn en ny tilstand fra filen gitt i argumentet `filename`. Hvis filen ikke kan åpnes eller leses skal det kastes et unntak, `std::runtime_error`, med meldingen "Could not load a Game of life state from <filename>.", der du bytter ut <filename> med filnavnet programmet ikke kan lese.

Kall på medlemsfunksjonen `Window::redraw()` etter at tilstanden er lest fra filen for å signalisere til vinduet at det skal oppdateres med endringene i den grafiske tilstanden til cellene.

Når du har implementert denne funksjonen skal det være mulig å laste inn universene vi har delt ut som del av `.zip`-filen. Hvis du laster inn `glider.cgo1` skal du se et vindu som ligner det i figur 5.

Tips: for å spare tid og slippe å skrive inn et filnavn hver gang programmet starter kan du legge til linjen `load("glider.cgo1")` på slutten av `Gameoflife`-konstruktøren for å laste inn glideren automatisk hver gang programmet starter på nytt.

12. (30 points) **G4: Implementer** `Gameoflife::step()`.

Denne oppgaven handler om `Gameoflife::step()` uten parametere i signaturen.

Medlemsfunksjonen vil overføre universet fra tilstand til tilstand ettersom det utvikler seg over tid. En celledes tilstand i tid  $n + 1$  er et produkt av universets tilstand i tid  $n$ .

Hvordan en celledes liv utvikler seg fra et tidspunkt til det neste kan defineres av disse tre reglene: ([https://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway's_Game_of_Life))

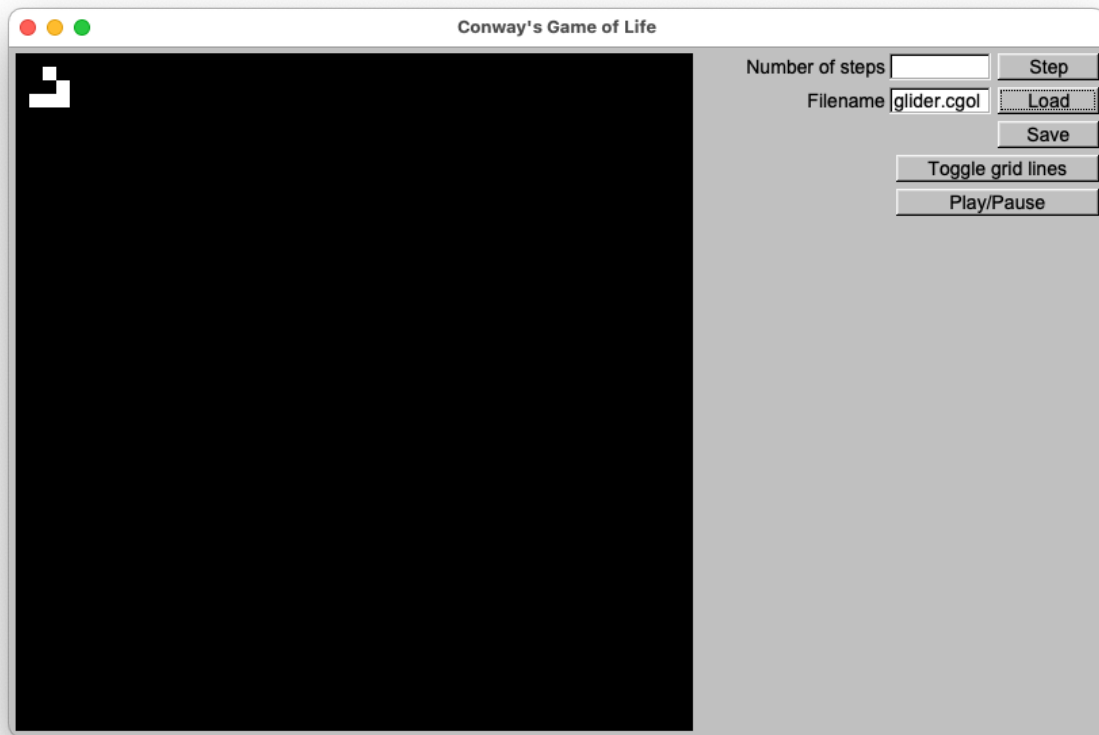
1. Enhver levende celle med to eller tre levende naboer overlever.
2. Enhver død celle med tre levende naboer blir til en levende celle.
3. Alle andre celler dør i den neste generasjonen. Likedan, alle andre døde celler forblir døde.

Merk at du har to rutenett til disposisjon. Et som holder den nåværende tilstanden som også er den som vises grafisk i vinduet, og et midlertidig rutenett du kan bruke som kladd (scratchpad) og mål for neste tilstand.

Din oppgave er å implementere overførselen fra den nåværende tilstanden til den neste tilstanden. Når `Gameoflife::step()`-funksjonen har returnert forventes det at både den nye nåværende tilstanden og den grafiske representasjonen (`Cell`-ens `rect`-medlemsvariabel) i Game of life-vinduet reflekterer den nylig beregnede tilstanden.

Universet i Game of life er uendelig, selv om vårt rutenett er endelig i 2D-rommet. Dette betyr at du må folde rundt (wrap around) rutenettets kanter for å beregne et steg i tid korrekt. Denne oppførselen er illustrert i figur 6.

Vær obs på at modulo-operatoren (%) har en noe ikke-intuitiv oppførsel ved behandling av negative heltall, som kan bli et problem ved folding rundt kantene. Positive heltall fungerer som vi forventer,



**Figur 5:** Glider lastet inn.

f.eks.  $4\%3 = 1$  og  $3\%4 = 3$ . Negative tall, derimot, vil for samme eksempel gi  $-4\%3 = -1$  og  $-3\%4 = -3$ .

I det siste eksempelet ser vi at  $-3$  ikke er heltallsresultatet vi ønsker oss for å folde rundt kanten. Vi hadde ønsket å få resultatet 1.

En måte å få modulo-oppførselen vi ønsker er å legge sammen venstre- og høyresiden, deretter ta modulo av summen med høyresiden. Algebraisk får vi da at  $a\%n$  blir  $(a + n)\%n$ . Fullføring av eksempelet med  $-3\%4$  blir da  $(-3 + 4)\%4 = 1$ , som ønsket.

13. (10 points) **G5: Implementer** `Gameoflife::step(int steps)`.

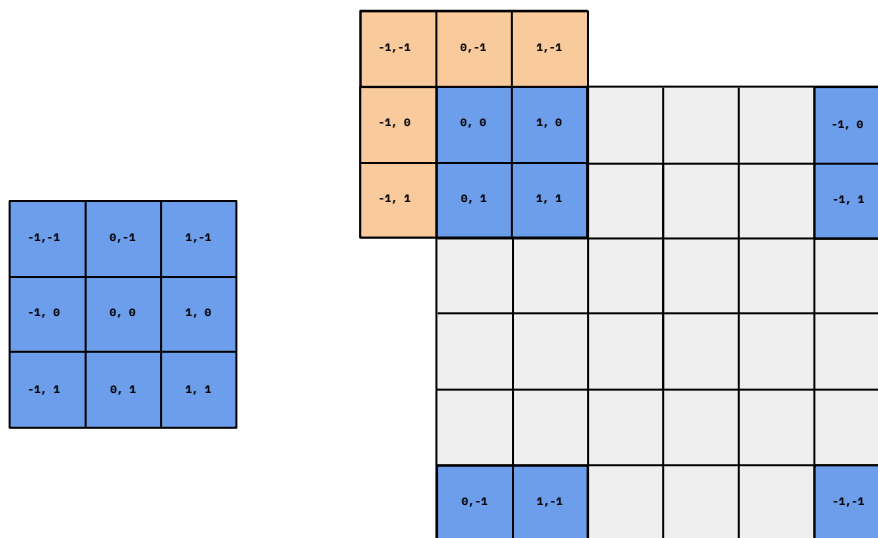
Denne medlemsfunksjonens oppgave er å gjøre fremskritt i tilstanden med `steps` antall tidstrinn.

Når funksjonen har utført forflytningen i tid må du kalle på den arvede medlemsfunksjonen `Window::redraw()` for å signalisere til vinduet at du har oppdatert grafikk inni det.

For å verifisere at programlogikken er korrekt kan du laste inn tilstanden fra filen `glider.cgol` og utføre 50 steg. Vinduet ditt skal etter stegene ligne det i figur 7. Du kan bruke knappen "Toggle grid lines" for å skru av eller på linjer i rutenettet for å lettere se hvordan cellene i universet utvikler seg.

Tips 1: når du skriver inn et tall i boksen med merkelappen "Number of steps" og trykker på "Step"-knappen vil tiden forflyttes med antall tidssteg i intervallet  $[1, 100]$  - det betyr bl.a. at 0 som verdi blir 1 steg når du trykker "Step".

Tips 2: når denne funksjonen er implementert kan du også trykke "Play/Pause"-knappen for å automatisk utvikle tiden slik at du får en animasjon i vinduet. Hvis du vil endre oppdateringsraten til



**Figur 6:** Eksempel på folding rundt kanter. Den blå kjernen til venstre anvendes på det lysegrå rutenettet til høyre. Anvendelsen av kjernen vises overlagt i rutenettets punkt (0,0). Orange firkanter viser delene av kjernen som må foldes rundt kantene til rutenettet. De blå firkantene er de konkrete avbildningene.

animasjonen kan du se på variabelen `animation_interval` i `Gameoflife.h`

## Ekstrafunksjonalitet (30 poeng)

### Veksle (toggle) en celle med pekerklikk

I den siste serien med oppgaver skal du gjøre rutenettet klikkbart og veksle tilstanden til cellen det klikkes på. Vi har allerede opprettet en behandler som fanger opp pekerklikk inni vinduet. Din oppgave er å prosessere klikkene, dvs. finne ut om klikket skjedde inni rutenettet og veksle cellen under pekeren.

14. (10 points) **E1: Implementer** `Gameoflife::cell_at_pos(Point pos)`.

Denne metoden tar inn en posisjon som argument og returnerer en peker til `Cell`-objektet som befinner seg på den posisjonen i vinduet. Hvis det ikke er en celle på posisjonen returneres en `nullptr`.

Merk at en posisjon som treffer en celle i den grafiske representasjonen er en celle i den *nåværende* tilstanden, så du skal returnere en peker til en celle fra det rutenettet.

15. (10 points) **E2: Implementer** `Cell::toggle()`.

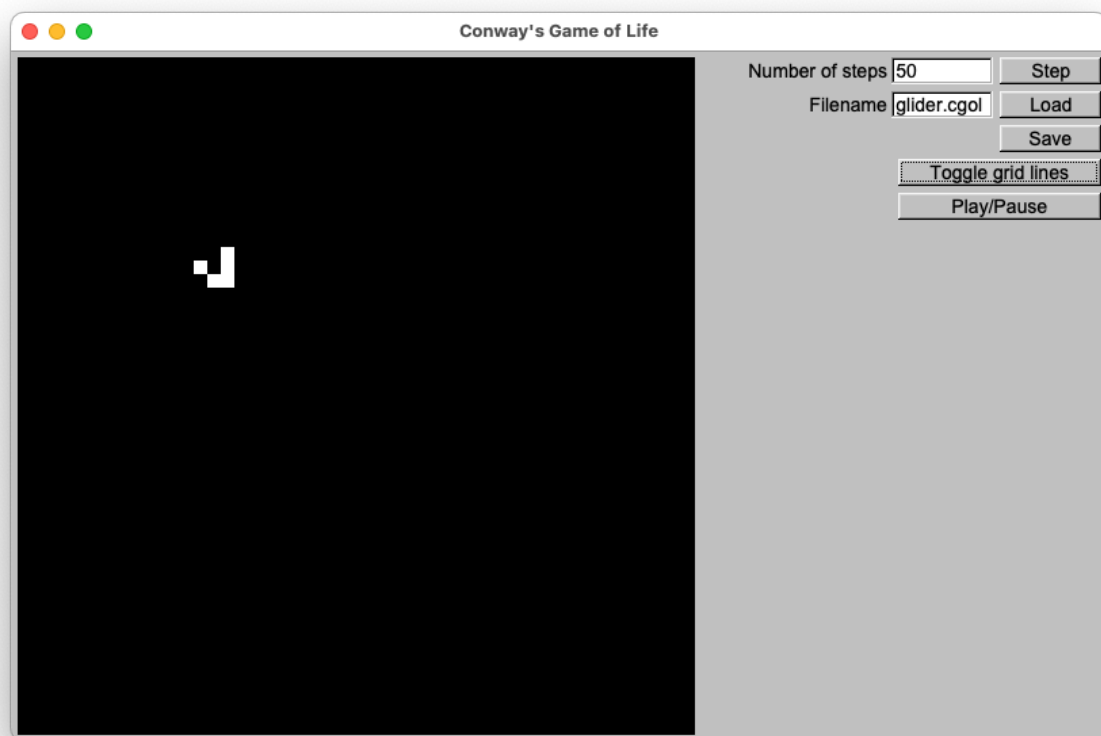
Denne medlemsfunksjonen skal veksle en celles tilstand. Er cellen død skal den bli levende og omvendt skal den bli død hvis den er levende.

16. (10 points) **E3: Implementer** `Gameoflife::toggle_cell(Point pos)`.

Medlemsfunksjonen tar inn en posisjon og veksler tilstanden til cellen på den posisjonen. Hvis den ikke lykkes i å veksle en celle, f.eks. fordi at det ikke finnes en celle på posisjonen, skal den returnere `false`, ellers skal den returnere `true`.

Siden du endrer på den nåværende (og grafiske) tilstanden må du signalisere til vinduet at grafikken er oppdatert.

Når du har gjennomført denne oppgaven skal du kunne klikke på en vilkårlig celle i rutenettet og visuelt se at cellens tilstand veksler mellom levende og død.



**Figur 7:** En glider lastet fra `glider.cgol` etter 50 steg .