



Institutt for datateknikk
og informasjonsvitenskap

TENTATIVT LØSNINGSFORSLAG TDT4102 - Prosedyre- og objektorientert programmering

Faglig kontakt under eksamen: Trond Aalberg

Tlf: 97631088

Eksamendato: 03. juni 2015

Eksamenstid: 09-13

Hjelpemiddelkode: C: Spesifiserte trykte og håndskrevne hjelpemidler tillatt.
Walter Savitch, Absolute C++ eller
Lyle Loudon, C++ Pocket Reference.

Målform/språk: Bokmål

Antall sider: 11 sider inklusive forside og vedlegg

Kontrollert av

Dato

Sign

Merk! Studenter finner sensur i Studentweb. Har du spørsmål om din sensur må du kontakte instituttet ditt. Eksamenskontoret vil ikke kunne svare på slike spørsmål.

Generell introduksjon

Les gjennom oppgavetekstene nøye. Noen av oppgavene har lengre tekst, men dette er for å gi kontekst, introduksjon og eksempler til oppgavene.

Når det står “*implementer*” eller “*lag*” skal du skrive en fungerende implementasjon: hvis det handler om en funksjon skal du skrive deklarasjonen med returtype og parametertype(r) og hele funksjons-kroppen.

Når det står “*deklarer*” er vi kun interessert i funksjons- eller klassedeklarasjonen. Typisk vil dette være deklarasjoner du vanligvis finner i header-filer.

Hvis det står “*vis*” eller “*forklar*” står du fritt i hvordan du svarer, men bruk enkle kodelinjer og/eller korte tekst-forklaringer og vær kort og presis.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger du finner nødvendig.

All kode skal være i C++. Det er ikke viktig å huske helt korrekt syntaks for bibliotekfunksjoner. Oppgaven krever ikke kjennskap til andre klasser og funksjoner enn de du har blitt godt kjent med i øvingsopplegget. Det er ikke nødvendig å bruke navnerom, ha med include-statement eller vise hvordan koden skal lagres i filer.

Hele oppgavesettet er arbeidskrevende og det er ikke forventet at alle skal klare alt. Tenk strategisk i forhold til ditt nivå og dine ambisjoner! Husk at tid du bruker på å lete i boka gir deg mindre tid til å svare på oppgaver. Det er ikke noen systematisk sammenheng mellom vanskelighetsgrad og nummerering av deloppgavene.

Hoveddelene av eksamensoppgaven teller med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan/vil likevel bli justert ved sensur basert på hvordan oppgavene har fungert. De enkelte deloppgaver vil bli tillagt forskjellig vekt basert på vanskelighetsgrad eller arbeidsmengde, men generelt er det antall riktig besvarte oppgaver som vil bestemme karakteren din.

Oppgave 1: Implementasjon av en klasse (45%)

Et rasjonalt tall er et tall som kan skrives som en brøk hvor teller og nevner er heltall. Koden under viser starten på en klasse for rasjonale tall. I deloppgavene under skal du implementere medlemsfunksjoner og -operatører for denne klassen samt svare på noen teorispørsmål.

```
class Rational{
private:
    int n;           //numerator (norsk: teller)
    int d;           //denominator (norsk: nevner)
    void reduce();  //reduce the fraction (norsk: forkorte brøken)
    int gcd(int a, int b); //finds the greatest common divisor
public:
    Rational();     //creates a value of 0/1
    Rational(int n, int d); //creates a value of n/d
    int numerator(); //returns numerator
    int denominator(); //returns denominator
    double to_double(); //returns fraction as double value
};

int Rational::gcd(int a, int b){
    //Euclids Algorithm - finds the greatest common divisor for a and b
    if ( b == 0 ){
        return a;
    }
    return gcd(b,a%b);
}
```

a) Implementer medlemsfunksjonen **reduce()**. Denne skal forkorte/forenkle **Rational**-instansen (brøken) den kalles på og er en hjelpefunksjon som eksempelvis skal brukes i konstruktøren eller operatører som endrer tilstanden på objektet. Et kall til **reduce()** skal resultere i følgende:

1) Brøken skal forkortes med hjelp av **gcd()**-funksjonen.

*gcd-funksjonen er en implementasjon av Euclids algoritme som gir deg høyeste fellesdivisor for to tall. Gitt et objekt med **n** = 2 og **d** = 4 så skal du bruke gcd til å finne høyeste fellesdivisor for disse tallene. Denne verdien brukes deretter til å forkorte medlemmene til **n** = 1 og **d** = 2.*

2) Funksjonen skal “rydde opp” i bruken av fortegn. Vi vil ha brøker hvor teller (**n**) kan være enten positiv eller negativt tall mens nevner (**d**) alltid skal være et positivt tall.

*Hvis **n** = 2 og **d** = -3 skal et kall til reduce() endre verdiene til **n** = -2 og **d** = 3.*

*Hvis **n** = -2 og **d** = -2 skal et kall til reduce() endre verdiene til **n** = 2 og **d** = 3.*

```
//Medlemsfunksjon som endrer på medlemsvariabler
//Vi ser etter at du har forstått oppgaven, og har gre løsning for 1) og 2)
void Rational::reduce() {
    //Finner høyeste fellesdivisor
    int divisor = gcd(n, d);
    //Reduserer brøken
    if (divisor != 1){
        n /= divisor; //n = n / divisor
        d /= divisor;
    }
    //sørger for å fjerne negativ nevner
    if (d < 0){
        n = -n; //alternativer n = n * -1; n *= -1;
        d = -d;
    }
    //Vi har her valgt å finne gcd først og deretter fjerne negativ nevner.
    //Da spiller det ingen rolle om gcd kalles med negative argument og gir
    //negativt svar
}
```

- b) Implementer konstruktøren **Rational(int n, int d)**. Her skal du selvsagt bruke **reduce()**.
Det er opp til den som bruker denne klassen å forhindre 0 i nevner.

```
//Konstruktør til Rational
Rational::Rational(int n, int d): n(n), d(d){
    reduce();
}
//Her er det ikke krav om bruk av initialiseringsliste,
//men da må du sette verdiene før du kaller reduce()
```

- c) Implementer medlemsfunksjonen **double to_double()**. Funksjonen skal returnere en flyttallsrepresentasjon av brøken. For verdien **a(1,2)** skal **a.to_double()** returnere verdien **0.5**.

```
//Viktig å sørge for flyttallsdivisjon
//da må en av operandene være double
double Rational::to_double() {
    return static_cast<double>(n) / d;
}
//Også greit å bruke C-style casting (double)n/d
```

- d) Implementer “*mindre enn*”-operatoren som medlem av klassen.

```
//Rett-frem-matematikk. Her er det selvsagt meningen å overlagre operator<
//med Rational som høyre og venstre operand.
//Generelt legger vi ikke vekt på const og call-by-reference i denne oppgaven
bool Rational::operator<(const Rational& rhs) const{
    return (this->n * rhs.d) < (rhs.n * this->d);
    //forsåvidt greit å bruke to_double() < rhs.to_double()
    //Mange har nok gjort det pga. sekvensen oppgavene var gitt i
}
```

- e) Vis hvordan “*mindre enn*”-operatoren kan brukes for å implementere “*er lik*”-operatoren.

```
//Her er det nok på vise innholdet i implementasjonen og ikke behov for å ha /
//med funksjonsheaderen.
//Vi ser etter riktig bruk av ! og &&
//også greit å løse med if-else
bool Rational::operator==(const Rational& rhs) const{
    return !(*this < rhs) && !(rhs < *this);
}
//Alternativt:
return !((*this < rhs) || (rhs < *this));
//Det viktigste her er at du gjenbraker operator< og har to
//sammenligninger hvor this og rhs bytter plass.
//Korrekthet i boolsk sammenligning er litt underordnet fordi dette er noe du
//vil debugge og prøve ut mens du programmerer.
```

- f) Implementer postfiks inkrementeringsoperator som medlem av klassen.

*Inkrementering skal øke verdien av brøken med 1. Hvis vi har objekt **a** med verdien $1/3$ (dvs. $n = 1$ og $d = 3$) og gjør inkrementeringen **a++**; skal verdien etterpå være $4/3$ (dvs. $n = 4$ og $d = 3$) fordi vi skal legge til $3/3$ som er det samme som 1. Husk at returverdi fra postfiks-operator skal være verdien objektet hadde før inkrementering.*

```
//Ved postfiks inkrementering skal du returnere verdien objektet hadde
//FØR inkrementeringen. Viktig med bruk av Rational temp og riktig
//inkrementering av teller. Dummy-parameteren er mindre viktig på eksamen.
Rational Rational::operator++(int){ //postfix
    Rational temp = *this; //husk bruk av temp
    n += d;
    return temp;
```

```
}
```

- g) Implementer som medlemmer av klassen de aritmetiske operatorene som er nødvendige for å kunne skrive `a *= b * c`; hvor `a`, `b` og `c` er objekter av typen `Rational`.

```
//Hvis vi implemeneterer operator*== først kan vi gjenbruke denne
//i operarator*. For å få fullt score må du huske å bruke reduce()
//Siden det er impl. som medlemsoperatore må vi returnere *this
//Sjekkliste: reduce(), return *this, fornuftig/enkel impl, gjenbruk
```

```
Rational Rational::operator*==(const Rational &rhs){
    n *= rhs.n;
    d *= rhs.d;
    reduce();
    return *this;
}
```

```
Rational Rational::operator*(const Rational &rhs){
    return Rational(*this) *= rhs;
}
```

- h) Er det nødvendig å implementere tilordningsoperator og/eller kopikonstruktør for denne klassen? Forklar kort hvorfor/hvorfor ikke.

Nei, ikke nødvendig fordi vi ikke har dynamisk allokert minne og de default genererte implementasjonene av kopikonstruktør og tilordningsoperator gjør det vi trenger (medlemsvis kopiering).

- i) C++ har forskjellige heltallstyper som `short`, `int`, `long` og disse kan være `signed` eller `unsigned`. Tilsvarende finnes det forskjellige datatyper for flyttall som `float` og `double`.
- 1) Deklarer `Rational` som template-klasse hvor talltype for teller og nevner bestemmes ved bruk.
 - 2) Ser du noen mulig fallgruve ved å lage en slik templateklasse? Begrunn svaret.

```
//Viser bare det som må endres:
//Medlemsvariablene, argumentene til konstruktøren, returtypen
template <typename T>
class Rational{
private:
    T n;
    T d;
public:
    Rational();
    Rational(T n, T d);
    T numerator();
    T denominator();
};
```

Ingen direkte fallgruver så lenge typen du bruker støtter operatorene som benyttes i klassen vil det jo kompilere. Det kan likevel diskuteres om det er hensiktsmessig å gjøre det mulig å lage en `Rational`-type hvor teller og nevner er flyttall siden dette strider mot def. av `Rational` - og det er relevant å spørre om det i det hele tatt er relevant å lage en slik templateklasse. Svaret er riktig hvis du minimum har `T` som vist over og gir en begrunnelse som viser evne til refleksjon.

- j) Hvordan må funksjonen `gcd()` være deklartert hvis den skal være del av klassen men også kunne brukes uten at du har en instans av `Rational`?

Vi er ute etter muligheten for å kunne skrive: `int dvs = Rational::gcd(8, 10)`;

Må være `public` og `static`

- k) I konstruktøren `Rational(int n, int d)` kunne vi kastet unntak av typen `invalid_argument` hvis nevner ble forsøkt satt til 0. Ville dette vært en hensiktsmessig og tilstrekkelig løsning for å forhindre kjøretidsfeil forårsaket av deling med 0? Ta utgangspunkt i en mer fullstendig implementasjon av `Rational` som inkluderer alle aritmetiske operasjoner. Begrunn svaret ditt.

Både ja og nei er greie svar så lenge du har et godt resonnement. Hovedpoenget er at det er viktig/lurt å være konsekvent.

JA, hvis du også sørger for at det kastes unntak andre steder hvor 0 i nevner kan oppstå (typisk i `operator/` eller `operator/=` hvor vi deler med en brøk som er 0).

NEI, fordi det er utilstrekkelig å bare gjøre dette i konstruktøren da det kan oppstå andre situasjoner hvor vi ender opp med 0 i nevner (typisk deling med 0)

- l) Gitt at du ønsker at klassen skal støtte følgende form for tilordning:

```
Rational r;  
r.denominator() = 5;
```

Hvordan kan du implementere dette kun ved å endre litt på funksjons-deklarasjonen?

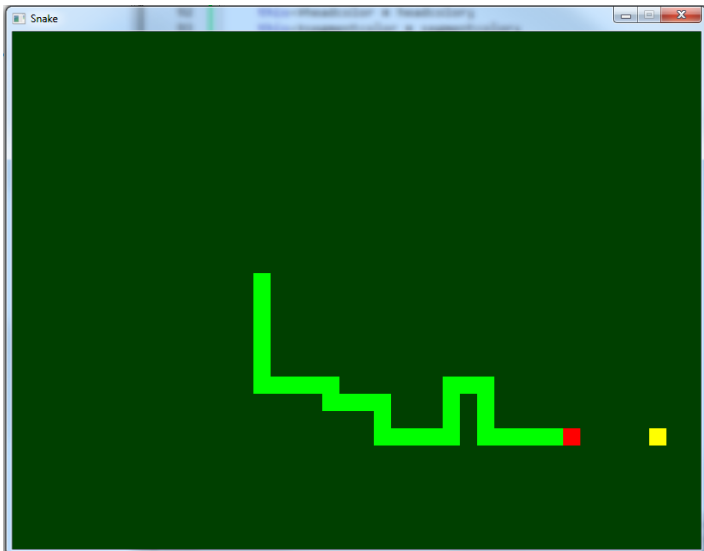
PS! dette er et rent teorispørsmål og ikke noe som ville vært naturlig å ha i denne klassen.

```
//Da må vi bruke return-by-reference  
int& denominator();
```

Oppgave 2: Snake-spillet (45%)

I denne oppgaven skal du implementere deler av et enkelt grafisk spill som er kjent under navnet **Snake**. I spillet kontrollerer bruker med piltastene en “slange” som beveger seg rundt på skjermen og “spiser” biter som dukker opp på tilfeldige plasser. Hver gang en bit er spist øker lengden på slangen. Hvis slangen treffer kanten av vinduet eller krysser seg selv er spillet over.

Bildet viser hvordan spillet kan se ut. Den gule firkanten i bildet er biten som slangen jakter på. Hodet til slangen er markert med rød firkant og resten av slangekroppen er lys grønn. Når du forandrer retningen på slangen vil resten av kroppen følge etter i samme spor slik at du får en slange med mange vinkler. Etter hvert som slangen blir lengre blir det vanskeligere å forhindre at slangen krysser seg selv eller at du treffer kanten.



I denne oppgaven er utfordringen å sette seg inn i et litt komplekst problem og lage kode som er sentral i spillet. Koden er delvis basert på SFML-biblioteket for grafikk og multimedia, men det er ikke nødvendig å kjenne dette biblioteket og vi har forenklet for å gjøre det lettere å forstå og skrive kode

I vedlegget finner du eksempelkode og forklaringer. Start med å sette deg inn i hvordan `main()` er bygd opp.

I vedlegget finner du også den fulle deklarasjonen av funksjonene du skal implementere i deloppgavene.

a) Implementer funksjonen `initSnake()`.

Funksjonen skal opprette en slange bestående av 4 rektangler plassert omtrent midt vinduet.

En slange er i praksis en samling rektangler plassert side ved side og i eksempelkode er det brukt en `vector<Rectangle>` som datatype for slangen. `Rectangle`-klassen har funksjonen `setPosition()` som brukes for å sette rektangelets posisjon i vinduet og `get`-funksjoner for å lese posisjon. Du finner eksempel på bruk av disse i vedlegget.

```
void initSnake(std::list<Rectangle>& snake){
    //Finner startposisjon (trenger bare være ca.), bruker list<> i stedet
    //for vector<>, men vi kunne like gjerne brukt vector
    int startposx = (WIN_WIDTH / 2) - (2 * RECT_WIDTH);
    int startposy = (WIN_HEIGHT / 2);
    //Legger inn 4 rektangler som er posisjoner etter hverandre
    //Her kan vi bruke teller * RECT_WIDTH for å finne x pos til rektangel
    //y pos blir samme verdi for alle rektangler
    for (int i = 0; i < 4; i++){
        sf::RectangleShape rectangle;
        rectangle.setSize(RECT_WIDTH, RECT_HEIGHT);
        rectangle.setPosition(startposx + RECT_WIDTH * i,
                             startposy + RECT_HEIGHT);
        snake.push_front(rectangle);
    }
    //om vi bruker push_front eller push_back spiller ingen rolle akkurat her
}
```

b) Les vedlegget og sett deg inn i hvordan slangens bevegelse skal implementeres.

- 1) Bibliotekstypen `vector<>` er egentlig dårlig egnet til vårt formål. Hvorfor?
- 2) Hvilken annen container-type i biblioteket bør du heller bruke?

Du trenger ikke gjøre om løsningen i 2a) selv om du foreslår annen datatype.

Her skal vi legge til i en ende og fjerne rektangler i andre enden. Da er vector lite egnet fordi det er kostbart å fjerne eller legge til f.eks. i starten av en vector siden alle objekter som skifter plass i realiteten må kopieres. Det vi trenger er en `list<Rectangle>` eller `deque<Rectangle>`. Disse har begge funksjoner for å legge til og fjerne i begge ender og er basert på lenkede lister hvor slike operasjoner er effektive. Mer effektive og enklere å programmere med.

- c) Implementer funksjonen `moveSnake()`. Ett kall til denne funksjon skal flytte slangen ett trinn i retningen som er angitt med `direction`. Anta at `direction` bestandig har en gyldig verdi.

Du trenger ikke ta hensyn til om slangen treffer kanten av vinduet og du står fritt til å erstatte vector med annen type i tråd med svaret du ga tidligere.

```
void moveSnake(list<Rectangle> &snake, Direction d){
    //Oppretter nytt rektangel
    Rectangle rectangle;
    rectangle.setSize(RECT_WIDTH, RECT_HEIGHT);

    //Finner posisjonen til hode på slangen
    //Vi bruker begin() som returnerer iterator (derav bruken av ->),
    //men det kan og gjøres med front() som returnerer referanse
    int posx = snake.begin()->getx(); //alternativ snake.front().getx()
    int posy = snake.begin()->gety(); //alternativ snake.front().gety()

    //Setter posisjonen til nytt rektangel foran eksisterende hode
    //og i forhold til retning (oppgaver spes. at direction er gyldig verdi og
    //derfor trenger vi ikke sjekke for om slangen snut 180 grader)
    //Her er det nok å vise to retninger slik at prinsippet er demonstrert
    if (d == DOWN){
        rectangle.setPosition(posx, posy + RECT_HEIGHT);
    }
    if (d == UP){
        rectangle.setPosition(posx, posy - RECT_HEIGHT);
    }
    if (d == LEFT){
        rectangle.setPosition(posx - RECT_WIDTH, posy);
    }
    if (d == RIGHT){
        rectangle.setPosition(posx + RECT_WIDTH, posy);
    }
    //legger til nytt rektangel først i lista, sletter sist i lista
    snake.push_front(rectangle);
    snake.pop_back();
}
```

- d) Implementer funksjonen `detectCollision()` som brukes for å sjekke om to rektangler har “kollidert” med hverandre. Kollisjon betyr i praksis at to rektangler overlapper.

Våre rektangler er orienterte langs x og y aksene i vinduet og trengs kun enkel testing som forklart i vedlegget.

```
//Her var det litt forvirring mht. navn på funksjonen, men det vi ser etter er
//at du har implementert den og seinere får til å bruke den
bool detectCollision(const Rectangle &a, const Rectangle &b){
    //Rett frem impl av løsningen skissert i vedlegget - husk &&
    return (a.getx() < b.getx() + RECT_WIDTH &&
            a.getx() + RECT_WIDTH > b.getx() &&
            a.gety() < b.gety() + RECT_HEIGHT &&
            RECT_HEIGHT + a.gety() > b.gety());
}
```


- e) Implementer funksjonen `placePiece()`. `Rectangle`-variabelen `piece` er den “biten” som du skal styre slangen mot. Funksjonen skal sette `piece`-variabelens posisjon til en tilfeldig plass innenfor vinduet, men skal også sørge for at biten ikke plasseres på slangen.

```
void placePiece(RectangleShape &piece, const list<RectangleShape> &snake){
    bool overlap = true; //så løkka kjører minst en gang
    //løkke som kjører helt til piece ikke overlapper med slangen
    while(overlap){
        //starter med å anta at piece er plassert riktig;
        //etterpå sjekker vi om piece overlapper med slangen
        overlap = false;
        piece.setPosition(rand() % (WIN_WIDTH - RECT_WIDTH),
                          rand() % (WIN_HEIGHT - RECT_HEIGHT));
        //tester mot slangen og bruker break hvis vi oppdager kollisjon
        for (RectangleShape const &r: snake){
            if (detectCollision(r, piece)){
                overlap = true;
                break;
            }
        }
    }
}
```

- f) Vis/forklar hvordan og hvor du i `main` vil bruke `detectCollision()` til å teste om slangen har “spist biten”. Dersom biten er spist skal den gies ny tilfeldig posisjon. I praksis skal du teste om slangens “hode” overlapper med biten (piece-rektangelen). Det vi kaller slangens hode er egentlig bare det kvadratet som vi regner som slangens “forreste” rektangel. Om dette er første eller siste element i kontaineren avhenger av rekkefølgen du har lagret rektangelene i. I spillet vil det se ut som slangen spiser en bit og en ny bit dukker opp et annet sted. I selve programmet kan vi bruke en og samme variabel for biten.

```
//Dette gjør vi inne i if-blokka, før eller etter at vi flytter slangen
//Utenfor if-blokka er syntaktisk ok, men vil medføre mye unødv. testing
if (timer.getElapsedTime().asMilliseconds() >= 100){
    moveSnake(snake, direction);
    timer.restart();
    //vi har basert oss på at hode er front()
    if (detectCollision(snake.front(), piece)){
        placePiece(piece, snake);
    }
}
```

- g) Dersom en bit er spist skal slangens lengde økes med ett rektangel. Vis og forklar hvordan du vil implementere dette. *Tips: Det kan løses ved å endre litt på en av de tidligere funksjonene vi har laget. Lengden skal økes, men du trenger kanskje ikke å legge til et nytt rektangel på enden?*

```
//En enkel løsning på dette er å legge til en bools variabel til i move() som
//forteller om slangen skal øke lengde eller ikke. Hvis false skal bevegelse
//være som før, hvis true kan vi øke lengden ved å IKKE fjerne et element (siden
//en move-operasjon innebærer både å legge til og fjerne).
//I tillegg må vi endre litt i main slik at move blir kalt med forskjellig verdi
//for extend basert på utfallet fra detectCollision.
```

```

void moveSnake(list<Rectangle> &snake, Direction d, bool extend) {
    //som forrige versjon, men følgende er forskjellig:
    snake.push_front(rectangle);
    //fjerner fra slutten bare hvis extend er true
    if (!extend) {
        snake.pop_back();
    }
}

//I main kan vi endre til dette:
if (timer.getElapsedTime().asMilliseconds() >= 100) {
    if (detectCollision(snake.front(), piece)) {
        moveSnake(snake, direction, true);
        placePiece(piece, snake);
    } else {
        moveSnake(snake, direction, false);
    }
    timer.restart(); //reset/restart the timer
}

```

- h)** Lag en ny funksjon **selfCollide()** for å teste om slangen kolliderer med seg selv. Dette skjer hvis slangens “hode” kolliderer med et av de andre rektanglene i slangens “kropp”. Pass på så du ikke tester om hodet overlapper med seg selv. Her bestemmer du selv returtype og parameterliste.

```

bool detectSelfCollision(const list<Rectangle> &snake) {
    //Her jukser vi litt og bruker auto
    //Det viktigste er at du bruker iterator og iterere over alle elementer
    //samt unnlater å kollisjonsteste hode mot seg selv.
    auto first = snake.begin(); //iterator to first
    for (auto it = snake.begin(); it != snake.end(); ++it) {
        //husk å sjekke at vi ikke sammenliger hode mot hode
        if (first != it && detectCollision(*first, *it)) {
            return true;
        }
    }
    return false;
}

```

i) En litt friere oppgave

Koden i vedlegget gir deg nok informasjon til å bli kjent med klassen **Rectangle** (hvilke public funksjoner den har og hva disse gjør). I denne oppgaven skal du lage en subklasse av **Rectangle** som vi kaller **MovingRectangle**. Denne skal være litt mer spesialisert og objektene av denne typen har en retning og en fart, endrer retning hvis de treffer kanten av vinduet o.l. Vi skal bruke denne datatypen for bitene slangen jakter på og ønsker biter som beveger seg rundt på skjermen og gjør slangens jakt litt mer utfordrende.

Krav til klassen er som følger:

- 1) Klassen skal arve fra **Rectangle**.
- 2) I svaret skal du gjengi en klassedeklarasjonen av **MovingRectangle** samt lage implementasjon av en **konstruktør** og medlemsfunksjonene **move()** og **place()**. Returtype og evt. parameterliste bestemmer du selv i denne oppgaven.
- 3) Klassen skal ha en konstruktør som oppretter **MovingRectangle**-objekter med en tilfeldig posisjon innenfor et vindu, samt gir objektene en tilfeldig fart og retning.
- 4) Klassen skal ha medlemsfunksjonen **place()** som flytter objektet til en ny tilfeldig posisjon innenfor vinduet og setter fart og retning. Denne kan brukes av konstruktøren og kalles når slan-

gen har spist “biten”. Denne funksjonen blir ganske lik `placePiece()` fra 2e), men her trenger du ikke ta hensyn til hvor slangen er.

- 5) Klassen skal ha medlemsfunksjonen `move()` som flytter objektet ett steg i retningen som er satt for objektet. Gitt objektet `MovingRectangle piece` er ideen at vi i `main` gjør et kall til `piece.move()` i hver iterasjon av løkken. Hvis objektet treffer en av kantene på vinduet skal det endre retning slik at det beveger seg bort fra kanten.
- 6) Koden din skal først og fremst demonstrere teknikken som brukes i klassen for å styre bevegelsen til en bit, men må kunne generaliseres/utvides til en mer fullstendig løsning.
- 7) Se i vedlegget for eksempel på hvordan du kontrollerer en bevegelse.

NB! Hele del 2i) kan løses med relativt lite kode.

```
//hele klassen kan gjøre så enkelt som dette
//husk arv
class MovingRectangle : public Rectangle{
private:
    double speedx;
    double speedy;
    int dx = -1;
    int dy = -1;
public:
    MovingRectangle(){
        place();
    }
    void place(){
        double posx = rand() % (WIN_WIDTH - RECT_WIDTH);
        double posy = rand() % (WIN_HEIGHT - RECT_HEIGHT);
        //kaller medlemsfunksjon vi arver
        setPosition(posx, posy);
        //setter tilfeldig fart og retning
        speedx = ((rand() % 5) + 1) / 10.0; // gir tall mellom 0,1 og 0,5
        speedy = ((rand() % 5) + 1) / 10.0; // et tilfeldig tall er ok svar
        //sjekker at vi ikke får 0 fart i begge retninger samtidig
        do{
            dx = (rand() % 3) - 1; // gir verdier mellom -1 og 1
            dy = (rand() % 3) - 1;
        }while(dx == 0 && dy == 0);
        //kaller medlemsfunksjon vi arver
        setSize(RECT_WIDTH, RECT_HEIGHT);
    }

    void move(){
        //flytter objektet
        double posx = getx() + (speedx * dx);
        double posy = gety() + (speedy * dy);
        //tester for om vi kolliderer med en av sidene og snur retning
        if (posx <= 0 || posx >= (WIN_WIDTH - RECT_WIDTH)){
            dx *= -1; //reversing direction
        }
        if (posy <= 0 || posy >= (WIN_HEIGHT - RECT_HEIGHT)){
            dy = dy * -1; //reversing direction
        }
        //setter posisjon med medlemsfunksjon vi arver
        setPosition(posx, posy);
    }
};
```

Oppgave 3: For deg som har tid igjen (10%)

I klassen `TwoDArrayImpl` allokterer konstruktøren minne til radene i en to-dimensjonal tabell og destruktøren tar seg av å friggi dette minnet. `TwoDArrayImpl` er subclasse av `TwoDArrayBase`.

```
class TwoDArrayBase {
public:
    TwoDArrayBase();
    ~TwoDArrayBase();
};

class TwoDArrayImpl : public TwoDArrayBase{
//Implementation of run-time allocated two-dimensional array
//using pointer to pointer technique
private:
    double **arr;
    int rows, columns;

public:
    TwoDArrayImpl( int rows, int columns) : arr(nullptr), rows(rows), columns(columns){
        //all pointers are initialized as nullptr during allocation
        arr = new double*[rows]{};
        for ( int i = 0; i < rows; i++ ) {
            arr[i] = new double[columns]{};
        }
    }
    ~TwoDArrayImpl() {
        for (int i = 0; i < rows; i++) {
            delete [] arr[i];
        }
        delete [] arr;
    }
};

int main() {
    try{
        TwoDArrayImpl *testarr = new TwoDArrayImpl(100, 50000);
        delete testarr;
    }catch(bad_alloc &exc){
        cout << "Error allocating memory" << endl;
    }
}
```

- a) I konstruktøren til `TwoDArrayImpl` har vi et problem. Dersom minneallokering med `new` feiler vil den kaste unntak av typen `bad_alloc` og her fanger vi dette unntaket i `main`. Problemet er at du kan gå tom for minne underveis i allokeringen og da vil konstruktøren avsluttes med delvis allokert minne, programmet hopper direkte til catch-blokka i `main` og minne blir ikke frigitt siden programmet hopper over setningen med `delete testarr`.

Vis hvordan du i konstruktøren kan fange opp `bad_alloc`-unntak ved minneallokeringsfeil, friggi det som er allokert og videresender unntaket slik at vi også kan fange dette i `main`.

Her får dere løsningen gratis siden minnehåndteringen allerede står i destruktøren. Trenger bare bruke try-catch i konstruktøren og gjøre litt testing så vi ikke starer å slette minne som ikke er allokert

```
TwoDArrayImpl( int rows, int columns) :
    arr(nullptr), rows(rows), columns(columns) {
    try{
        arr = new double*[rows]{};
        for ( int i = 0; i < rows; i++ ) {
            arr[i] = new double[columns]{};
        }
    }catch(bad_alloc &exc){
        if (arr != nullptr){
```

```

//tester at vi har fått allokert radpeker-arrayet
for (int i = 0; i < rows; i++) {
//avslutter ved nullptr, mest for effektivitetens skyld
//denne testen er ikke viktig
    if (arr[i] == nullptr){
        break;
    }
    //frigjør radene
    delete [] arr[i];
}
}
delete [] arr;
throw;
//eller throw exc; sørger for at unntaket "kastes videre"
}
}

```

- b) Som koden over viser er `TwoDArrayImpl` en subklasse av `TwoDArrayBase`. Når du deklarerer variabelen `testarr` som peker til superklassen, som vist i koden under, oppdager du at subklassen sin destruktør aldri blir kalt (uavhengig av om det kastes unntak). Hva er typisk årsak til denne feilen?

```

int main() {
    try{
        TwoDArrayBase *testarr = new TwoDArrayImpl(100, 50000);
        delete testarr;
    }catch(std::bad_alloc &exc){
        cout << "Error allocating memory" << endl;
    }
}

```

Denne feilen tyder på at det er superklassens destruktør som kalles - siden det garantert er en destruktør som blir kalt. Da er eneste mulige feil at destruktøren i superklassen ikke er deklarerert som virtual

Vedlegg

Snake main-function with game-loop

Et Window-objekt representerer vinduet for spillet og en while-løkke brukes for å skape et levende spill. For hver iterasjon av løkka visker du ut forrige bilde og tegner opp ett nytt bilde på skjermen. Ved å tegne de samme objektene på nye posisjoner i vinduet skaper du inntrykk av bevegelse.

```
const int RECT_WIDTH = 10;
const int RECT_HEIGHT = 10;
const int WIN_WIDTH = 1200;
const int WIN_HEIGHT = 800;
//We are using constants for size of rectangles and size of window
//We do not set any specific colors and rely on default
//which is black background and white shapes

enum Direction {UP, DOWN, LEFT, RIGHT};

//Functions explained in the assignments
void initSnake(vector<Rectangle> &snake);
void placePiece(Rectangle &piece, const vector<Rectangle> &snake);
void moveSnake(vector<Rectangle> &snake, Direction direction);
bool detectCollision(const Rectangle &a, const Rectangle &b);

//getDirection retrieves information from the keyboard and updates the direction
//variable according to pressed arrow key and valid change of direction
void getDirection(Window &window, Direction &direction);

int main(){
    srand(time(0)); //Seeding rand() which is used by the functions
    Clock timer; //A timer is comparable to a stop watch
    Direction direction = RIGHT; //Set an initial movement direction of the snake

    Window window(WIN_WIDTH, WIN_HEIGHT, "snake"); // GUI window

    vector<Rectangle> snake; //Our snake is a collection of rectangles
    initSnake(snake); //initialized with 4 adjacent rectangles

    Rectangle piece; //The objekt that the snake is hunting
    piece.setSize(RECT_WIDTH, RECT_HEIGHT); //Setting size and position of piece
    placePiece(snake, piece);

    while (window.isOpen()){ //game loop that runs until window is closed

        getDirection(window, direction); //Updates the direction

        window.clear(); //Clears the screen

        if (timer.getElapsedMilliseconds() >= 100){ //Check if 0.1 ms has passed
            moveSnake(snake, direction);
            timer.restart(); //Reset/restart the timer
        }

        for (Rectangle &s: snake){
            window.draw(s); //Draw each rectangle of the snake on screen
        }
        window.draw(piece); //Draw the piece rectangle

        window.display(); //Display the screen
    }
}
```

Placing rectangles next to each other

```
//Example code for drawing rectangles side by side along x-axis
//This code shows all Rectangle-functions you need.
//Coordinate (0, 0) is upper left corner of window.
//The position of a rectangle is its upper left corner.

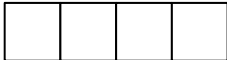
const int size = 10;           // example rectangle size
Rectangle first;
Rectangle second;

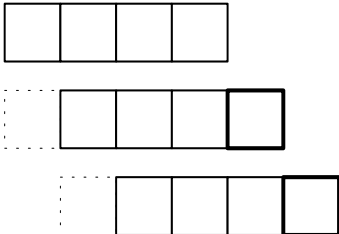
first.setSize(size, size); // set width and height
first.setPosition(250, 250); // set position of first rectangle on screen

second.setSize(size, size);
second.setPosition(first.getX() + size, first.getY());

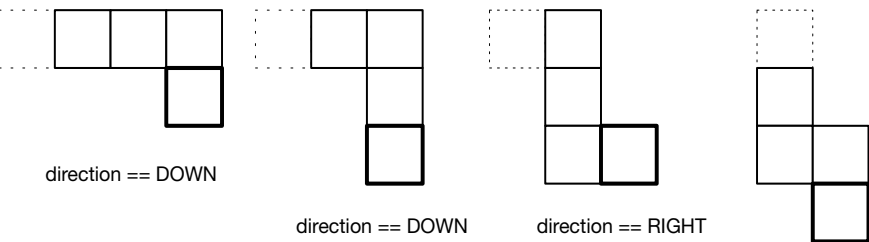
window.draw(second);
window.draw(first);
```

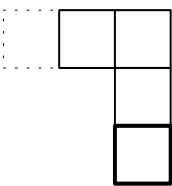
Moving the snake

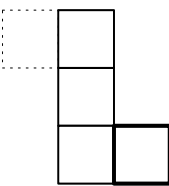
 Start with 4 equivalently sized rectangles placed side by side

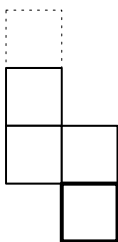
 Create movement by adding one new rectangle (thick line) at one end and remove one rectangle (dotted line) from the other end.

In this example we create a movement towards the right. Note that the snake-movement is not smooth but stepwise. Every operation moves the snake in a step equal to the size of the rectangle.

 direction == DOWN

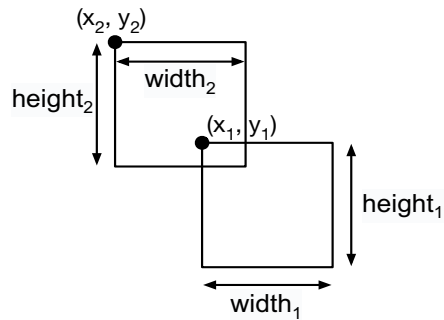
 direction == DOWN

 direction == RIGHT

 direction == DOWN

The placement of new rectangles decides how the snake moves and turns on the screen.

Collision detection



The following statements are all true if the rectangles overlap:

$$\begin{aligned}x_1 &< x_2 + \text{width}_2 \\x_1 + \text{width}_1 &> x_2 \\y_1 &< y_2 + \text{height}_2 \\ \text{height}_1 + y_1 &> y_2\end{aligned}$$

Controlling the movement of an object

```
//Example code to incrementally change the position of an object
//Speed and direction can also be managed using a single variable for velocity
void movementexample(){
    int dx = -1, dy = 1;           //moving down and towards left
    double speedx = 0.2, speedy = 0.2; //same speed along both axis
    double x = 500, y = 50;       //example starting coordinate
    int width = 1000, height = 1000; //window size
    while(x > 0 && x < width && y > 0 && y < height){
        //game-loop to move the coordinate until it hits the edges
        x = x + (speedx * dx);
        y = y + (speedy * dy);
        cout << x << ", " << y << endl;
    }
}
```


Sequence containers in C++ standard library

Headers		<code><array></code>	<code><vector></code>	<code><deque></code>	<code><forward_list></code>	<code><list></code>
Members		array	vector	deque	forward_list	list
	<i>constructor</i>	<i>implicit</i>	vector	deque	forward_list	list
	<i>destructor</i>	<i>implicit</i>	~vector	~deque	~forward_list	~list
	<i>operator=</i>	<i>implicit</i>	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin before_begin	begin
	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin		rbegin
	rend	rend	rend	rend		rend
const iterators	begin	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin		crbegin
	crend	crend	crend	crend		crend
capacity	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty
	resize		resize	resize	resize	resize
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
	capacity		capacity			
	reserve		reserve			
element access	front	front	front	front	front	front
	back	back	back	back		back
	operator[]	operator[]	operator[]	operator[]		
	at	at	at	at		
modifiers	assign		assign	assign	assign	assign
	emplace		emplace	emplace	emplace_after	emplace
	insert		insert	insert	insert_after	insert
	erase		erase	erase	erase_after	erase
	emplace_back		emplace_back	emplace_back		emplace_back
	push_back		push_back	push_back		push_back
	pop_back		pop_back	pop_back		pop_back
	emplace_front			emplace_front	emplace_front	emplace_front
	push_front			push_front	push_front	push_front
	pop_front			pop_front	pop_front	pop_front
	clear		clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap
	list operations	splice				splice_after
remove					remove	remove
remove_if					remove_if	remove_if
unique					unique	unique
merge					merge	merge
sort					sort	sort
reverse					reverse	reverse