Institutt for datateknikk
og informasjonsvitenskap

LF

TDT4102 - Prosedyre- og objektorientert programmering

TENTATIVT LØSNINGSFORSLAG MED FORBEHOLD OM FEIL
LF KAN BLI ENDRET ETTER HVERT SOM VI SER FORSKJELLIGE LØSNINGER SOM STUDENTENE HAR VALGT

Lørdag 19 mai 2012, 09:00

Kontaktperson under eksamen: Trond Aalberg (97631088)

Eksamensoppgaven er utarbeidet av Trond Aalberg
og kvalitetssikret av Hallvard Trætteberg

Språkform: Bokmål

Tillatte hjelpmidler (hjelpemiddelkode C):
Walter Savitch, Absolute C++ eller Lyle Loudon, C++ Pocket Reference.
Bestemt, enkel kalkulator tillatt.

Sensurfrist: Mandag 11 juni.

Generell introduksjon

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

Når det står “implementer” eller “lag” skal du skrive en implementasjon. Hvis det står “vis” eller “forklar” står du fritt i hvordan du svarer, men bruk enkle kodelinjer og/eller korte forklaringer og vær kort og presis. I noen oppgaver er det brukt nummererte linjer for koden slik at det skal være lett å referere til spesifikke linjer. Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig.

Hver enkelt oppgave er ikke ment å være mer omfattende enn det som er beskrevet. Noen oppgaver fokuserer bare på enkeltfunksjoner og da er det utelukkende denne funksjonen som er tema. Andre oppgaver er “oppskriftsbasert” og vi spør etter forskjellige deler av en klasse, samarbeidende klasser eller et program. Du kan velge selv om du vil løse dette trinnvis ved å ta del for del, eller om du vil lage en samlet implementasjon. Sørg for at det går tydelig frem hvilke spørsmål du har svart på hvor i koden din.

Det er ikke viktig å huske helt korrekt syntaks for biblioteksfunksjoner. Oppgaven krever ikke kjennskap til andre klasser og funksjoner enn de du har blitt kjent med i øvingsopplegget. All kode skal være i C++.

Hele oppgavesettet er arbeidskrevende og det er ikke foreventet at alle skal klare alt. Tenk strategisk i forhold til ditt nivå og dine ambisjoner! Velg ut deloppgaver som du tror du mestrer og løs disse først. Slå opp i boka kun i nødsfall. All tid du bruker på å lete i boka gir deg mindre tid til å svare på oppgaver. Deloppgavene i de “tematiske” oppgavene er organisert i en logisk rekkefølge, men det betyr IKKE at det er direkte sammenheng mellom vanskelighetsgrad og nummereringen av deloppgavene.

Oppgavene 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 kan også bli tillagt forskjellig vekt.

Oppgave 1: Kodeforståelse (20%)

a) Gitt følgende variabler og tilordninger

```
int a = 5;
int b = 5;
int *c = &a;
int *d = &b;
```

Hva blir resultatet av disse uttrykkene: true eller false?

```
a == b      //true
&a == c     //true
c == d      //false
*c == *d    //true
```

b) Hva skrives ut av følgende kode?

```
int a = 5.8 * 2;
cout << "a = " << a << endl;
//11
double b = 5 / 2;
cout << "b = " << b << endl;
//2
int x = 5;
int y = 10;
int c = x++ + ++y;
cout << "c = " << c << endl;
//16
```

I 1b trakk jeg bare to poeng for å svare 2.5 på andre utskrift, siden det var en feil som de fleste gjorde.

c) Klassen `Node` under brukes til å lage en dobbeltlenket liste. Den har en medlemsfunksjon `void insertAfter(Node *n)` som skal sette inn noden n på plassen etter objektet funksjonen kalles i kontekst av. Her blir det litt kluss med pekertilordningen, noe du oppdager når du prøver å iterere over alle nodene i en liste du har laget (uendelig løkke).

1) Vis hva som blir feil med en tegning av node- og pekerstrukturen for eksempelet i boksen under (bruk bokser for noder og piler for pekere).

2) Vis hvilke endringer som må gjøres i koden for å få riktig oppførsel.

```
1:  class Node {
2:  private:
3:     string value;
4:     Node* next;
5:     Node* prev;
6:  public:
7:     Node(const string &value): value(value), next(NULL), prev(NULL){}
8:     void insertAfter(Node* n);
9:     string getValue();
10: };
11:
12: void Node::insertAfter(Node *n){
13:    if (next == NULL){
14:       next = n;
15:       n->prev = this;
16:    }else{
17:       next = n;
18:       n->prev = this;
19:       next->prev = n;
20:       n->next = next;
21:    }
22: };
```

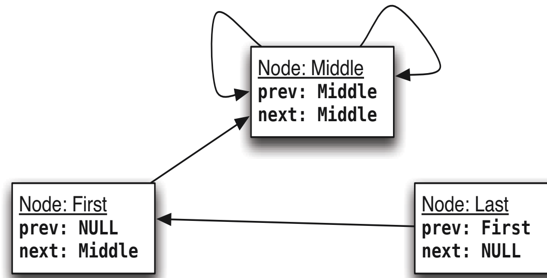
```
Node *first = new Node("First");
Node *middle = new Node("Middle");
Node *last = new Node("Last");

first->insertAfter(last);
first->insertAfter(middle);
```

```
void Node::insertAfter(Node *n){
  if (next != NULL){
    next->prev = n;
    n->next = next;
  }
  next = n;
  n->prev = this;
};
```

Et annet svar er å bare flytte next = n til slutten av else-blokka.

Illustrasjon av feilen.



Introduksjon til oppgavene 1d - 1h:

Bubblesort (boblesortering) er en velkjent algoritme for sortering av en sekvens av elementer (for eksempel verdiene i en array/tabell eller en vector). Den brukes fordi den er enkel å forstå og implementere, men er kun egnet til små sorteringsoppgaver siden den har dårlig ytelse.

I boblesortering brukes en nøstet løkke. I den innerste løkka itererer vi over alle elementer og hvis `arr[j] > arr[j+1]` så bytter vi om på disse to verdiene. Hvis vi starter med sekvensen `{5,4,3,2,1,0}`, vil første iterasjon av den innerste løkka føre til at **5** forflyttes til siste posisjon `{4,3,2,1,0,5}`. Ved andre iterasjon vil den nest høyste verdien (**4**) forflyttes til sin rette plass `{3,2,1,0,4,5}` etc. Algoritmen kalles boblesortering fordi de minste verdiene langsomt "bobler" fremover til sine rette posisjoner.

Funksjonen under er et første utkast til implementasjon av boblesortering for typen **`vector<int>`**. I de følgende delspørsmål skal du identifisere feil og endre koden.

```
1:   void bubblesort(vector<int> arr){
2:     for(int i = arr.size(); i > 0; --i){
3:       for (int j = 0; j < arr.size(); ++j){
4:         if (arr[j] > arr[j+1]){
5:           int temp = arr[j];
6:           arr[j] = arr[j+1];
7:           arr[j+1] = temp;
8:         }
9:       }
10:    }
11:  }
```

d) Hvilken C++ biblioteksfunksjon kunne du brukt i stedet for kodelinjene 5-7?

```
void swap(T &a, T &b); //men det holder at du sier swap
```

e) For å teste implementasjonen lager du en variabel **`vector<int> testvec`** i main, legger inn noen verdier ved hjelp av **`push_back`** og gjør kallet **`bubblesort(testvec)`**. Når du etterpå skriver ut verdiene i **`testvec`** til **`cout`**, ser det ut som **`testvec`** fortsatt er usortert! Hva er problemet og hva må endres for å rette denne feilen?

Må bruke call-by-reference (&-tegnet mangler for parameter arr)

```
1:   void bubblesort(vector<int> &arr){
```

f) Etter å ha rettet feilen i forrige deloppgave, prøver du å sortere en vector med innholdet `{5,4,3,2,1}`. Etter sortering er innholdet blitt `{0,1,2,3,4}`.
Hva er feil i implementasjonen og hvordan rette opp? Hint: hvor havnet tallet 5 etter første gjennomløp......

```
Vi swaper siste element med det som kommer etter siste element.
Her var det gitt hint om hvor det ble av tallet 5. At du får inn tallet
0 er mindre vesentlig - det er bare det tallet i minnet som du swapper med.
Kan enkelt fikses med å bruke size() - 1 eller på annen måte sørge for
at vi ikke swapper med en posisjon som er utenfor vectoren

2:     for(int i = arr.size(); i > 0; --i){
3:       for (int j = 0; j < arr.size() - 1; ++j){
```

g) Ytelsen til bubblesort kan forbedres ved å redusere antallet iterasjoner i den innerste løkka. Etter første gjennomløp (iterasjon) av den innerste løkka vil største verdi være plassert sist. Ved andre gjennomløp kan du derfor ignorere siste element. Ved tredje gjennomløp kan du ignorere de to siste elementene etc. Vis hvordan koden kan endres til å implementere denne forbedringen.

```
Vi kan iterere over ett element mindre for hvert gjennomløp av
innerste iterasjon. Se løsningen under. Siden vi teller ned i ytterste
løkke kan vi bruke telleren i som max-verdi i innerste løkke. Det var
ikke uten grunn at vi telte nedover i ytterste løkke. Det viktigste her
er at logikken er riktig,
```

h) En annen ytelsesforbedring kan oppnåes ved å avslutte når du vet at elementene er sortert. For en sekvens bestående av `{1,2,3,5,4}` vil implementasjonen gjøre mange unødvendige iterasjoner. Forklar hvordan du underveis kan teste om sekvensen er sortert og avslutte hvis det ikke er behov for flere iterasjoner.

```
Vi kan ha en bool som settes til true i innerste løkke hvis verdier blir
swappet. I tilfellet ingen swappes i løpet av en iterasjon er sekvensen
ferdig sortert.

void bubblesortS(vector<int> &arr){
  for(int i = arr.size() - 1; i > 0; --i){ //har flyttet - 1 hit
    bool swapped = false;
    for (int j = 0; j < i; ++j){
      if (arr[j] > arr[j+1]){
        swap(arr[j], arr[j + 1]);
        swapped = true;
      }
    }
    if (!swapped){
      return;
    }
  }
}
```

Oppgave 2: Funksjoner (20%)

a) Implementer en funksjon `int round(double d)` som runder av et flyttall til nærmeste hele tall (returnerer en `int`). Funksjonen skal også håndtere negative tall.

Eksempler: `round(3.45)` gir 3, `round(3.60)` gir 4, `round(3.50)` gir 4 (vi runder oppover ved .50), `round(-3.45)` gir `-3`, `round(-3.6)` gir `-4`.

Vi baserer oss på implisitt konvertering av flyttall til heltall, men trikser med +/- 0.5 for å få avrundet riktig: 2.6 + 0.5 = 3.1 som blir konvertert til heltallet 3. 2.4 + 0.5 = 2.9 som blir konvertert til heltallet 2. For negative tall må vi trekke fra. Her er det brukt if-else operatoren, men det er helt greit å bruke vanlig if-else setninger. Kan også løses med ceil og floor, men det er litt unødvendig.

```
int round(double x){
    return int(x >= 0.0 ? x + 0.5 : x - 0.5);
}
```

Her var det mange varianter og kun noen få som gjorde det på denne enkle måten. Det viktigste er at du har en logikk som runder av riktig, at du håndterer positive og negative tall og at du ikke bruker operatorer på en riv ruskende gal måte (f.eks. bruk av % modulo på flyttall)

b) Implementer en funksjon `int adjacent_find(int arr[], int first, int last)` som leter i tabellen `arr` etter naboelementer med lik verdi. Funksjonen skal returnere indeksen til første element av disse. Argumentene `first` og `last` angir området i tabellen den skal lete i. Funksjonen skal returnere `-1` hvis den ikke finner noen like naboelementer.

Eksempel: Gitt `int x[] = {0,4,4,0,3,3};` så skal `adjacent_find(x,0,6)` returnere `1` siden dette er indeksen til det første tallet i første par funksjonen finner.

```
int adjfind(int arr[], int first, int last){
    while (first < (last - 1)){
        if (arr[first] == arr[first + 1]){
            return first;
        }
        first++;
    }
    return -1;
}
```

Her skilte vi ikke mellom de som tolket last som siste posisjon og de som tolket last som posisjonen etter siste (som eksempelet viser).

c) Implementer en funksjon `void print_adjacent(int arr[], int size)` som bruker funksjonen fra oppgaven over og skriver ut til cout alle par av naboverdier som finnes i `arr`, hvor `size` er antallet elementer i tabellen.

Eks: Gitt `int x[] = {0,4,4,5,3,3,3,7}` så skal funksjonen skrive ut: 4 4 - 3 3 - 3 3. Merk at vi får skrevet ut 3 3 to ganger siden både posisjon 4 og 5 har et neste element som er samme verdi. Formatteringen av utskriften er uvesentlig i denne oppgaven.

```
//Her vises to veldig kompakte varianter, men mindre kompakte løsninger
//er OK. En gjennomgående feil var å ikke iterere riktig.
```

```
void print_adjacent(const int t[], int size){ //bruker while-løkke
    int pos = 0;
    while ((pos = adjfind(t, pos, size)) != -1){
        cout << t[pos] << "," << t[pos + 1] << " " << endl;
        pos++;
    }
}
void print_adjacent(const int t[], int size){ //med for-løkke
    for(int pos = 0; (pos = adjfind(t, pos, size)) != -1; pos++){
        cout << t[pos] << "," << t[pos + 1] << " " << endl;;
    }
}
//Mange løste oppgaven på denne måten:
//Det viktigste er at du fortsetter iterasjonen en posisjon etter
//den posisjonen du får returnert
void printAdjWhile2(const int arr[], int size){
    int a = adjfind(arr, 0, size);
    while (a != -1){
        cout << arr[a] << "-" << arr[a+1] << " ";
        a = adjfind(arr, a + 1, size);
    }
    cout << endl;
}
```

d) Digitale bilder er bygd opp av punkter i et jevnt mønster (se under). Når du skal tegne opp (plotte) en linje fra et startpunktet **`(x1,y1)`** til et sluttpunktet **`(x2,y2)`** må du regne ut hvilke punkter som skal tegnes opp mellom disse to punktene. Linja du tegner opp vil bli en tilnærming, men siden punktene er små og tett plasserte vil det se ut som ei jevn linje.

En enkel (og ikke så veldig effektiv) algoritme for dette baserer seg på den velkjente formelen for linjer: **`y = a*x + b`** hvor **`a`** er stigningstallet **`(y2-y1)/(x2-x1)`** og **`b`** er linjas skjæringspunkt på y-aksen. Tallet **`b`** (skjæringspunktet) kan regnes ut ved å sette inn en av koordinatene som er input til funksjonen. Figuren under viser 3 forskjellige linjer og punktene som algoritmen har regnet ut for disse.

1) Implementer funksjonen **`void plotline(int x1,int y1,int x2,int y2)`** som skriver til cout alle x,y-punktene som skal plottes for linja fra **`(x1,y1)`** til **`(x2,y2)`**. Du kan iterere over x og regne ut y, men husk at x og y til et punkt skal være det koordinat som er nærmest den ideelle linja. Det holder om løsningen dekker stigningstall fra 0 til 1.

```
//Dette holder som svar på denne oppgaven. Vi er ute etter basislogikken,
//de forskjellige "spesialtilfellene" er tema for deloppgavene under
void plotline(int x1, int y1, int x2, int y2){
    double dx = x2-x1;     // bruk datatypen double her eller casting
    douyble dy = y2-y1;    // i divisjonen under
    double a = dy / dx; // hvis dy og dx er int bruker du dy/(double)dx
    double b = y1 - (a * x1);
    for (int x = x1; x <= x2; ++x){
        int y = round((a * x) + b); //husk avrunding
        cout << "(" << x << "," << y << ")" << endl;
    }
}
```

2) Hva skjer hvis stigningstallet til linja er større enn 1?
Forklar med kode eller tekst hvordan du kan håndtere dette i funksjonen din.

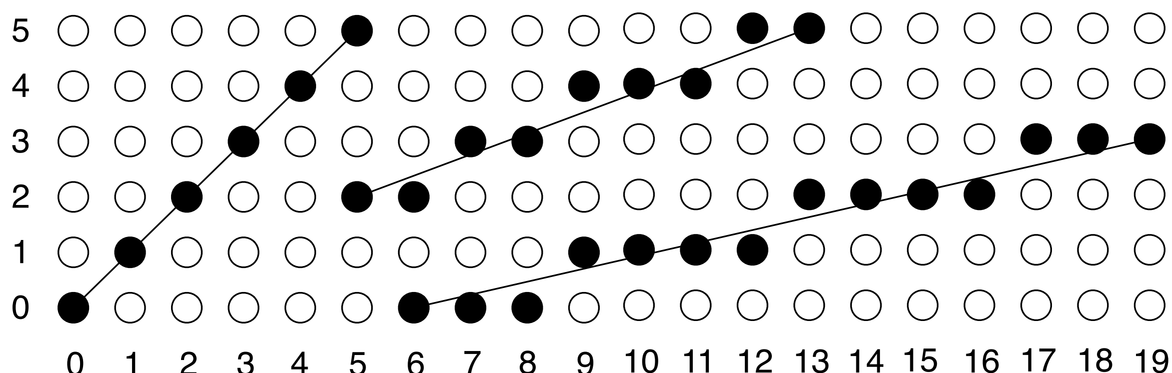
Da kan vi ikke lenger iterere over x siden vi vil tegne opp for få punkter For linja (1,1) til (2,8) vil vi eksempelvis bare tegne opp to punkt. Dette kan løses ved å teste på om dy > dx og iterere over y og regne ut x i stedet. Gode beskrivelser og hederlige forsøk på implementsjon honoreres likt.

3) Hvilket problem kan du få hvis `(x2-x1) == 0` ?

Divisjon med 0. Hvis vi håndterere deloppgave 2 riktig kan vi redusere faren for dette, men vi står igjen med problemet hvis input er to like koordinater. Bruk av unntakshåndtering vurderes ikke som noen god løsning, men det bør løses med en if-test og kode for å skrive ut en rett linje.

4) Er det andre ting som implementasjonen bør ta hånd om?

Sjekke at koordinatene er "sortert": at x1 < x2, hvis ikke blir inkrementeringen (++x) feil. Løsningen er å swappe koordinatene.

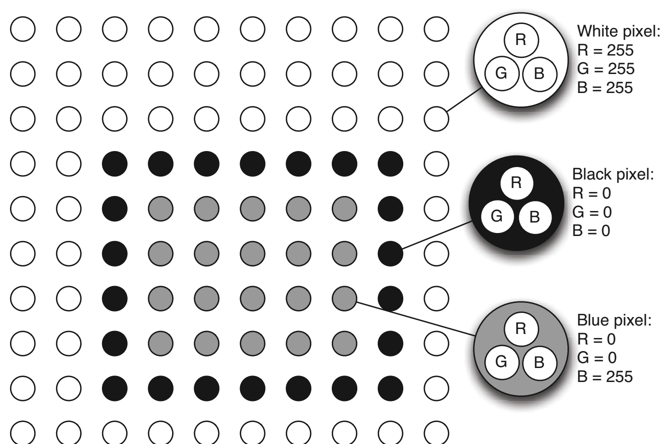


Oppgave 3: Minnehåndtering (25%)

I denne oppgaven skal du implementere en klasse **`Image`** som brukes for å representere fargebilder internt i et program. Et bilde har en gitt bredde (**`w`**) og høyde (**`h`**) og består av **`w * h`** bildepunkter. Hvert bildepunkt kan adresseres med **`(x, y)`**-koordinater. Fargen til et bildepunkt angis med tre enkeltverdier; en verdi for hver av primærfargene rød, grønn og blå (**RGB**). Verdien for hver av primærfargene angis med et heltall mellom 0-255. Slike bildepunkter i digitale bilder kalles for **pixler**.

En foreløpig deklarasjon av klassen **`Image`** er gitt under. Illustrasjonen viser eksempel på et 10x10 bilde og hvordan det logisk er bygd opp. Siden vi er interessert i effektiv minnebruk, rask skriving til fil etc. skal vi bruke en "lavnivå" datastruktur hvor vi allokerer et sammenhengende minneområde av typen **`unsigned char`** (som ett enkelt dynamisk allokert array). Du får med andre ord ikke bruke en sammensatt datatype for lagring av de enkelte bildepunkt.

```
class Image{
private:
   int width, height;
   unsigned char *pixels;
public:
   Image();
   Image(int w, int h);
   int getWidth();
   int getHeight();
   void resize(int w, int h);
   void save(const char *name);
   ~Image();
};
```



a) **`pixels`** skal peke til det dynamisk allokerte minnet vi bruker.
Hvorfor bruker vi datatypen **`unsigned char`** og hva betyr **`*`** i denne sammenhengen?
Vi skal lagre verdier mellom 0-255. Da er det mest plassbesparende å bruke char siden denne bruker 1 byte. Ja, char kan brukes også for tall. unsigned betyr at vi bare bruker positive verdier: 0-255 (uten unsigned brukes -128 til +127). * betyr av vi deklarerer en pekervariabel. Mange slet med å skjønne hva denne oppgaven egentlig ba om siden de assosiserte char* med c-streng.

b) For hver pixel (bildepunkt) trenger vi separate verdier for rød, grønn og blå. Gitt et bilde med bredde **`w`** og høyde **`h`**, hvor stort minne trenger du (antall unsigned char)?
Vi trenger w * h * 3 fordi vi skal lagre 3 char-verdier for hvert punkt. Her ble det gitt poeng også for "forståelige feiltolkninger". De som svarte w * h * 9 trodde f.eks. at du måtte lagre {'2', '5', '5'} oa.

c) Implementer klassens to konstruktører og destruktøren.

Det viktigste her er at konstruktørene og destruktoren er fornuftig i forhold til hva du svarte i 2b, at du allokerer minne i andre konstruktør og at du frigjør det samme minnet i destruktøren.

```
Image::Image(): pixels(NULL), width(0), height(0){
}

Image::Image(int w, int h): pixels(NULL), width(w), height(h){
    pixels = new unsigned char[3 * width * height];
}

Image::~Image(){
    delete [] pixels;
}
```

d) Implementer følgende get- og set-funksjoner for å sette/lese fargeverdiene til en pixel. For å forenkle litt bruker vi en struct-type `Colour` som parameter og returtype, men merk at datatypen for det dynamisk allokerte minnet er unsigned char.

```
struct Colour{ unsigned char red, green, blue;};
void setPixel(int x, int y, const Colour &c);
Colour getPixel(int x, int y);
```

Tips: For å regne om fra todimensjonal syntaks basert på (x,y) til endimensjonal indeks kan du bruke formelen ***indeks = w * y + x*** *Merk at denne må tilpasses oppgaven.*

```
//Her var det selvsagt medlemsfunksjoner for Image vi mente...
//Må regne ut riktig posisjon som rød, grønn og blå skal lagres i
//Oppgaven evalueres i forhold til hva du har svart i de foregående
//deloppgavene og det viktigste er at du skjønner at du må regne ut
//indeksen og sette flere verdier
void Image::setPixel(int x, int y, const Colour &c){
    pixels[3 * (width * y + x)] = c.red; // eller + 0
    pixels[3 * (width * y + x) + 1] = c.green;
    pixels[3 * (width * y + x) + 2] = c.blue;
}
Colour Image::getPixel(int x, int y){
    Colour temp;
    temp.red = pixels[3 * (width * y + x)];
    temp.green = pixels[3 * (width * y + x) + 1];
    temp.blue = pixels[3 * (width * y + x) + 2];
    return temp;
}
```

e) Hva betyr parameterdeklarasjonen `const Colour &c` i set-funksjonen beskrevet i deloppgave over og hvorfor bruker vi denne deklarasjonen?

Vi sender over en referanse til en Colour-variabel i stedet for at funksjonen mottar en kopi. Siden vi ikke skal endre argumentvariabelen inne i funksjonen setter vi den til const.

f) Implementer medlemsfunksjonen `void resize(int w, int h)`. Denne brukes for å endre størrelse på bildet. Hvis bredde og/eller høyde økes skal det legges til nye punkter uten at eksisterende bilde blir endret (forvrengt). Hvis høyde og eller bredde reduseres skal effekten være at du "klipper kantene" av eksisterende bilde.

```
//Denne oppgaven krever at du allokerer minne til en ny tab ell, kopierer
//over og frigjør gammel tabell. Også viktig at du finner minste høyde
//og bredde og kopierer riktig.
```

```
void Image::resize(int w, int h){
    if (height == h && width == w){
        return; //Vi trenger ikke resize
    }
    unsigned char *temp = new unsigned char[3 * w * h];
    int minheight = height < h ? height : h; //finner minste hoyde
    int minwidth = width < w ? width : w;  //finner minste bredde
    for (int i = 0; i < minheight; i++){
        for (int j = 0; j < minwidth; j++){
            temp[3 * (w * i + j)] = pixels[3 * (width * i + j)];
            temp[3 * (w * i + j) + 1] = pixels[3 * (width * i + j) + 1];
            temp[3 * (w * i + j) + 2] = pixels[3 * (width * i + j) + 2];
        }
    }
    width = w; //setter ny verdi for width medlemsvariabelen
    height = h; /setter ny verdi for height medlemsvariabelen
    delete [] pixels; //frigjør gammel tabell
    pixels = temp;  //setter pixels til å peke til ny tabell
}
```

Oppgave 4: Klasser, objekter og et litt større prosjekt (35%)

I denne oppgaven skal du jobbe med en klasse **Drawing** for tegninger med forskjellige typer figurer. Vi lar **Drawing**-klassen arve fra **Image** (oppgave 3) og bruker også **Colour** (fra 3d).

Denne oppgavesiden er en introduksjon til oppgaven, deloppgavene står på neste side!

Merk at det ikke er nødvendig å ha løst noen av deloppgavene i 3) for å gjøre denne oppgaven. Det er bare nødvendig å forstå spesifikasjonen av **Image** og hvordan medlems-funksjonene skal/ kan brukes.

Koden i den følgende forklaringen er bare ment som et utgangspunkt (et første utkast) for oppgavene du skal løse, og du kan legge til klasser og endre på de som er deklarert.

```
class Drawing: public Image{
private:
  Colour bg;
  void update();
public:
  Drawing(int width, int height, const Colour &bg);
  void setBackground(const Colour &c);
  void addLine(int x1, int y1, int x2, int y2, const Colour &c);
  void addRectangle(int x, int y, int width, int height, const Colour &c);
  ~Drawing();
};
```

Vi ønsker å kunne legge til linjer og rektangler til et **Drawing**-objekt med forskjellige medlems-funksjoner (**addLine**, **addRectangle**). Disse funksjonene tar argumenter som spesifiserer hvordan objektet som skal tegnes. For linjer må vi ha startpunkt (x1,y1), endepunkt (x2,y2) og farge. For rektangler må vi ha et hjørnekoordinat (x,y), bredde, høyde og farge.

Drawing-klassen skal være i stand til å huske hvilke figurer du har lagt til tegningen. Når du bruker add-funksjonene skal det instansieres objekter som skal "administreres" av Drawing-klassen. I første utkast til kode har vi laget følgende klasser for linje og rektangel:

```
class Line{
public:
  Line(int x1, int y1, int x2, int y2, const Colour &c);
  void paint(Image *img);
};
class Rectangle{
public:
  Rectangle(int x, int y, int width, int height, const Colour &c);
  void paint(Image *img)
};
```

Drawing arver fra **Image** og har dermed det som trengs for å lage et "punkt-grafikk" bilde. Linjer og rektangler som legges til skal tegnes opp slik at du f.eks. kan vise bildet på skjerm eller lagre til ei bildefil. Siden linje og rektangel må tegnes opp med forskjellig algoritme, har disse klassene hver sin **paint**-funksjon med **Image*** som parameter. Objektene skal med andre ord kunne tegne "seg selv" på et Image-objekt.

Figurene skal tegnes opp etter hvert som add-funksjonene kalles. Hvis det gjøres andre endringer må hele bildet tegnes opp på nytt, eksempelvis hvis du skifter bakgrunn. Vi kunne også hatt funksjoner som endret tegningen på andre måter f.eks. en funksjon for å fjerne spesifikke linjer og rektangler fra bildet.

a) Implementer konstruktøren for `Drawing`-klassen.

```
Drawing::Drawing(int width, int height,
                 const Colour &bg): Image(width, height){
     setBackground(bg);
};
//bruk av supertypens konstruktør er viktig i denne oppgaven
//vi kan bruke setBackground selv om vi ikke har impl. den ennå.
//men det er også rimelig greit å bare sette bg f.eks. i initialiserings
//lista med bg(bg)
```

b) `Drawing`-objektene må kunne iterere over alle linjer og rektangler som er lagt til og be disse om å tegne seg selv. Vis hvordan du kan bruke teknikken med arv, virtuelle funksjoner og polymorfi for å realisere dette. Vis også hvilken medlemsvariabel du vil deklarere i `Drawing`-klassen for å håndtere linjer og rektangler.

```
Se koden for Figure og Rectangle på slutten av LF.

1) Vi kan bruke arv og innfører derfor supertypen Figure og
gjør Rectangle og Line til subtyper av denne. (Ikke så vedlig ulikt øving
8, men der ble den kalt Shape).

2) Funksjonen paint() settes som pure virtual i Figur-klassen. Vi kan
også legge farge i supertypen, (ikke nødvendig siden det er litt disku-
tabelt om det er en god ide siden)

3) Som medlemsvariabel i Drawing kan vi bruke vector<Figure*> figures.
Må bruke pekertypen for å oppnå polymorfi.Noen har brukt to vectorer, men
da kan du droppe pekere. Ulempen med en slik løsning er at den er tungvint
å utvide med nye figurtyper, og du klarer ikke å holde styr på rekkefølgen
av linjer og rektangler (dvs hva som skal ligge utenpå en annen figur).
Svært mange trodde de skulle arve fra Drawing.
```

c) Implementer `addRectangle`-funksjonen i `Drawing`-klassen (se klassedeklarasjonen for parameterliste og returtype).

```
void Drawing::addRectangle(int x, int y, int width,
                           int height, const Colour &c){
  Figure *f = new Rectangle(x, y, width, height, c);
  f->paint(this); //vi tegner opp etter hvert som de legges til
  figures.push_back(f);
}

//Løsningen din vurderes i forhold til hva du har foreslått i 4b.
//Det viktigste her er at du tar vare på objektene.
```

d) Implementer `void paint(Image *img)` for `Rectangle`-klassen. Den skal sørge for at rektangelet blir tegnet opp i `img`-objektet. Hele flaten til rektangelet skal settes til fargen som rektangelet har; dvs. du skal sette fargen til alle bildepunkter som rektangelet dekker.

```
void Rectangle::paint(Image *img){
    for (int i = x; i < x + width; i++){
        for(int j = y; j < y + height; j++){
            img->setPixel(i, j, colour);
        }
    }
}
```

e) Medlemsfunksjonen `void update()` i `Drawing`-klassen skal brukes til å oppdatere hele bildet når dette trengs, mens `void setBackground(const Colour &c)` skal brukes til å endre bakgrunnsfarge. Implementer begge funksjonene og legg vekt på "samspillet" mellom dem: dvs. at den ene bruker den andre på en fornuftig måte.

```
void Drawing::update(){
    //fargelegger bakgrunnen først
    for (int i = 0; i < getWidth(); i++){
        for (int j = 0; j < getHeight(); j++){
            setPixel(i, j, bg);
        }
    }
    //tegner opp figurene
    for (int i = 0; i < figures.size(); i++){
        figures[i]->paint(this);
    }
}

void Drawing::setBackground(const Colour &c){
    bg = c;
    update();
}

//Merk at vi kun setter background-variabelen her
//Vi overlater til update å "fargelegge" bakgrunnen
//Da unngår vi surr i oppdateringslogikken
//Ikke nødvendig med helt perfekt løsning her så lenge du har skjønt at
//den ene funksjonen kan bruke den andre
```

f) Hva skjer hvis en figur er helt eller delvis utenfor bildeflaten? Vis og forklar hvor i koden og hvordan du kan teste på slike situasjoner. Bruk unntaksmekanismen for å si fra om slike situasjoner (bruk en eksisterende unntakstype eller lag en egen type).

```
class outside_of_imagearea_exception{};

bool Drawing::isWithin(int x1, int y1, int x2, int y2){
     //test på om noe er innenfor bilde kan baseres på to koordinater
     //som angir "bounding box" for figuren
     //kan være start og slutt på ei linje eller
```

```
    //nedre venstre og øvre høyre hjørne for et rektangel
    return      (x1 > 0 && x1 < getWidth())
            && (y1 > 0 && y1 < getHeight())
            && (x2 > 0 && x2 < getWidth())
            && (y2 > 0 && y2 < getHeight());
}
void Drawing::addRectangle(int x, int y, int width,
                           int height, const Colour &c){
    if (isWithin(x, y, x + width, y + height)){
        Figure *f = new Rectangle(x, y, width, height, c);
        f->paint(this);
        figures.push_back(f);
    }else{
        throw outside_of_imagearea_exception();
    }
}
```

g) I **`Image`**-klassen er det en **`void resize()`** funksjon som brukes for å endre størrelsen på bildet. Er det behov for å redefinere denne i **`Drawing`**-klassen og hvordan vil du i såfall implementere denne (og forklar hvorfor/hvorfor ikke)?

```
//Ja, denne må vi redefinere hvis vi ønsker at figurene automatisk skal
//oppdateres etter en resize. Vi kan jo bare kalle supertypens resize
//først og oppdatere bildet etterpå

void Drawing::resize(int width, int height){
    Image::resize(width, height);
    setBackground(bg);
    update();
}
```

PS! I denne løsningen hopper vi glatt over problemet at figurene kanskje ikke får plass i bildet etterpå. Dette kunne vi håndtert ved å teste på om en figur får plass før den blir tegnet opp, og evt. la være å tegne opp de som ikke får plass, eller "klippe" dem så de passer. Det er en litt dårlig design å kaste unntak for slikt siden det er noe vi bør ha løst på en "normal" måte - som ikke avbryter opptegningen av figurene. Har du tatt for deg unntakshånderingsproblematikk i stedet, er det greit svart likevel.

Noen baserte sitt svar på problemer knyttet til at figurene havnet utenfor, andre så bare behovet for å sette bakgrunnen på nytt. Så lenge du skjønner at du har behov for en redefinert resize-funksjon og gir et godt argument for dette har du egentlig svart på spørsmålet.

h) Gitt den implementasjonen du har laget, trenger du å implementere en destruktør? Hvis ikke må du forklare hvorfor. Implementer destruktøren hvis du trenger en.
Ja, vi trenger en destruktor fordi vi har instansisert dynamisk allokerte Figur-objekter. Selv om vektoren slettes automatisk vil ikke objektene vi har allokert slettes - det er bare pekervariablene inne i vektoren som vil slettes automatisk. Bruker du ikke dynamisk allokerte variabler trenger du ikke en destructor.

```
Drawing::~Drawing(){
    std::vector<Figure*>::iterator it;
    for (it = figures.begin(); it != figures.end(); ++it){
        delete *it; //iteratoren dereferert gir oss en peker
        *it = NULL; //egentlig ikke nødvendig
    }
}
```

i) Vi har latt **Drawing** arve fra **Image** i denne oppgaven. Hvilket annet alternativ har vi for å la **Drawing**-klassen bruke **Image**? Diskuter kort fordeler og ulemper ved begge løsninger.

Vi kunne hatt en medlemsvariabel av typen Image* i stedet. Vi bruker jo Image kun gjennom medlemsfunksjoner så det er fullt mulig. Fordelene med medlemsvariabel er at vi kunne erstattet objektet vi tegner opp på med spesialiserte subtyper. Det er også kanskje mer naturlig å bruke Image* som medlemsvariabel siden disse klassene egentlig er temmelig forskjellig. Fordelen med arv er at det gir en enkel design og at vi kan bruke supertypens funksjoner i subtypeobjekter. PS! Dette er forøvrig et eksempel hvor privat arv hadde vært en mulig løsning: class Drawing: private Image.

Mange har svart at vi kan gjøre den til friend, men det er dårlig svart med mindre du også poengterer at den skal være medlemsvariabel. Friendship alene oppretter ikke noe medlem, og i praksis klarer vi oss godt uten direkte tilgang til arrayet i Image.

j) For å forenkle håndteringen av fargene i Drawing-klassen skal du bruke et “fargekart” som leses fra fil. Du finner et eksempel på ei slik fil i **appendiks 1**. I fila er hver farge beskrevet på en separat linje med et navn først, etterfulgt av verdiene for henholdsvis rød, grønn og blå. Det er brukt tabulator (whitespace) mellom verdiene.

1) Deklarer en egnet medlemsvariabel for fargekartet.

```
map<string, Colour> colourmap;
```

2) Implementer en medlemsfunksjon i Drawing-klassen for å lese inn fra fil: **void loadColourMap(const char* filename)**.

```
void Drawing::loadColourMap(const char* s){
    std::ifstream infile(s);
    if (!infile){
        std::cerr << "error opening file: " << s << std::endl;
        return; //vi avslutter funksjonen, men ikke programmet
    }
    std::string name;
    unsigned int red = 0, green = 0, blue = 0;
    while(infile >> name >> red >> green >> blue){
        Colour c = { (unsigned char) red,
                     (unsigned char) green,
                     (unsigned char) blue};
        colourmap[name] = c;
    }
    infile.close();
}
```

Dette var den mest kompakte implementasjonen vi klarte å lage
Her regner vi med at dere gjør det litt mer omstendelig.....

Vi benytter oss av "kjede" med >> for å lese 4 verdier for hver iterasjon. Må gå omveien om en int for å lese tallene (og ikke bare en bokstav som er tilfellet hvis du bruker infile >> char. Vi bruker også casting og initialisering av en Colour-variabel med initialiseringsliste.

3) Implementer en overlagret versjon av en av funksjonene i **Drawing**-klassen (fritt valg) som bruker **string** (for fargenavn) i parameterlista i stedet for typen **Colour**.

```
void Drawing::addRectangle(int x, int y, int width, int height,
                           const std::string &name){

     //Helt OK å svare:
     addRectangle(x, y, width, height, colourmap[name]);

     //bedre løsning som sjekker om nøkkel finnes først.
     //men denne er litt mer enn forventet.
    std::map<std::string, Colour>::iterator it;
    if ((it = colourmap.find(name)) == colourmap.end()){
        cout << name << " is not a defined colour!" << endl;
    }else{
        addRectangle(x, y, width, height, it->second);
    }
}
```

Figure og Rectangle:

```
class Figure{
protected:
    Colour colour;
public:
    Figure(const Colour &c);
    virtual void paint(Image *img) = 0;
    virtual ~Figure(){};
};


class Line: public Figure{
private:
    int x1, y1, x2, y2;
public:
    Line(int x1, int y1, int x2, int y2, const Colour &c);
    virtual void paint(Image *img);
};

class Rectangle: public Figure{
private:
    int x, y, width, height;
public:
    Rectangle(int x, int y, int width, int height, const Colour &c);
    virtual void paint(Image *img);
};

Figure::Figure(const Colour &c): colour(c){}

Line::Line(int x1, int y1, int x2, int y2, const Colour &c):
x1(x1), y1(y1), x2(x2), Figure(c){}

void Line::paint(Image *img){
    //denne kan du implementere selv med logikken fra 2d:-)
}

Rectangle::Rectangle(int x, int y, int width, int height, const Colour
&c):
    x(x), y(y), width(width), height(height), Figure(c){}

void Rectangle::paint(Image *img){
    for (int i = x; i < x + width; i++){
        for(int j = y; j < y + height; j++){
            img->setPixel(i, j, colour);
        }
    }
}
```

Appendiks 1

Eksempel på innhold i ei colormap fil (oppgave 4 j):

```
aliceblue       240  248  255
antiquewhite    250  235  215
aqua            0    255  255
aquamarine      127  255  212
azure           240  255  255
beige           245  245  220
bisque          255  228  196
black           0    0    0
blanchedalmond  255  235  205
blue            0    0    255
cornsilk        255  248  220
crimson         220  20   60
cyan            0    255  255
darkblue        0    0    139
darkcyan        0    139  139
darkslategray   47   79   79
deepskyblue     0    191  255
dimgray         105  105  105
white           255  255  255
red             255  0    0
lime            0    255  0
green           0    128  0
```