Uppgift

Se separat fil “Exempel Dugga” för själva uppgiften.

Övning inför duggan

Duggan över du till kontinuerligt genom att lösa laborationerna. Det är viktigt ni under arbetet med laborationerna turas om att vara den som kommer fram till vilken kod som ska skrivas och att vara den som sitter i bakgrunden och kontrollerar och fungerar som bollplank.

Det är inte tänkt du ska behöva plugga inför duggan i någon stor omfattning. Dock behöver du **minst** avsätta två timmar till att på egen hand, **utan hjälpmedel**, lösa en duggauppgift (se exempeldugga). Under det arbetet ska du skriva ned allt du fastnar på. Sedan tar du reda på bästa lösningen på de problemen du råkade ut för och tar med som hjälpmedel (se exempeldugga). Det är även läge att prova alternativa lösningar och avsiktliga fel. Vad händer? Vad säger kompilatorn? Vad säger valgrind? Prova utelämna nyckelord, prova spara olika typer i vektorn. Då lär du dig mer och får lättare lösa dessa problem om de uppstår på duggan.

Introduktion till arv och polymorfi

Här bygger vi steg för steg upp en lösning på exempelduggan som en introduktion till arv och polymorfi. Dessa ämnen är centrala i Kalkylatorlabben, för att klara Duggan och för att bidra med kod till Projektet.

Vi börjar med att *noga* läsa uppgiften och fundera igenom vilka klasser som kan behövas. Detta kallas objektorienterad analys (OOA). Ett sätt att hitta kandidater är att leta efter “saker” (substantiv) i uppgifts-beskrivningen. Här är det ganska tydligt att vi ska göra rektanglar och cirklar i programmet. Hur kan vi representera respektive form som en klass? Även här ger uppgifts-texten en hel del idéer. Låt oss börja med cirkeln.

Klassen Circle

Från uppgiften får vi att cirkeln *har* en position, en radie och en utritningssymbol. Detta vet vi sedan Klockslags-labben hur vi beskriver i en klass som datamedlemmar. Ur uppgiften framgår att vi ska beräkna avstånd som inte nödvändigtvis är heltal (kvadratroten) så vi bör välja flyttal som datatyp. Datatyp för utritningssymbolen är mer oklar, men studeras den givna koden i `draw.cc` framgår att den sparas i en teckenvariabel. Vi kan nu teckna ned cirkelns datamedlemmar i konkret kod:

```
class Circle
{
  private:
    float x_pos;
    float y_pos;
    float radius;
    char symbol;
};
```

Nästa steg är att fundera på vilka medlemsfunktioner som behövs. Även detta är ett steg i OOA. En konstruktor kommer behövas eftersom vi har privata datamedlemmar. Det ska även gå att fråga cirkeln vilken utritningssymbol den har. I övrigt ska det går fråga om cirkeln täcker en viss koordinat. Vi får:

```
class Circle
{
```

```
  public:
    Circle( /* vad behövs för att skapa en cirkel ? */ );

    bool covers(float x, float y) const;
    char get_symbol() const;
};
```

Vilka parametrar behöver konstruktorn ta emot? Dels vet vi att varje datamedlem behöver ett värde, och dels vet vi att det givna huvudprogrammet ska fungera. I huvudprogrammet ser vi att en cirkel skapas som `Circle{10.0,60.0, 'x', 5.0}`. Det är rimligt att anta att talparet $(10, 60)$ som anges först är mittpunktens (x, y), tecknet `'x'` är utritningssymbolen, och sistaargumentet (5.0) är radien. Det ger oss:

```
class Circle
{
  public:
    Circle(float x, float y, char s, float r);

    bool covers(float x, float y) const;
    char get_symbol() const;

  private:
    float x_pos;
    float y_pos;
    float radius;
    char symbol;
};
```

Nu har vi en fullständig deklaration av cirkelklassen vilket betyder att vi av bara farten gjort en objektorienterad design (OOD) av klassen. OOA identifierar klassens övergripande ansvar, uppbyggnad och ev samarbetande klasser/typer i pseudokod eller diagram. OOD tar OOA vidare till konkreta deklarationer som går att implementera. Som sista steget implementerar vi allt i OOD till ett färdigt program. Nu är alltså läge att göra klart huvudprogrammet för att se till att det fungerar med bara cirklar.

Huvudprogram med bara Cirklar

Vi konkretiserar den givna vektorn genom att låta den lagra cirkelobjekt (precis som vi hade en vektor med löpare i första laborationen):

```
int main()
{
  vector< Circle > train {
    Circle{10.0,60.0, 'x', 5.0},
    Circle{20.0,60.0, 'x', 5.0},
    Circle{30.0,60.0, 'x', 5.0},
    Circle{12.0,38.0, '%', 3.0},
    Circle{16.0,26.0, ';', 6.0},
    Circle{28.0,10.0, '.', 9.0},
  };

  draw( train );

  return 0;
}
```

När vi kompilerar märker vi att vi även behöver fixa till den övriga givna koden:

Övning: Fyll i luckorna i den givna koden!

Försök färdigställa `get_symbol_for` och parameterlistan till `draw` på egen hand innan du bläddrar till lösningen på nästa sida!

Nedan redovisas *en* lösning med en *range-based* for-loop (det går bra att låta loopen använda index eller iteratorer också):

```
char get_symbol_for(float x, float y, vector< Circle > const& shape_list)
{
  char to_print {' '};

  for ( Circle const& c : shape_list )
  {
    if ( c.covers(x, y) )
    {
      to_print = c.get_symbol();
    }
  }
  return to_print;
}

// utritning av alla former i en vektor
void draw(vector< Circle > const& shapes)
```

När vi kompilerar koden igen får vi från länkaren felmeddelande *"undefined reference to"* för `Circle::Circle`, `Circle::covers` och `Circle::get_symbol`. Vi har inte implementerat dem ännu!

Övning: Implementera cirkelklassens medlemsfunktioner!

Försök färdigställa konstruktor, `covers` och `get_symbol` på egen hand innan du bläddrar till lösningen på nästa sida! Du kan behöva kontrollera `pow` och eventuellt `sqrt` på `cppreference.com`.

Bra jobbat! Här redovisas en rimlig lösning:

```
Circle::Circle(float x, float y, char s, float r)
  : x_pos{x}, y_pos{}, symbol{s}, radius{r}
{
}

bool Circle::covers(float x, float y) const
{
  double dist {pow(x - x_pos, 2) + pow(y - y_pos, 2)};
  return dist < pow(radius, 2);
}

char Circle::get_symbol() const
{
  return symbol;
}
```

Nu ska koden gå att både kompilera och köra!

Blir cirklarna inte som i körexemplet? Då behöver du felsöka. Har du missat eller feltolkat något i uppgiften? Har du feltolkat given kod, t.ex. argumenten till cirkelkonstruktorn? Har du råkar skriva fel någonstans? Gå inte vidare förrän cirklarna fungerar!

Klassen Rectangle

Gör nu samma övning för rektangeln som vi gjorde för cirkeln.

Övning: Skriv rektangelklassen!

Försök färdigställa klassen själv innan du bläddrar till lösningen på nästa sida!

Du bör komma fram till något väldigt likt följande:

```
class Rectangle
{
  public:
    Rectangle(float x, float y, char s, float w, float h)

    bool covers(float x, float y);
    char get_symbol();

  private:
    float x_pos;
    float y_pos;
    float width;
    float height;
    char symbol;
};

Rectangle::Rectangle(float x, float y, char s, float w, float h)
  : x_pos{x}, y_pos{y}, symbol{s}, width{w}, height{h}
{
}

bool Rectangle::covers(float x, float y) const
{
  return (x < x_pos + width && x >= x_pos &&
          y < y_pos + height && y >= y_pos);
}

char Rectangle::get_symbol() const
{
  return symbol;
}
```

Ta nu ett steg tillbaka och jämför klassen för cirkeln med klassen för rektangeln. Vilka likheter finns? Vilka skillnader?

Övning: Jämför cirkel- och rektangelklassens skillnader och likheter!

Fundera både konkret (deklarationer) och abstrakt (vad som konceptuellt representeras eller utförs). Skriv ned de likheter och skillnader du hittar innan du bläddrar till facit på nästa sida.

Likheter:

- Båda klasserna representerar en form.
- Båda har en position (x,y).
- Båda har en utritningssymbol med tillhörande get-funktion.
- Båda har funktionen "covers" med uppgiften "avgöra om formen täcker positionen".

Skillnader:

- Klassnamn och konstruktor.
- Ett par datamedlemmar.
- Detaljimplementation av funktionen "covers".

Alla likheter är... KODUPPREPNING! Detta måste vi naturligtvis lösa.

Nu vore det praktiskt om vi kunde skriva de delar som är lika för sig och sedan importera till cirkeln respektive rektangeln. Just detta gör vi genom att skapa en *basklass* med de delar som är lika, och sedan importera den till cirkeln respektive rektangeln. Själva importen benämns *arv* (eng. *inheritance*), och vår cirkel och rektangel blir *härledda* klasser från basklassen.

Ett arv representerar inte bara liknande medlemmar i två godtyckliga klasser, utan även att klasserna är en specialisering av basklassen, eller omvänt, basklassen är en generalisering av varje härledd klass. Härledda klasser bör alltid ha något konceptuellt eller abstrakt gemensamt: Kaniner och Katter är Djur, Plus och Minus är Operatorer, Bool och Char är Datatyper, Bil och Båt är Fordon. Bok och Film är Media.

Basklassen behöver vi alltså namnge finurligt så den beskriver något som *både* en cirkel och en rektangel **är** för att arvet ska få en tydlig innebörd. I detta fall kan vi säga "en cirkel **är** en slags form" och "en rektangel **är** en slag form". Det ger att basklassen bör beskriva en generell "form" (eng. "Shape"). Klasserna för rektangel och cirkel representerar en *specialiserad* eller *konkret* version av basklassens typ (en "cirkel" är mer konkret än bara en "form").

Ibland kan vi ha klasser som ser lika ut utan att ha en gemensam generalisering. I dessa lägen är arv olämpligt.

I andra fall kan objekten se ut att vara oförenliga, t.ex. "Bananskal" och "Voldemort", men båda kan mycket väl vara "Objekt som skadar spelaren" i ditt nya dataspel. Det är alltså i stor utsträckning upp till dig som programmerare att tydligt definiera vad som förenar olika klasser i en gemensam basklass.

Till slut kan vi ha klasser som *bara* har en gemensam generalisering men egentligen inga gemensamma datamedlemmar. Då kan arv fortfarande vara både lämpligt och användbart. I C++ är både `std::ostringstream` och `std::ofstream` härledda klasser till basklassen `std::ostream`. Det som är gemensamt i detta fall är hur de kan användas! Båda är destinationer för utmatning och används på samma sätt, även om destinationen här helt olika.

Övning: Skriv en klass enbart med cirkelns och rektangelns gemensamma delar!

Försök färdigställa klassen själv innan du bläddrar till lösningen på nästa sida!

Basklassen Shape

Du bör komma fram till ungefär följande:

```
class Shape
{
  public:
    Shape(float x, float y, char s);

    bool covers(float x, float y);
    char get_symbol();

  private:
    float x_pos;
    float y_pos;
    char symbol;
};

Shape::Shape(float x, float y, char s)
  : x_pos{x}, y_pos{y}, symbol{s}
{
}

bool Shape::covers(float x, float y) const
{
  // hur gör vi här ???
}

char Circle::get_symbol() const
{
  return symbol;
}
```

Inte så svårt? Men vi har stött på ett problem. Hur räknar vi ut om en generell "form" täcker en viss koordinat? Det går ju inte om vi inte vet vilken *konkret* form vi har med att göra!

I C++ kan vi explicit ange att det är implementationen i den härledda klassen som ska användas om den finns:

```
class Shape
{
  public:
   ...
   virtual bool covers(float x, float y);
   ...
};
```

Och vi kan explicit ange att det inte finns någon lämplig implementation av en virtual-funktion:

```
class Shape
{
  public:
   ...
  virtual bool covers(float x, float y) = 0;
   ...
};
```

Det blir nu upp till varje enskild härledd klass att fylla på med en implementation. Det innebär även att basklassen inte blir komplett, implementationen av en medlemsfunktion saknas ju. Vi kallar då basklassen en *abstrakt* och funktionen *pure virtual*. I C++ hanteras detta genom att kompilatorn förbjuder oss att faktiskt skapa objekt av sådana klasser. Detta sammanfaller oftast med att vi inte har behov att kunna skapa sådana generella objekt. I vårt ritprogram finns t.ex. ingen anledning att skapa objekt av typen `Shape`, för vilken form får vi då? Istället skapar vi cirklar eller rektanglar.

När vi har en funktion *virtual* innebär det att anrop av den sker med *dynamisk bindning*. Under programkörningen, precis innan anropet, kontrolleras helt enkelt vilken klass som *faktiskt* ligger i minnet och anrop till den klassens implementation görs. Det normala förfarandet är annars *statisk bindning* vilket innebär det är deklarationen i programkoden som avgör vilken implementation som anropas.

Härledda klasserna Circle och Rectangle

Nu återstår att faktisk ta bort koduppprepningen genom att låta Circle och Rectangle bygga ut (specialisera/ärva) basklassen:

```
class Shape
{
public:
  Shape(float x_pos, float y_pos, char symbol);

  virtual bool covers(float x, float y) const = 0;
  char get_symbol() const;

private:
  float x_pos;
  float y_pos;
  char symbol;
};

class Rectangle : public Shape
{
public:
  Rectangle(float x_pos, float y_pos, char symbol, float width, float height);

  bool covers(float x, float y) const override;

private:
  float width;
  float height;
};

class Circle : public Shape
{
public:
  Circle(float x_pos, float y_pos, char symbol, float radius);

  bool covers(float x, float y) const override;

private:
  float radius;
};
```

Notera att vi lägger till *override* på de deklarationer som ska ersätta basklassens virtual-funktion. Så klart stöter vi på några problem. Hur initieras basklassen och hur kommer vi åt privata medlemmar i basklassen när vi ska implementera `covers`?

```
  Circle::Circle(float x, float y, char s, float r)
    : radius{r} // hur initieras medlemmar ärvda från baslassen?
  {
  }

  bool Circle::covers(float x, float y) const
  {
    double dist {pow(x - x_pos, 2) + pow(y - y_pos, 2)}; // x_pos och y_pos är privata i basklassen
    return dist < pow(radius, 2);
  }
```

Lösningen är att alltid anropa basklass-konstruktorn och byta “private” till “protected” i basklassen:

```
  Circle::Circle(float x, float y, char s, float r)
    // Anrop av basklass-konstruktorn sker först:
    : Shape{x, y, s}, radius{r}
  {
  }

  // Datamedlemmar som ska användas i härledd klass placeras
  // under "protected" istället för "private" i basklassen
```

Övning: Skriv klart klasserna med användning av arv!

Till detta finns ingen lösning på nästa sida! Med ledning av ovan kan du pussla ihop cirkelklassen och skriva rektangelklassen enligt samma principer på egen hand.

Huvudprogram med både Cirklar och Rektanglar

Nu är det dags att lägga till rektanglar i huvudprogrammet. Hur gör vi det?

En frestande men mycket ***problematisk*** lösning:

```
int main()
{
  vector< Circle > c {
    Circle{10.0,60.0, 'x', 5.0},
    ...
  };

  vector< Rectangle > r {
    Rectangle{5,49, '#', 31,6},
    ...
  };

  draw( c );
  draw( r );

  return 0;
}
```

Varför är detta problematiskt? Det kommer att bli mer och mer programkod för varje ny form som läggs till. Tänk dig att du ska lägga till romber, trianglar, kvadrater, pentagram, ovaler, månskäror och hexagoner. Det blir lika många vektorer, och varje vektor måste ha en egen "draw". Redan här är vi uppe i 9 `draw`-funktioner att underhålla för varje enskild ändring.[1]

Behöver vi senare dessutom undersöka om former överlappar för att kunna markera överlapp i det som ritas ut kommer det snabbt bli en exponentiell ökning av mängden kod. För att hitta överlapp måste varje form, dvs allt i varje vector, jämföras mot alla andra lagrade former, dvs jämföras med allt i varje annan vector. För varje unikt par av vektortyper blir det alltså en loop. Varje sådan loop skulle behöva sin egen kod. Med 9 olika former blir det 9 vektorer, första vektorn behöver jämföras med de 8 andra, andra vektorn behöver jämföras med de 7 återstående, osv. Det blir totalt 36 olika loopar för överlapp-kontroll![2]

Kanske vill vi även sortera våra former i storleksordning så vi kan rita de största först. Fundera själv på hur alla former i 9 olika vektorer kan sorteras i storleksordning. Om du känner att det blir riktigt besvärligt är du på rätt spår. Att använda flera vektorer är helt enkelt en återvändsgränd.

För att lösa dessa problem behöver vi ha ***en*** vector med ***alla*** objekt. Vi kan utnyttja att både cirklar och rektanglar och varje ny typ av form har en gemensam basklass. Alla konkreta former **är** ju både sig själva och sin basklass. Vi fyller vektorn med basklass-objekt:

```
int main()
{
  vector< Shape > shapes {
    Circle{10.0,60.0, 'x', 5.0},
    ...
    Rectangle{5,49, '#', 31,6},
    ...
  };

  draw( shapes );

  return 0;
}
```

[1] Just detta kan avancerad C++ rädda oss från relativt enkelt. Läs TDDD38.

[2] Normalt använder vi Ordo-begreppet för att resonera kring hur mycket mängden operationer i en beräkning ökar i förhållande till mängden indata. Detta ger oss en uppfattning om hur mycket beräkningens körtid kommer att öka när indatamängden ökar. Även om det är helt atypiskt kan vi även använda begreppet för att undersöka hur snabbt mängden programkod att skriva växer i förhållande till antalet klasser N. I detta fall får vi vid kontroll av överlapp fler och fler loopar för varje klass som läggs till. Antalet loopar blir summan av alla tal $(N-1)$ ned till 1, dvs $N*(N-1)/2$ vilket ger $Ordo(N^2)$. Vi skulle vilja ha en lösning där vi kan lägga till en klass utan att lägga till ytterligare kod, dvs $Ordo(1)$. Eftersom det till slut är *arbetstid att skriva koden* detta atypiska komplexitets-resonemang mäter är det extremt viktigt att hålla ned komplexiteten.

Det ser mycket bättre ut! Bara en vektor att hantera och bara en `draw`-funktion att skriva. Det blir "bara" en dubbelloop för att jämföra alla mot alla eller för att sortera. Denna lösning är mycket mer hanterbar och skalbar. Vi har dock ett problem. Kompilatorn säger att `Shape` måste ha en default-konstruktor för att få finnas i en vektor, och dessutom kan vi ju överhuvudtaget inte skapa objekt av typen `Shape` eftersom klassen inte är klar. Kom ihåg att vi gjorde `covers` till en *pure-virtual*-funktion utan implementation i basklassen.

Nu är det frestande att lägga till en default-konstruktor i `Shape` och göra en tom implementation av `covers`. Det kommer dock bara ta oss till ytterligare en återvändsgränd. Om vektorn på varje plats lagrar en `Shape` så kommer det inte att få plats en cirkel eller rektangel på den platsen, de härledda klasserna har ju fler datamedlemmar än basklassen. Om vi försöker stoppa in en `Circle` i vektorn kommer allt som finns i `Circle` men som inte finns i `Shape` att skalas bort. Detta kallas *slicing* och är *aldrig* önskat även om kompilatorn inte säger något om det.

Det vi istället måste göra är att låta vektorn *hänvisa* till former som skapats på en annan plats i minnet. Det kan vi göra med pekare. Då måste vi också allokera minne till våra objekt med *new* och av-allokera det med *delete* när vi är klara med vektorn. Vår slutliga `main`:

```
int main()
{
  vector< Shape* > train {
    new Circle{10.0,60.0, 'x', 5.0},
    ...
    new Rectangle{5,49, '#', 31,6},
    ...
  };

  draw( shapes );

  for ( Shape* shape : train )
  {
    delete shape;
  }

  return 0;
}
```

Kompilerar vi nu med rätt uppsättning varningar kommer kompilatorn rekommendera en virtuell destruktor i basklassen (`virtual ~Shape()`). Kör vi programmet i valgrind kan valgrind visa på minnesläckor trots att vi gjort delete! Detta beror på att statisk bindning används till destruktor-anropet. Vi allokerar minne till cirklar och rektanglar, men vi tar bara bort minne för former. Destruktorn för `Circle` och `Rectangle` kommer aldrig anropas, trots att det är cirklar och rektanglar vi tar bort. Genom att markera `Shape`-destruktorn som *virtual* gör vi precis före anropet en kontroll av vilken klasstyp som *faktiskt* ligger i minnet som pekaren hänvisar till, och sedan anropar vi den klasstypens destruktor. Destruktorn i en härledd klass kommer automatiskt att anropa destruktorn för sin basklass så att alla destruktorer körs.

Slutligen behöver du justera funktionerna `draw` och `get_symbol_for` så även de använder pekare och så ska programmet fungera!

Övning: Lägg till en ny klass för att representera parallellogram!

Ett parallellogram är en fyrhörning där motstående sidor är parallella. Vi låter övre och undre sidan vara vågräta med ett visst avstånd emellan (höjden), medan höger och vänster sida har samma lutning och ett visst avstånd emellan (bredden). För varje steg i y-led ökar x-koordinaten i förhållande till lutningen. Vårt parallellogram täcker en koordinat (x, y) om y befinner sig mellan övre och undre sidan och x befinner sig mellan x_pos och $x\_pos + width + dx$ där $dx = (y - y\_pos) * slope$.