DD2387 Programsystemkonstruktion med C++

Projektuppgift: Äventyrsspel

30 oktober 2014

Bonuspoäng: *4p om redovisningen sker i tid*

I den här labben kommer du att lära dig att skriva ett lite större program med hjälp av arv, polymorfi och standardklasserna (STL). Ni får jobba i grupper om högst två personer. Vid redovisningen ska alla gruppmedlemmar kunna svara på frågor om alla delar av koden. Läs igenom hela lydelsen innan du börjar. För att uppgifterna ska kännas mindre lösryckta hänger de ofta ihop.

Spelet är ditt sätt att visa att du behärskar centrala delar av språket C++. Att du vet hur polymorfism och virtual fungerar, att du kan sätta restriktioner på data med public/private/friends/const, att du kan hantera minnesalokering rätt och att du kan använda STL.

Allmänna krav:

- Nyckelord som `const`, `virtual`, etc. ska användas på ett korrekt sätt.
- Din kod ska vara modulariserad i klasser och filer.
- Ditt program ska inte läcka minne, så var noga med dina konstruktorer och destruktorer.
- I denna labb ska du använda STL
- Du måste använda en makefile för att bygga dina klasser så att byggningen inte tar så lång tid. Ett exempel på en makefile hittar du på kurskatalogen.
- Uppgiften är löst hållen för att inte påverka din kreativitet negativt. Se dock till att du uppfyller kraven i 3.2. Är du osäker på omfattningen av spelet så kan du fråga kursledare eller övningsassistent.
- Extrauppgifterna har ibland förslag på använding av nya standarden C++11, det är inga krav men rekomenderas varmt. Överhuvudtaget rekomenderas att prova på C++11 specifika saker som lamdafunktioner, for (each) loopar, auto m.m.

Vid redovisning ska följande vara förberett:

- En editor där din makefile är öppnad.
- Ett kommandoskal (shell) där prompten står i labbkatalogen.
- Alla program ska vara indenterade (*M-x indent-buffer* i emacs).
- Koden ska vara färdigkompilerad.
- Ni ska veta i vilken fil varje uppgift finns.
- Ni ska ha några skisser över kodstrukturen tillgängliga.
- Läst på och vara förberedd på frågeställningarna i 3.2

- Komprimerat källkoden i en arkivfil. Ett sätt att komprimera är att använda javas jar-program t.ex. `jar cvf source.zip *.cpp *.h makefile`

Se till att labhandledaren skriver under på ditt kvittenspapper.

Lycka till!

Introduktion: Ett rollspel var från början ett brädspel där flera personer deltog. Spelet utspelade sig ofta i en fiktiv värld (*fantasy*) där spelarna mötte trollkarlar, krigare, drakar och andra varelser. Alla varelser i spelet hade egenskaper såsom styrka, uthållighet och magi. Spelet leddes av en spelledare som berättade för spelarna vem de mötte och vad som utspelade sig. Ett exempel på denna typ av brädspel är Drakar och Demoner (*Dungeons & Dragons*).

Du ska i den här labben implementera ett fullskaligt äventyrsspel. Dina klasser kommer att representera troll, drakar m.m. Varelsernas egenskaper kommer att representeras av klassernas medlemmar. Spelledarens roll kommer att skötas av en kommandotolk, dvs ett textbaserat gränssnitt där du anger vad du vill göra. Efter du gjort ditt drag kommer spelledaren låta övriga varelser göra sitt.

Om du inte gillar drakar och demoner kan du istället skriva ett valfritt äventyrsspel, t.ex. om hur du pallar äpplen av dina grannar på fredagskvällarna.

3.1 (arvsstruktur)

I denna uppgift kommer du att definiera klasshierarkier och objekt. Sist i uppgiften kommer objekten samverka med varandra.

- Börja med att välja ett lämpligt namn på den namnrymd som ska innehålla klasser och data i äventyrsspelet.

a) Gör en klasshierarki för **aktörerna** i ett äventyrsspel med attribut och metoder:

- Skapa en klasshierarki för aktörerna. Låt några av klasserna vara instansierbara och representera aktörer i spelet. *Exempel*: en trollkarl är en människa som är en aktör.
- Låt varje aktörklass ha en egenskap (gärna fler) specifik för den aktör de representerar. Några av värdena för varje aktör ska variera över spelets gång. *Exempel*: en trollkarl har ett värde för magi som minskar då han trollar. Alla varelser har kroppspoäng (liv). Genom att äta spenat kan en människa få mer kroppspoäng, dock inte över ett maximalt värde. Alla varelser har styrka.
- Låt aktörerna ha en mängd val de kan göra då det är deras tur. Låt valen bero på var de är, vem de har i närheten och hur starka deras egenskaper är. *Exempel*: ett skadat troll letar mat, ett friskt troll springer runt slumpmässigt, ett troll i närheten av en människa blir argt.

Förslag på funktioner i basklassen för aktörerna:

- `type()` – returnerar namnet på arten, t.ex. trollkarl eller drake
- `name()` – returnerar namnet på varelsen, t.ex. Merlin
- `action()` – aktörens tur att agera

- `go(direction)` – gå åt håll
- `fight(Character)` – slåss med
- `pick_up(Object)` – ta upp sak
- `drop(Object)` – släpp sak på marken
- `talk_to(Character)` – konversera med

b) Gör en klasshierarki för **miljön** med attribut och metoder:

- Skapa en klasshierarki som representerar miljön i ditt spel. Låt några av klasserna vara instansierbara och representera olika varianter på miljöer. *Exempel*: rum → inomhusmiljö → miljö, djungel → utomhustyp → miljö, strand → utomhustyp → miljö, hav → vatten → utomhustyp → miljö.
- Låt de olika typerna av miljöer ha utgångar åt olika håll. Ett miljöobjekt måste kunna svara på vilka riktningar som spelarna kan gå. Bortom varje utgång ska man komma till ytterligare ett miljöobjekt. *Exempel*: rum har bara fyra riktningar, skog har åtta. Vissa riktningar är blockerade av väggar, murar, berg och träd.
- Låt varje instans av en miljöklass ha en *beskrivande* text. Om du låter texten beskriva utgångar och föremål så blir spelet mer levande och roligare att spela.
- Olika typer av miljöer kan ha olika funktionalitet. På stranden kan man sola sig och bli mer attraktiv, i djungeln ser man bara gula ögon om natten, grottor är mörka och det regnar inte i dem, kvicksand får man inte stå still för länge på för då trillar man ner i en annan miljö.

Förslag på funktioner i basklassen för miljöerna:

- `directions()` – returnera vilka utgångar som finns
- `neighbor(direction)` – returnera granne (t.ex. referens till objekt) i given riktning
- `description()` – returnera beskrivning av vad miljön innehåller, vilka föremål man kan ta och vilka aktörer som befinner sig på platsen.
- `enter(Character)` – aktör kommer till platsen
- `leave(Character)` – aktör går från platsen
- `pick_up(Object)` – någon tar upp ett föremål som finns på platsen
- `drop(Object)` – någon lägger ner ett föremål på platsen

c) Gör en klasshierarki för **föremål** med attribut och metoder:

- Skapa en hierarki av klasser som representerar föremålen i ditt äventyrsspel. *Exempel*: börs → behållare → föremål, ring → föremål

- Låt alla föremål ha egenskaper. Alla föremål måste kunna svara på frågor om deras egenskaper, såsom dess pris, vikt, volym etc. *Exempel*: En ryggsäck har plats för ett visst antal saker/viss volym och går sönder om man lastar den för tungt, en ring kostar 500 silverpengar, ett svärd väger 2 kg.

Förslag på funktioner i basklasserna för föremålen:

- `weight()` – vikt
- `volume()` – volym
- `price()` – pris

Förslag på funktioner för behållare:

- `hold_weight()` – vikt innan behållaren går sönder
- `hold_volume()` – volym innan behållaren blir full
- `add(Object)` – lägg objekt i behållaren
- `remove(Object)` – ta bort objekt från behållaren

d) Sätt ihop en handfull miljöinstanser så att de bildar en liten spelplan. Instansiera några aktörer och lägg dem i en vektor `Vector<Character *>`. Lägg även ut instanser av föremål, varav någon behållare, i de olika miljöerna.

Iterera över vektorn och låt varje aktör utföra funktionen `action()`. Låt aktörerna plocka upp objekt, lägga ned objekt, gå genom dörrar, prata med varandra, bli arga och slåss etc. Skriv ut information om hur spelet fortlöper, aktörernas namn och typ och vad de gör.

Tips: Du kan behöva en slumpgenerator `rand()` och en slumpinitierare `srand()` (se `man rand`). Om man använder `rand()` utan `srand()` får man alltid samma slumptalsföljd, vilket kan vara användbart när man letar fel.

3.2 Du ska nu skapa ett riktigt, spelbart äventyrsspel för en spelare. Följande funktionalitet ska finnas:

- Ge en introduktion före spelet börjar där bakgrundshistorien berättas, målet står specificerat och några kommandon omnämns.
- Spelet ska ha en kommandotolk som sköter all inmatning till spelet. Speltolken tar ett kommando från tangentbordet och utför handlingen. Kommandotolken bör klara alla funktioner som aktörernas basklass specificerar. *Exempel*: `pick up sword`, `go north`, `buy shield`.

  Alternativt: visa spelplanen med teckengrafik. Låt användaren manövrera genom att hämta in ett tecken med `getch`.
- Låt eventuella strider ske i omgångar tills någon part flyr eller avlider. Striderna kan t.ex. styras med tärningar.

- Låt minst en händelse bero på yttre/tidigare omständigheter. *Exempel*: dörren öppnas bara du har rätt nyckel, lönndörren visar sig bara om du försöker gå i den riktningen, vakten släpper in dig bara om du tidigare talat med en speciell instans av en aktör, falluckan öppnar sig bara om du väger tillräckligt mycket.
- Spelet ska ha ett mål. När målet är uppfyllt ska spelet avslutas.
- Du som kan spelet (och vet var den hemliga nyckeln m.m. finns) ska kunna spela spelet från start till mål under en redovisning.
- Du ska kunna visa upp några virtuellt nedärvda metoder vars implementering väsentligen skiljer sig och du ska kunna diskutera designskillnaden mellan att deklarera metoder virtuella eller inte. Därför måste spelet ha en viss omfattning, ett alltför tunnt spel kan underkännas. Om du kan visa detta med enbart en arvstruktur med flera intressanta exempel av virtuella funktioner och dynamisk bindning så är det tillräckligt. Alla tre arvsstrukturer kan göras intressanta. En del kanske vill ha många olika sorters aktörer (Harry Potters kompisar, lärare, husdjur, fiender, porträtt, spöken m.m.) eller invecklade strider (expelliarmus) andra kanske vill ha roliga föremål och gåtor (da Vinci koden) eller spännande miljöer (vidbehovs rum, lustiga huset m.m.) Om du känner dig osäker, ta kontakt med din övningsledare.
- Du ska lösa hur olika objekt når varandra. Håller rummen reda på vad som är i dem och vet aktörerna vilket rum de är i? Vet objekten vem som håller i dem. Man kan ha endast en global vektor där allting finns och som man letar upp vad man vill ha. Eller så har man flera behållare som pekar eller refererar till samma objekt, t.ex. att både rummen och aktörerna håller reda på var aktören befinner sig. Man kan också tänka sig överföring av ägande, sjön ger bort excalibur till prins Arthur (jämför med autopointers/uniquepointers funktionalitet). I princip kan man jämföra designproblemet med hur objekt når varandra som valet mellan enkellänkad och dubbellänkad lista. Det är enklare att nå vissa saker med en dubbellänkad lista men man måste komma ihåg att uppdatera både `next` och `previous`.

Inför redovisning

Förbered några handritade skisser för att underlätta förståelsen av er design. Om du gjort UML-diagram påtala detta vid redovisning.

- Hur sker minnesallokeringen? Var görs allokering och destruktion?
- Läcker programmet minne? Kör en gång med valgrind.
- Är alla read-only metoder const-deklarerade?
- Beskriv klasshierarkin? Visa klassdiagram.
- Hur ser slingan ut som hanterar händelser. Hur hanteras händelser?

- Vad är det som håller reda på var spelaren är? Vad håller reda på alla andra objekt i spelet? Hur ser det ut i minnet, visa en minnesbild.
- Hur kopplas miljöerna ihop? Visa minnesbild.
- Hur hittar man saker/grannar? Hur sker uppslagningen?
- På vilket sätt skiljer sig karaktärer i spelet?
- Hur sker inmatning? Hur sker parsningen av det som inmatas?
- Hur fungerar action-metoden?

Betygshöjande extrauppgifter

Extrauppgift 3.1 (6p krav för betyg D) Använd pekare till medlemsfunktioner och lambdafunktioner i ditt program.

En teknik för att undvika stora if-satser är att lägga alternativen tillsammans med funktionspekare i en map. Skriv kommandotolken med hjälp av funktionspekare och pekare till medlemsfunktioner. Skapa en `std::map` och låt namnet på varje kommando och/eller kortkommando vara nyckel till en pekare till kommandot. Skapa ytterligare en `std::map` och låt namnet på en aktör vara nyckeln till det objekt aktören representeras av. När ett kommando verkar på en aktör, använd funktionspekaren genom objektet. Olika kommandon kan ge upphov till funktionspekare av olika typer. Ibland vill man ha ropa på funktioner med olika antal element. Ett simpelt sätt är att använda en map på första nyckelordet och skicka vidare resten av argumenten till en ny f-sats/map-uppslagning. I nya C++11 standarden finns det `variadic templates`, som tillhandahåller ett mer avancerat sätt att hantera problemet.

Syntaxen för en medlemspekare är krångligare (pekaren till objektet måste med, går att lösa med std::function) än en vanlig funktionspekare och därför ska den användas så att ni i framtiden inte backar för krånglig syntax.

Använd även en lamdafunktion någonstans i ditt program. Om du har en elegantare lösning på denna extrauppgift med funktorer och/eller templates kan du redovisa den elegantare lösningen. En annan lösning som ligger nära till hands är en map med omväxlande funktionspekare och lamdafunktioner men om du redovisar en alternative lösning så visa även ett exempel på en medlemsfunktionspekare (deklarering/anrop) så att handledaren övertygas om att du behärskar syntaxen.

Extrauppgift 3.2 (4p) Spelets position ska gå att spara till fil och ladda från fil. Använd strömmar (*streams*). *Exempel*: `save my_game1`, `load game7`.

Man ska kunna utvidga ditt spel så att det går att ladda ett nytt spel under spelets gång utan att det läcker minne.

Extrauppgift 3.3 (5p) Låt spelets karta och aktörer definieras av en fil. Ladda all information såsom miljöer, karta (miljöernas förhållande till varandra), aktörer och föremål från fil. Observera att det bara är objekten och deras egenskaper som ska läsas från filen medan objektens funktioner och beteenden ligger i klasserna de instansierar.

Filformatet kan t.ex. se ut som

```
MIL1:Du står i ett liten grotta. En svag belysning
  avslöjar ett stort hål i golvet.:MIL2:OBJ2:AKT3
MIL2:Du befinner dig i en trång gång mellan två
  salar. Väggarna är våta och hala.:MIL1,MIL3:OBJ1:
MIL3:Du är i ett stor sal där tre stora bord står
  dukade.:MIL2::AKT1,AKT2
AKT1:TROLL:Gruff-Gruff:kroppspoäng=17,iq=3:
  OBJ4,OBJ5
OBJ1:BEHÅLLARE:OBJ6,OBJ7:en liten ryggsäck:10kg,
  10liter,2daler
OBJ2:PENGAR:en mängd daler:0kg,0liter,10daler
OBJ3:ENHET:ett bredsvärd med förgylld egg:3kg,
  1liter,350daler
...
```

Filen ska kollas så att den innehåller konsistenta användningar av objekt: utgångar måste vara till rum som finns, föremål som används ska vara definierade etc. Hur ska man hantera händelser som beror på yttre/tidigare omständigheter?

Tips: Gör denna uppgift tillsammans med extrauppgift 3.2 så får du mycket på köpet.

Extrauppgift 3.4 (5p) krav för betyg B Inför minst tre sorters objekt som kan konstrueras (och destrueras) under spelets gång och inte enbart när spelet tar slut. Du får själv välja om det ska vara aktörer, föremål eller miljöer. Ett exempel för aktörer kan vara att det då och då vaknar upp vampyrer som börjar gå omkring i världen. Dessa vampyrer kan slås ihjäl någon helt annanstans i spelet av spelaren eller av någon annan vampyrdräpare som råkar gå förbi. Ett annat exempel för föremål kan vara ett äppelträd där spelaren eller andra busungar kan palla mogna eller omogna frukter. Äpplena kanske man kan ge vidare till någon annan, t.ex. sin söndagsskolefröken som gör äppelpaj av dem. Dynamiska miljöer som det kan skapas flera av, isflak som flyter förbi? Tjänsterum på byråkrativerket som ibland infinner sig på slumpmässigt våningsplan ...

Redovisa hur spelet håller reda på dessa objekt som skapas och dör under spelets gång. Du ska kunna argumentera för att din lösning håller för att utvidgas med fler sorter av den här typen av objekt.

I C++11 finns shared_ptr och weak_ptr som du kan experimentera med. Minst en av dina dynamiskt allokerade objekt ska dock allokeras och deallokeras med new och delete så att du kan visa dina färdigheter i att hantera dynamiskt allokerat minne. Testa minnesläckor med valgrind.

Extrauppgift 3.5 (9p) Gör ett grafiskt händelseorienterat (t.ex. klickbart) gränssnitt till spelet. `Varning` - räkna med åtskilliga arbetstimmar, det är kanske den mest dyrköpta extrapoängsuppgiften. **Koden måste vara väl strukturerad** använd t.ex. designmönstret model-view-controller. Grafikkoden ska i huvudsak vara åtskild från den övriga koden och uppdelad i klasser. Kom ihåg att labben ska imponera med dina kunskaper i C++ inte dina kunskaper i användandet av grafikrutiner. Extrauppgiften med en map med funktionspekare

kan inte göras som beskrivet på kommandotolken. Gör istället en annan map med medlemsfunktionspekare som ersätter någon annan stor if-sats i koden.

Extrauppgift 3.6 (9p) Gör ett trådat nätverksspel. Inför ett 'tick' som går ett visst antal gånger per sekund genom spelet. Låt alla aktörer som inte är spelare agera efter ett visst antal 'ticks'. Exempel:

```
A troll has arrived from the east
[tre sekunder går]
A troll says Yum, yum I will eat you
A troll hits you
...
```

Gör det möjligt att låta fler än en spelare spela spelet. `Varning` - räkna med åtskilliga arbetstimmar, det är en ganska dyr extrapoängsuppgift. **Koden måste vara väl strukturerad**. Nätverkskoden ska i huvudsak vara åtskild från den övriga koden och uppdelad i klasser. Kom ihåg att labben ska imponera med dina kunskaper i C++ inte dina kunskaper i användandet av tråd- och nätverksbibliotek. Använd gärna trådbiblioteket i nya C++-standarden.

Extrauppgift 3.7 (4p)

UML (Unified Modeling Language) kan man använda för att dokumentera ett kodprojekt. På www.uml.org finns länkar till UML-verktyg och UML-tutorials. I den här extrauppgiften ska du dels visa upp korrekta klassdiagram (det finns verktyg som kan generera dem från koden), dels ska du skriva minst **fem** scenarion och rita tillhörande sekvensdiagram, dels ska du rita minst ett tillståndsdiagram.

Ett scenario är en textbeskrivning där man steg för steg i textform beskriver ett användarscenario. T.ex. aktören plockar upp ett aztekiskt guldmynt från en piratkista. En dov röst hörs som förbannar aktören. Ett spöke kommer fram ur skuggorna och slåss med aktören. Spöket dör.

Till varje scenario hör ett eller flera sekvensdiagram.

Tänk på att varje scenario bara kan illustrera en gren av en if-sats, eller ett visst antal varv i en loop.

Identifiera och redovisa **de scenarion som är spelmässigt eller programmeringstekniskt intressanta** i just ditt spel. Programmeringstekniskt intressanta kan t.ex. vara ett scenario där man vinner. Spelmässigt intressanta kan vara ett scenario där man får en en avgörande ledtråd till hur man ska klara spelet.