Tentamensregler

Hjälpmedel

Följande får tas med på tentan:

- En bok om c++. För boken gäller följande regler:
  - Kommentarer/noteringar som direkt rör text och exempel på sidan i fråga får finnas i sidmarginalen.
  - Egna sidflikar för att enkelt kunna hitta t.ex. de olika kapitlen är tillåtna.
  - Inga extra ark eller lappar, lösa eller fastsatta, får finnas.
  - Tomma sidor, insidan av pärmarna, försättsblad, etc., får inte innehålla programkod.
- Ett A4-ark med valfritt innehåll på vardera sidor (går ej att ersätta med två enkelsidiga ark).
- Penna för att anteckna under tentan. Ni kommer förses med blanka papper

Följande får INTE tas med:

- Elektroniska prylar. Dit hör till exempel miniräknare, mobiltelefon, hörlurar, tangentbord, smartklocka etc.

Utloggning

När du är nöjd med ditt betyg (som står i tentaklienten) kan du avsluta tentan. Stäng alla program och spara alla filer. Sedan loggar du ut som vanligt i menyn. Lämna inte din plats innan vanliga inloggningsskärmen syns.

Bedömning

Tentan har fyra praktiska uppgifter. Två praktiska uppgifter är på grundnivå. Där visar du att du uppnår kraven för godkänt. De två praktiska uppgifterna är på djupare nivå. I dessa kan du visa att du nått upp till kraven för ett högre betyg.

För betyg 3 på tentan krävs lösning av en uppgift på grundnivå. För betyg 4 krävs dessutom lösning av en påbyggnadsuppgift och för betyg 5 krävs en grunduppgift och två påbyggnads-uppgifter.

För att en uppgift ska anses godkänd krävs följande:

- att du noga följt alla instruktioner och krav ställda i uppgiften
- din kod följer god programmeringsstil (se labseriens rättningsguide)
- att klasser har ett tydligt ansvar och funktioner har en väl definierad uppgift
- att din kod har bra inkapsling och resurshantering

Bonus från labserien

Om du har bonus från labserien minskas antalet påbyggnadsuppgifter som krävs med ett, dvs för betyg 4 krävs endast en grunduppgift och för betyg 5 krävs en av varje. Bonusen är endast giltig det år den erhölls.

Frågor om uppgifter

Frågor om tentan i stort eller uppgiftspecifika frågor ska ställas via tenta-klienten. Detta för att vi ska ha en historik av konversationen samt för att vi ska kunna ge samma hjälp till olika studenter.

Systemfrågor

Om du har systemfrågor som t.ex. problem med tentaklienten eller terminalen räcker du upp handen så kommer en assistent och hjälper till.

C++ referens

Det finns tillgång till valda delar av `cppreference.com`. Du måste starta webbläsaren via skrivbordsikonen “Web access” för att komma åt sidan.

Alias för kompilering

Under tentan finns det tre alias att använda sig av för kompilering med c++17:
`w++17` Rekommenderas!
`e++17` Kompilering med alla varningar som fel.
`g++17` Kompilering utan varningar.

Uppgift 1 - Molnigt spel - Grundnivå

En känd profil på SMHI är Lisa Frost. Hennes påhittade kollega Henri Storm har fått en ny innovativ datorspelsidé för flera spelare. Grundtanken är att varje spelare ska styra ett moln som svävar över spelvärlden. Om spelarens moln kolliderar med en annan spelares moln kommer molnen att slås ihop till ett större moln. Spelaren med det minsta molnet förlorar och den andre spelaren fortsätter spelet med det större molnet. Senare planerar Henri flertalet powerups och annat väderrelaterat individuellt beteende för olika moln men just nu gäller att få till grunden för ett moln.

Ett moln behöver hålla reda på spelarens namn, sin position och sin radie. Vi antar att alla moln är perfekt cirkulära. Ett moln konsumerar ett annat genom att dess radie ökar med 1 och det andra molnets radie sätts till 0. Henri föreställer sig test-koden i filen `cloud.cc` för att hantera moln som kolliderar.

Henri har inte själv tillräcklig kunskap i programmering för att lösa detta, och han är inte heller säker på hur funktionerna `overlap` och `consume` ska fungera. Turligt nog har Henris dotter Henrietta som går i högstadiet just lärt sig Pythagoras sats och tillsammans kommer de fram till nedan formler. Du har fått uppgiften att skapa en klass `Cloud` som fungerar så som Henri vill.

Överlapp: Cirkel 1 överlappar cirkel 2 om $(x_1 - x_2)^2 + (y_1 - y_2)^2$ är mindre än $(r_1 + r_2)^2$.
Sammanslagning: $r_{this} = r_{this} + 1$, $r_{other} = 0$

Självklart gäller att molnklassen ska följa goda konventioner och goda objektorienterade principer i avsikt att vara så användbar och utvecklingsbar som möjligt. Korrekt filuppdelning krävs.

Den givna test-koden ska *utan att modifieras* producera följande utdata (utskrift av avslutande nollor sker här, men är inte ett krav):

Körexempel (användarinmatning i fet stil)

```
$ ./a.out
Klas(5.000000), Eric(1.000000)
Klas(6.000000), Eric(0.000000)
Klas(6.000000), Maria(8.000000)
Klas(0.000000), Maria(9.000000)
Maria(9.000000), Nadim(1.000000)
Maria(9.000000), Nadim(1.000000)
```

Uppgift 2 - Resultathantering - Grundnivå

Fredrika som pluggar datateknik ska just öva inför tentamen i programmering. Hon har sedan tidigare noterat att det ofta tar lång tid innan kurser blir "slutförda" i LADOK trots att alla poäng är klara. Hon bestämmer sig därför för att som övning skriva ett litet personligt LADOK som automatiskt räknar ut när en kurs är slutförd. Hon tänker sig följande klasser:

Klassen `Course` håller reda på en kurs med kurskod och totalt antal poäng kursen ger. Till en kurs ska dess dessutom gå att lägga till examinationsmoment. Klassen `Part` representerar ett examinationsmoment och har ett namn, ger ett antal poäng och ett betyg.

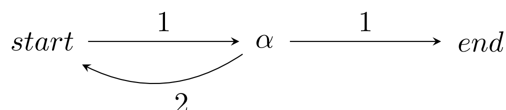
Ett personligt ladok-utdrag byggs upp direkt i huvudprogrammet (se `pladok.cc`) med hjälp av klasserna och skrivs till slut ut prydligt inklusive markering om kursen är slutförd eller ej. Se kodexempel i `pladok.cc` och körexempel nedan.

Självklart gäller att *båda* klasserna ska följa goda konventioner och goda objektorienterade principer i avsikt att vara så användbara och utvecklingsbara som möjligt. Du får frångå god konvention på punkten att all kod får ligga i samma fil.

Körexempel (användarinmatning i fet stil)

```
$ ./a.out
[X]TSIU05:
 - UPG1 (2hp): G
 - LAB1 (2hp): G
 - TEN1 (4hp): 5
[ ]TDIU20:
 - LAB1 (3hp): 3
[X]TAIU10:
 - TEN1 (6hp): 4
```

Uppgift 3 - Tillståndsmaskin - Påbyggnad för högre betyg



Tillståndsmaskiner är en abstrakt modell som kan användas inom datorvetenskapen. En sådan maskin består av ett antal tillstånd. Ett tillstånd kan påverkas av in-signaler. Varje giltig in-signal leder till ett nytt tillstånd. Klasserna som krävs för att skapa en tillståndsmaskin som kan känna igen sekvenser av heltal beskrivs nedan. Komplettera `state.cc` med klasserna.

BaseState

- Denna klass deklarerar funktionen **eval** som tar en parameter av typen `std::istream &`

State (Härleds från BaseState)

- Klassen kan hantera alla signaler i intervallet $[0 - 9]$. En `std::vector<BaseState*>` dimensionerad för 10 olika signaler används som datalagring. Index i vektorn motsvarar signalvärdet och innehållet (pekaren) vart respektive signal leder. Oanvända platser i vektorn lagrar `nullptr`.
- Klassen implementerar `eval` (från `BaseState`). Funktionen hämtar nästa signal (heltal) från strömmen och slår upp vilket tillstånd som maskinen ska gå vidare till. Om signalen leder till ett nytt tillstånd ska `eval` anropas på det nya tillståndet. I alla andra fall ska funktionen returnera utan att något mer händer.
- Klassen har en medlemsfunktion `add_signal` som tar emot en pekare av typen `BaseState` och ett heltal. Heltalet representerar signalen och pekaren är den pekare som ska associeras med signalen. Dessa sparas sedan undan i din vektor. Valfritt undantag kastas om signalen är utanför givet intervall (om indexet inte finns i vektorn).

EndState (Härleds från BaseState)

- Klassen implementerar `eval` (från `BaseState`). Texten `Sequence is correct` ska skrivas ut om strömmen inte har några tecken kvar. **Tips:** Tänk på att en inläsning från en ström är konverterbar till `bool` baserat på om inläsningen lyckades eller ej.

Exempel: Antag att vi har tillståndsmaskinen som syns i den inledande figuren. Som in-signaler har vi talsekvensen `1 2 1 1`. Vi börjar i *start* tillståndet och läser första talet `1`. Vi följer då pilen märkt `1` som leder till α. Nästa tal i sekvensen är `2`. När vi följer motsvarande pil hamnar vi återigen i *start* tillståndet. Efter det läser vi `1` igen, vilket tar oss tillbaka till α. Nästa tal (som också är sista) är `1` vilket leder oss till *end*. Nu har vi nått slutet på talsekvensen och befinner oss i *end*, vilket betyder att talsekvensen matchade!

Körexempel:

```
tried to add signal out of range
testing sequence: 1 1
Sequence is correct
testing sequence: 1 2 1 2 1 1
Sequence is correct
testing sequence: 1 1 1 2 1 1
testing sequence: 1 1 1
testing sequence: 1 5 1
testing sequence: 1 25 1
```

Uppgift 4 - Referensräkning - Påbyggnad för högre betyg

Att räkna referenser är ett vanligt sätt att hantera objekt som delas mellan många olika aktörer i din kod. Referensräkning innebär att vi lagrar en variabel av heltalstyp tillsammans med ett objekt och när någon aktör refererar till objektet så inkrementeras räknaren. Om referensen förstörs så dekrementeras räknaren.

I denna uppgift ska du implementera 2 klasser.

Object som representerar det lagrade objektet. Denna klass lagrar en `std::string` som är datan och ett heltal som är räknaren. Datan måste initieras med en lämplig konstruktor. Lägg märke till att räknaren *alltid* initieras till 0.

Klassen har 5 medlemsfunktioner:

- `data()` returnerar en referens till den lagrade strängen.
- `get()` returnerar ett `Reference` objekt (se beskrivning nedan) som "hänvisar" till detta objekt.
- `increase()` och `decrease()` inkrementerar respektive dekrementerar räknaren.
- `count()` returnerar nuvarande antal aktiva referenser (värdet som lagras i räknaren).

Reference som representerar en "referens" till ett `Object`. Den lagrar en *pekare* till ett `Object`. Den har följande medlemsfunktioner:

- En konstruktor som tar en `Object`-pekare. Denna parameter måste ha default-värdet `nullptr`.
- De fem speciella medlemsfunktionerna (kopierings- och flytt-konstruktion, kopierings- och flytt-tilldelning, samt destruktor). Notera särskilt att kopiering ska anropa `increment()` på det `Object` som pekaren hänvisar till (om den inte är `nullptr`). Flytt ska **inte** modifiera räknaren. Destruktorn ska anropa `decrement()` genom pekaren till `Object` (om pekaren inte är `nullptr`).
- Funktionen `data()` som returnerar en referens till strängen som `Object`-pekaren hänvisar till.

För att klara denna uppgift måste du uppfylla följande krav:

- Du måste använda kopierings-konstruktorn när du implementerar kopierings-tilldelnings-operatorn.
- De givna testfallen ska fungera utan att modifieras.
- Implementationen ska filuppdelas korrekt i `object.h` och `object.cc`. Båda klasserna placeras tillsammans i sin header resp. implementationsfil.
- Huvudprogrammet i `object_main.cc` ska fungera utan modifikationer (men var snäll och lämna in den givna filen).

Det finns givna testfall i `object_main.cc`.

Tips: En av dessa klasser behöver använda en fördeklaration (forward declaration) eftersom det finns ett cirkulärt beroende.