TDP004 - Tenta

2024-01-10

Regler

- All kod som skickas in för rättning ska kompilera och vara väl testad.
- Inga elektroniska hjälpmedel får medtas. Mobiltelefon ska vara avstängd och ligga i jacka eller väska.
- Inga ytterkläder eller väskor vid skrivplatsen.
- Student får lämna salen tidigast en timme efter tentamens start.
- Vid toalettbesök ska pauslista utanför salen fyllas i.
- All form av kontakt mellan studenter under tentamens gång är strängt förbjuden.
- Böcker och anteckningssidor kan komma att granskas av assistent, vakt eller examinator under tentamens gång.
- Frågor om specifika uppgifter eller om tentamen i stort ska ställas via tentasystemets kommunikationsklient.
- Systemfrågor kan ställas till assistent i sal genom att räcka upp handen.
- Endast uppgifter inskickade före tentamenstidens slut rättas.
- Ingen uppgift kan kompletteras under tentamens sista kvart.
- En uppgift kan som regel kompletteras tills den är antingen "Godkänd" eller "Underkänd". En uppgift bedöms som "Underkänd" om ingen markant förbättring skett sedan tidigare inlämning.
- Kompilerande kod, fullständig kravuppfyllnad och följande av god stil och goda konventioner enligt god programmeringssed är krav för att en uppgift ska bedömas "Godkänd".

Hjälpmedel	En C++-bok (t.ex. Stroustrup) Ett A4-ark med egna anteckningar

Information

Betygssättning vid tentamen

Tentamen består av ett antal uppgifter på varierande nivå. Uppgifter som uppfyller specifikationen samt följer god sed och konventioner ges omdömet "Godkänd". Annars ges omdömet "Kompletteras" eller "Underkänd". Tentamen kräver två godkända uppgifter för betyg 3. Alla betygsgränser ses i tabell 1 och 2. *För betyg 3 har du alltid hela tentamenstiden, varken mer eller mindre.* (För student som från LiU fått rätt till förlängd skrivtid förlängs betygsgränserna i proportion till den förlängda skrivtiden.)

Tid	Lösta uppgifter	Betyg	Tillgodo
4 h	3	5	inget
2.5 h	2	4	inget
4 h	2	3	inget
4 timmar	1	|	löst uppgift

Tabell 1: Betygsättning vid förtentamen (dugga), 4 uppgifter ges

Tid	Lösta uppgifter	Betyg
3 h $+B$	3	5
4 h $+B$	4	5
4 h $+B$	3	4
2 h $+B$	2	4
5 timmar	2	3

Tabell 2: Betygsättning vid sluttentamen, 5 uppgifter ges

Bonustid ($+B$)

All bonustid gäller endast under den första ordinarie tentamen i samband med kursen (januari). Varje moment i kursen som ger bonus ger 5 minuter extra tid för högre betyg på sluttentamen, upp till maximalt 45 minuter. Detta är markerat med $+B$ i tabellen där bonus räknas.

Tillgodoräknanden

Tillgodoräknanden gäller endast under den första ordinarie tentamen i samband med kursen. Om du på förtentamen endast lyckas lösa en uppgift kan du tillgodoräkna motsvarande uppgift på sluttentamen (se tabell 1). Du har då bara en uppgift kvar till betyg 3. För högre betyg (om du löser mer än en uppgift på förtentamen) kan du inte tillgodoräkna något. Om du på sluttentamen löser en andra uppgift snabbt och vill sikta på högre betyg kan du lösa den tillgodoräknade uppgiften "igen", men den räknas endast mot högre betyg, **inte** som andrauppgift för betyg 3.

Inloggning

Logga in på datorn i labbsalen med ditt liu-id och lösenord. Detta är samma inloggningsuppgifter som du använder i Lisam.

Skrivbordsmiljön

När du kommit in i tentasystemet har du en normal skrivbordsmiljö (Mate-session i Ubuntu). Efter en stund kommer även din kommunikationsklient att dyka upp automatiskt. Startmenyn

är nedskalad till enbart det som examinator bedömt relevant. Andra program kan fortfarande finnas tillgängliga genom att starta dem från en terminal. Observera att en del program som använder nätverkstjänster inte fungerar normalt, eftersom nätverket inte är åtkomligt.

När du är inloggad är det viktigt att du har tentaklienten igång hela tiden. Om den inte dykt upp fem minuter efter inloggning och inte heller när du startar den manuellt från menyn (fisken) tar du kontakt med assistent eller vakt i sal.

Terminalkommandon

`e++17` används för att kompilera med "alla" varningar *som fel*.
`w++17` används för att kompilera med "alla" varningar. **Rekommenderas.**
`g++17` används för att kompilera `utan` varningar.
`valgrind --tool=memcheck` används för att leta minnesläckor.

C++ referenssidor

På tentan har du tillgång till referenssidorna på `http://www.cppreference.com/` via webbläsaren Chrome. Starta `chromium-browser` i terminalen eller välj lämpligt alternativ från startmenyn. Observera att allt utom referenssidorna är avstängt. Om du inte kan komma åt en sida du tycker hör till referenssidorna (som kanske blockerats av misstag) kan du skicka ett meddelande via tentaklienten. Tag hjälp av assistent i sal om det inte fungerar.

Givna filer

Eventuella givna filer finns i katalogen `given_files`. Denna underkatalog är skrivskyddad, så det är ingen risk du råkar ändra på dessa filer. Skrivskyddet gör dock att du måste kopiera in de givna filer du vill använda till tentakontots hemkatalog. Hur du listar och kopierar filer ska du kunna. Hemkatalogen står du i från början, och du kommer alltid tillbaka till den genom att bara exekvera kommandot `cd` i terminalen.

Avslutning

När dina uppgiftsbetyg och ditt slutbetyg i kommunikationsklienten stämmer överens med det du förväntar och du är nöjd, eller när tentamenstiden är slut, är det dags att logga ut. Hinner du inte se ditt betyg får du höra av dig till examinator via epost efter tentamen.

Avsluta alla öppna program och logga ut ur datorn. Lämna inte datorn innan du ser att du är utloggad.

Uppgift 1 - klasser och operatorer

Nummer och tal är en väldigt användbar konstruktion som förekommer i de flesta aspekter av våra liv. Det finns ett antal olika typer av nummer och en av dessa kallas *ordinaler*. En ordinal är ett nummer som representerar en *ordning*. Exempel på ordinaler är *första*, *andra*, *tredje* o.s.v.

I matematiken definierar man ordinalerna rekursivt genom att först definiera talet `0` som en tom lista, d.v.s. `{}` och sedan låta den `n`:te ordinalen definieras som en lista av alla tal som den är större än. Detta betyder alltså att talet `1` definieras som `{0}`, talet `2` definieras som `{0, 1}` o.s.v.

Inkrement (även kallat *successor funktionen*) av ordinaler definieras då helt enkelt som att man lägger till sig själv i listan för att få nästa tal, medan addition definieras som upprepad inkrement.

I denna uppgift ska du skapa en klass `Ordinal` som representerar en s.k. ordinal. Den ska uppfylla följande krav:

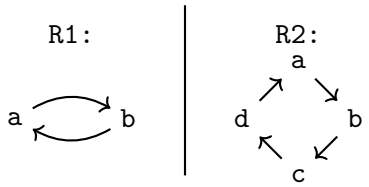
- Ska innehålla en lista av `unsigned` värden.
- Den ska gå att default-initiera så att den interna listan är tom (d.v.s. att den representerar talet `0`).
- Det ska finnas en medlemsfunktion `successor()` som returnerar nästa ordinal.
- Båda varianterna av `operator++` ska finnas. Dessa operatorer omvandlar nuvarande tal till dess successor. Ett sätt vi kan implementera detta på är att helt enkelt lägga till nuvarande storlek på listan i listan.
- Definiera `operator+` när man adderar två ordinaler `lhs` och `rhs`. Detta implementeras genom att upprepat tillämpa inkrement operatorn på en kopia av `lhs` lika många gånger som antalet element i listan inuti `rhs`.
- Det ska gå att skriva ut ordinalerna som en komma separerad lista m.h.a. `operator<<` (se körexempel).

Körexempel

```
0:
1: 0
2: 0, 1
2: 0, 1
3: 0, 1, 2
4: 0, 1, 2, 3
```

Uppgift 2 - Klasser

I datavetenskapen finns ett teoretiskt koncept som kallas *relationer*. Detta är en typ av tabell som associerar två objekt med varandra. Ofta ritas de som diagram, t.ex. såhär:



Här ser vi exempel på två relationer: `R1` och `R2`. Varje pil representerar att ett objekt relateras till ett annat. I `R1` ser vi att `a` relaterar till `b` och vice versa. Medan i `R2` ser vi att `a` relaterar till `b`, medan `b` relaterar till `c` o.s.v.

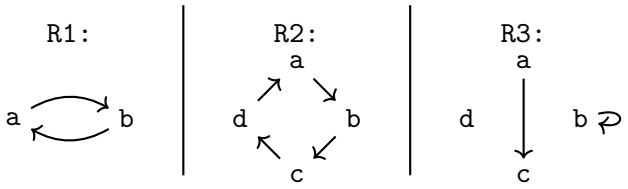
I den här uppgiften ska du skapa en klass `Relation` för att implementera relationer. Den ska uppfylla följande krav:

- Den ska innehålla en vector. I denna vector lagras par av karaktärer (`char`) som representerar relationen. `R1` från diagrammet ovan skulle alltså innehålla paren `'a'`, `'b'` och `'b'`, `'a'`.
- Det ska finnas en `add_relation()` medlemsfunktion som tar ett par och lägger in den i vektorn. Notera att det *får* förekomma fler än en pil per par av objekt.
- Det ska finnas en `to_string()` medlemsfunktion som returnerar en sträng representation av relationerna (se körexempel).

Utöver dessa krav ska det även finnas en medlemsfunktion vid namn `compose()` som tar ett `Relation` objekt som parameter och returnerar ett nytt `Relation` objekt som är den s.k. *kompositionen* av nuvarande relation med den som skickades in.

Två relationer `A` och `B` komponeras till en *ny* relation `C` genom att gå igenom alla par av pilar från `A` och `B`: om pilen i `A` slutar på samma objekt som pilen i `B` börjar på så lägger man in en relation i `C` som börjar på start objektet i `A` pilen och slutar på slut objektet i pilen från `B` (det finns pseudo kod för detta givet i `uppgift2.cc`).

Exempel: Här nedanför kan vi se hur kompositionen (`R3`) av `R1` och `R2` ser ut:



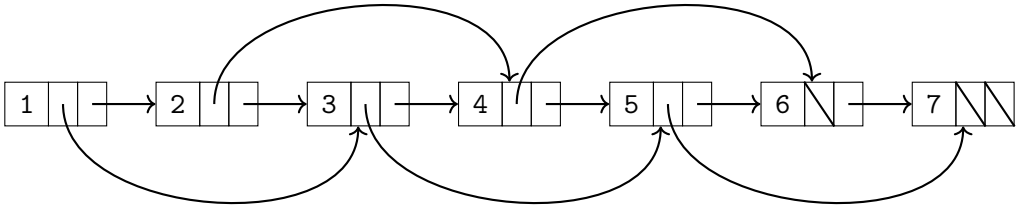
Notera alltså att det finns en pil från `a` till `b` i `R1` och sedan finns det en pil från `b` till `c` i `R2`. Detta betyder att vi i `R3` har en pil från `a` till `c`.

Körexempel

```
r1: a->b b->a
r2: a->b b->c c->d d->a
r3: a->c b->b
```

Uppgift 3 - Minneshantering

I `uppgift3.cc` finns det en given implementation för en något optimerad länkad lista som vi kallar en `Jump_List`. Skillnaden på en vanlig länkad lista och vår `Jump_List` är att varje nod har två pekare istället för en. En `next` pekare som precis som en vanlig länkad lista pekar på nästa element. Men sen finns det också en `jump` pekare som pekar två element framåt. Detta tillåter oss att söka igenom listan mycket snabbare eftersom att vi i regel kan hoppa över vartannat element (se bild).



Din uppgift är att implementera följande speciella medlemsfunktioner för `Jump_List`:

- Kopieringskonstruktorn
- Kopieringstilldelningsoperatorn
- Flyttkonstruktorn
- Flyttilldelningsoperatorn

Kom ihåg att listan måste hela tiden bibehålla den givna strukturen för att `at()` funktionen ska fungera ordentligt.

Krav: Det får inte finnas några minnesläckor.

Krav: Du måste utöka det givna testprogrammet så att de testar alla de speciella medlemsfunktionerna.

Tips: Kopieringstilldelningsoperatorn kan återanvända kopieringskonstruktorn genom att skapa en kopia som en lokal variabel.

Tips: Du får implementera kopieringen iterativt eller rekursivt, det är upp till dig, men en rekursiv lösningen är något simplare.

Uppgift 4 - Polymorfi

Du och dina vänner har bestämt er för att utveckla ett enkelt äventyrsspel. En viktig aspekt av ert spel är att samla på sig diverse olika föremål och samla dem i en väska. I den här uppgiften ska du skapa en klasshierarki för olika föremål samt vapen.

I denna uppgift ska du implementera följande klasser:

Item Agerar som basklass för hela hierarkin. Denna klass ska ha en datamedlem vid namn `weight` av typen `double`, med en tillhörande getter-funktion `get_weight()` som returnerar vikten. Det ska även finnas en lämplig konstruktor som sätter `weight`.

Utöver detta ska den även ha två virtuella funktioner: `print(std::ostream& os)` som är pure-virtual (abstrakt) och medlemsfunktionen `print_info(std::ostream& os)`.

`print_info()` ska skriva ut `(<weight> kg)` på `os` där `<weight>` ersätts med det faktiska värdet av datamedlemmen `weight`. Notera att denna funktion inte ska gå att anropa utanför klasshierarkin.

Shovel ärver från `Item`. Klassen ska ha en default konstruktor som sätter `Item::weight` till `3.0`. `Shovel` ska även överlagra `print()` funktionen så att den skriver ut strängen `"a shovel "` på `os` och sedan anropa `print_info()`.

Weapon ärver från `Item`. Klassen ska ha datamedlemmen `damage` av typen `double`. `Weapon` ska ha en lämplig konstruktor som initierar `damage` och `Item::weight`. Det ska även finnas en `get_damage()` medlemsfunktion som returnerar värdet av `damage`.

Denna klass ska överlagra `print_info()` så att den skriver ut `"dealing <damage> damage "` på `os` där `<damage>` ersätts med faktiska värdet av `damage` variabeln. Sedan ska den anropa basklassens variant av `print_info()`.

Sword ärver från `Weapon`. Klassen har datamedlemmen `name` som är av typen `std::string`. Det ska finnas en lämplig konstruktor som initierar `name`, `Weapon::damage` och `Item::weight`.

`Sword` överlagrar `print()` så att den skriver ut `"A sword named '<name>' "` på `os` där `<name>` ersätts med det faktiska namnet. Sedan ska den anropa `print_info()`.

Bow ärver från `Weapon`. Klassen ska använda samma konstruktor som `Weapon` och överlagra `print()` så att den skriver ut strängen `"A bow "` på `os` och sedan anropar `print_info()`.

Det finns början på ett givet testprogram i `uppgift4.cc` som du ska modifera så att det fungerar som tänkt.

Krav: Det ska inte finnas några minnesläckor.

Krav: `Weapon::get_damage()` får **inte** vara virtuell och får **inte** förekomma i `Item`.

Uppgift 5 - STL

Det är ett förhållandevis vanligt problem inom programmering att ta emot en lista av värden och ta bort alla dubletter. I denna uppgift ska du lösa en något försvårad variation av det problemet vilket är att ta bort alla dubletter men samtidigt bibehålla den inmatade ordningen på de unika värdena.

Exempel: Antag att användaren matar in `can you can a can as a canner can can a can` då ser vi att orden `can` och `a` upprepas flera gånger så programmet ska ta bort alla förekomster av dessa utom de första, sedan ska följande skrivas ut: `can you a as canner`. Notera alltså att resultatet som skrivs ut är sorterat enligt inmatnings ordningen.

Krav: För att lösa detta problem måste du använda STL algoritmer och lämpliga behållare. Inga loopar, rekursion eller `std::for_each` får förekomma.

Krav: Du får endast skapa *en* behållare i din lösning.

Tips: Skapa en behållare som innehåller ett `std::pair<int, std::string>` där första värdet representerar vilket index ordet var i inläsningsordningen och det andra värdet är ordet i fråga. Med hjälp av detta kan du alltså ändra ordningen fritt och sedan nyttja det första värdet i paret för att återställa ordningen genom sortering.

Körexempel (fetstil är användarinmatning)

```
$ ./a.out
can you can a can as a canner can can a can <ctrl+D>
can you a as canner
```

```
$ ./a.out
I bet they bet on black <ctrl+D>
I bet they on black
```

```
$ ./a.out
d a b z b c a <ctrl+D>
d a b z c
```