Tentamen i TDP004
Objektorienterad Programmering
Lösningsförslag

Datum:	2009-08-24
Tid:	14-18
Plats:	SU-salar i B-huset.
Jour:	Per-Magnus Olsson, tel 285607 Jourhavande kommer att besöka skrivsalarna ungefär varje timme under skrivtiden.
Hjälpmedel:	Teoretisk del: Inga. Praktisk del: Den C++ information som finns i systemet.
Betygsättning:	Max antal poäng: 48 med 24 poäng vardera på teori och praktikdel.

Poäng	**Betyg**
41-48	**5**
33-40	**4**
25-32	**3**
0-24	**U**

Anvisningar:	Börja med den teoretiska delen. När du är klar med den lämnar du in den och får den praktiska delen. När du har lämnat in den teoretiska delen kan du inte återvända till den. Skriv svaret på varje teoretisk uppgift på ett separat blad. Uppgifterna är inte ordnade efter svårighetsgrad.

Lycka till!

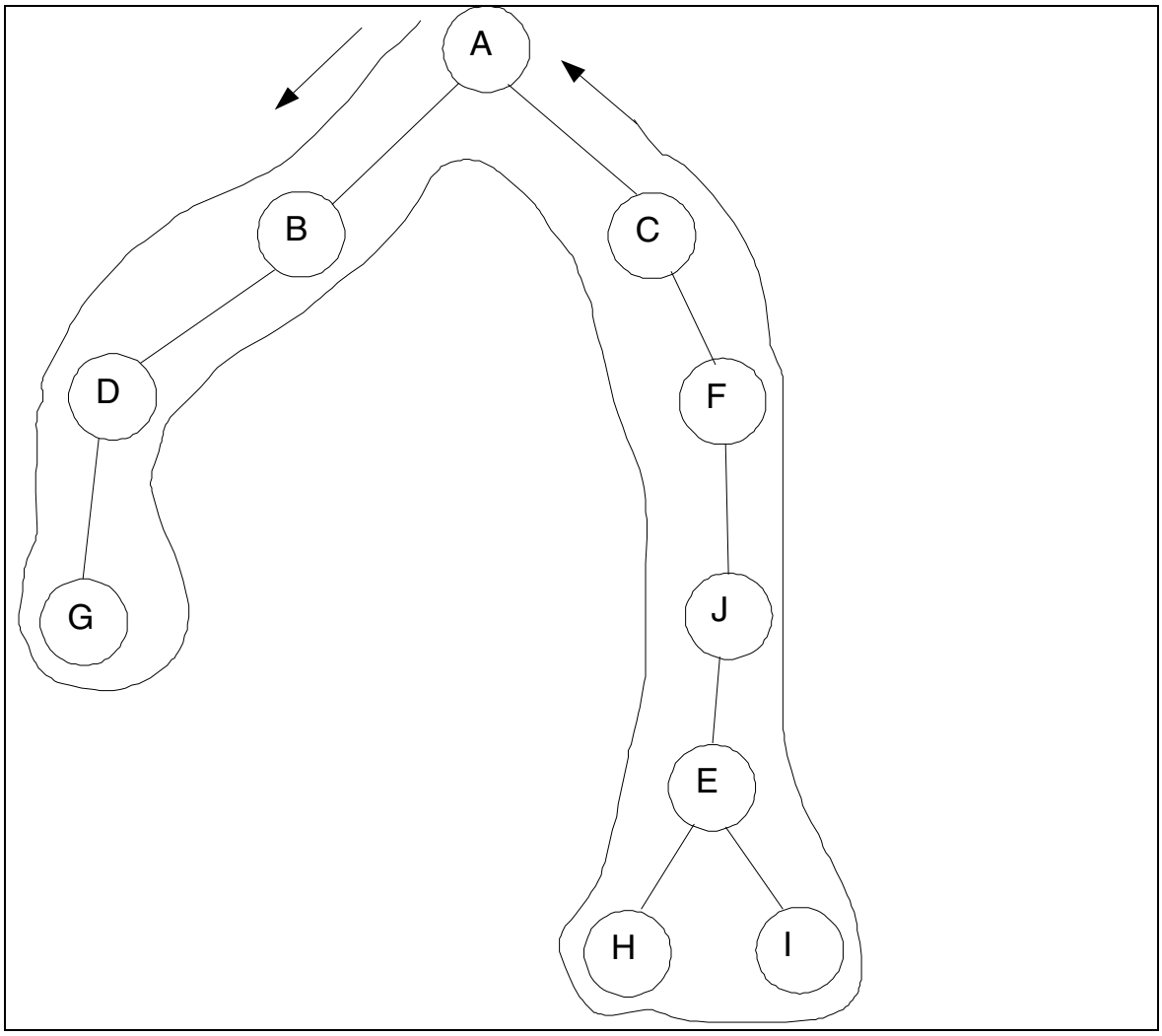
TDP004 Objektorienterad Programmering
Teoretisk del

1. Inom objektorientering används ofta termerna abstraktion, arv och inkapsling. Ge exempel på andra sammanhang där termerna används och motivera kortfattat dina exempel. (6p)

2. a) I större projekt är det vanligt att en kodstandard används. Vilka 4 saker tycker du är viktigast i en kodstandard? Du får ett poäng per sak samt en poäng per sak för bra motivering. (8p)

3. Ge förslag på sätt att rätta eventuella fel på följande kodrader. Är det inget fel så behöver du inte skriva något. (3p)
   a) `int int_p= new int;`
   b) `double & double_r;`
   c) `cout >>` ”Hejsan”;

4. (Svårare uppgift) Inom matematiken finns ett koncept som kallas *thread index*, vilket betyder att man lagrar i vilken antalet barn för alla noder i ett träd som symboliserar en graf. I bilden nedan markeras en nod med en rund ring och en nods barn är alla noder som ligger ”under” den i trädet. Man får reda på antalet barn genom att gå runt trädet i den riktning som pilarna i bilden visar: först A, sedan B, D, G, C osv tills man kommer tillbaka till A.
   Uppgiften fortsätter på nästa sida.

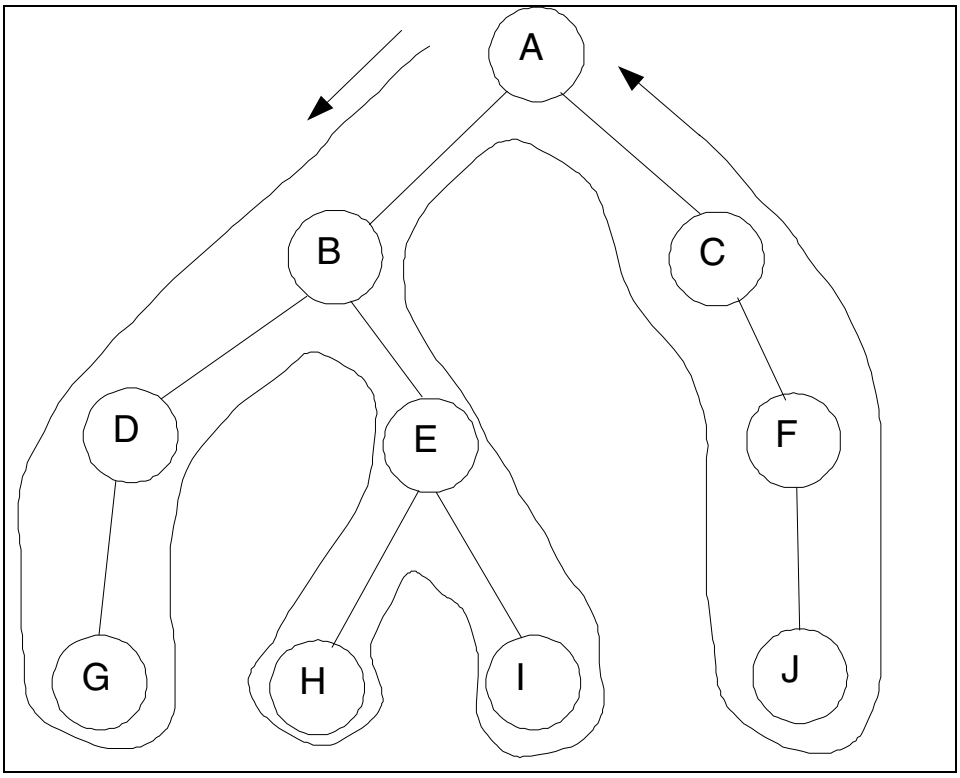


A	B	D	G	C	F	J	E	H	I
9	2	1	0	5	4	3	2	0	0

Man tänker sig att man har en viss ordning på noderna, vilket är den ordning som man kommer till dem när går igenom trädet, se tabellen ovan. Dock vet man inte hur många barn en nod har förrän man kommer till noden för *sista* gången. Därför måste man vänta med att lägga antalet barn tills man besöker noden för sista gången. När man har gått igenom trädet och kommit tillbaka till nod A vet man antalet barn för alla noder, vilket visas i ovanstående tabell. Till exempel har nod J har 3 barn: noderna E, H och I, varav nod E i sin tur har 2 barn: noderna H och I.

a) Ge förslag på hur man kan förknippa noden med antalet barn. (2p)
b) Välj en lämplig datastruktur för att lagra thread index. Även om tabellen ovan använder en tabell-liknande struktur behöver du inte göra det. Du ska endast ta hänsyn till ovanstående information. Motivera ditt svar. (2p)

c) Efter att thread index har beräknats används det som indata till funktioner som ofta modifierar trädet så att antalet barn ändras. Då måste man naturligtvis uppdatera thread index. I exemplet nedan är delträdet bestående av noderna E, H and I är flyttat till att ligga under nod B. Det är alltid ett enda delträd som flyttas.



A	B	D	G	E	H	I	C	F	J
9	5	1	0	2	0	0	2	1	0

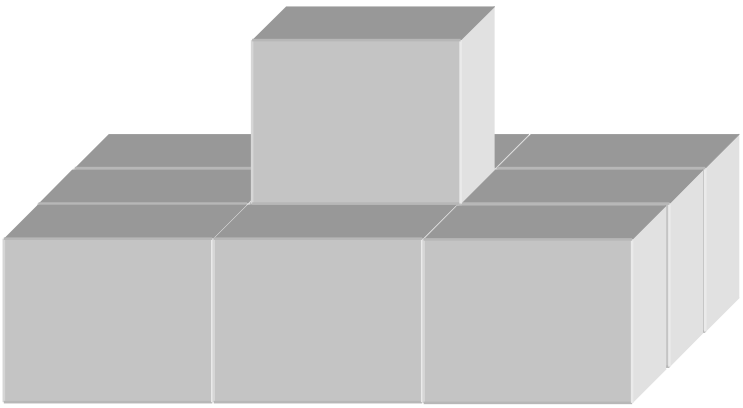
Ändringen ger ovanstående thread index där noderna E, H och I har flyttats. Eftersom sådana uppdateringar måste gå väldigt snabbt är det viktigt att välja en passande datastruktur. Trädet kan innehålla miljontals noder så man vill undvika att gå igenom hela trädet igen efter en uppdatering. Istället håller man reda på det delträd som flyttades och flyttar detta delträd till sin nya plats (och uppdaterar antalet barn för alla involverade noder. I bilden ovan har trädet uppdaterats och då måste antalet barn minskas för noderna C, F och J och antalet barn ökas för nod B). Nu ska du igen välja en lämplig datastruktur och den här gången ska du även ta hänsyn till informationen i deluppgifterna a) och b). Motivera ditt svar. (3p)

Praktisk del

1. En palindrom är ett ord eller en mening som blir likadan när den läses framlänges som baklänges. Ett exempel är ” dromedaren alpotto planerade mord” eller det lite mer kända ”sirap i paris”.
   A. Konstruera en funktion som kontrollerar om indatan i form av en std::string är en palindrom eller inte. Indatan kan bestå av blandade små och stora bokstäver. Det är endast en palindrom om bokstäverna som jämförs är båda antingen små eller stora. T ex är ”Sirap i Paris” inte en palindrom, men det är ”SiraP i PariS”. (3p)

   B. Gör en ny funktion där indatan är en palindrom om bokstäverna som jämförs är samma, oavsett om de är stora eller små. I det här fallet är ”Sirap i Paris” en palindrom. **Ledtråd**: i systemfilen cctype finns del användbara funktioner. (4p)

2. Pyramider är högsta mode (igen). Numera byggs de av identiska, rektangulära stenblock som staplas på varandra i lager. Lager 1 är alltid det översta lagret, lager 2 det näst översta osv. Bilden visar en pyramid med två lager. Lager 1 består alltid av ett stenblock och lager 2 består alltid av 9 stenblock, osv. Deluppgifterna kan lösas oberoende av varandra.



   A. Skriv en funktion som beräknar hur många stenblock som går åt för att skapa en pyramid med ett visst antal lager. Givet antalet lager (vilket är ≥ 0) i pyramiden ska den returnera antalet stenblock som behövs. Svaret ska sedan skrivas ut på skärmen tillsammans med en kort förklarande text. (4p)

   B. En del kunder tycker att det är vulgärt med toppiga pyramider och vill därför ha pyramider som inte har ett visst antal av de övre lagren. De vill t ex ha en pyramid endast bestående av lager 3-6. Skriv en funktion som beräknar antalet stenblock som behövs för en sådan pyramid. Svaret ska sedan skrivas ut på skärmen tillsammans med en kort förklarande text. (3p)

C. Din chef vill bygga en pyramid utanför kontoret av de överblivna stenblock som ligger och skräpar. Skriv en funktion som givet ett visst antal stenblock beräknar antalet lager i den största pyramid som kan byggas. Du kan anta att antalet stenblock är ≥ 0. Pyramiden ska börja med lager 1 och måste bestå av hela lager. Svaret ska sedan skrivas ut på skärmen tillsammans med en kort förklarande text. (5p)

3. I katalogen given_files finns filerna Oven.h samt Oven.cc, vilka modellerar en ugn. Kopiera filerna till ditt workspace. Klassen har en del fel och är inte så väl genomtänkt med tanke på objektorientering. Åtgärda felen och modifiera den så att den blir mer lämpad att användas i ett objektorienterat sammanhang. Tänk t ex på vanliga objektorienterade principer. Markera vad du har ändrat och motivera kortfattat varför (max 1 sida). Om du vill införa egna hjälp-funktioner eller nya klasser så ska dessa kortfattat förklaras och ges beskrivande namn. (5p)

Lösningsförslag teoretisk del

1. Exempel på abstraktion: tändningen i en bil som aktiveras genom nyckel i tändningslåset/knapp på instrumentbrädan. Exakt hur systemet fungerar är oklart för användaren, man kan tänka sig det som en svart låda.

Moderna fordon kan ses som exempel på inkapsling där det allra mesta informationen inte finns öppet tillgänglig, utan enbart tillgänglig genom specificerade interface.

Ett exempel på arv är det system för kategorisering av växter och djur som utvecklades av Carl von Linné.

2. Exempel på svar är: bra variabelnamn. Bra funktions-/varabel-/klassnamn (beskrivande, lagom långa). Kommentarer på rätt ställe i koden och på rätt abstraktionsnivå. Koden och kommentarerna korrekt indenterade.

3.
a) `int int_p = new int;` är fel därför att man inte kan tilldela den pekare som skapas i högerledet till den int som deklareras i vänsteledet. Detta är mest logiskt att ändra det till
`int* int_p = new int;` eller
`int int_p;` om det inte finns något värde att tilldela (men man bör fundera på att byta namn på variabeln om kodstandarden säger att suffuxet _p ska nvändas när det är en pekare.

b) `double & double_r;` kräver lite mera arbete. En referens till en double deklareras i vänsterledet, vilken måste tilldelas när den skapas. Det krävs en double som double_r kan tilldelas. En möjlig lösning är

```
double a;
double & double_r = a;
```

Det är naturligtvis lämpligt att a tilldelas ett värde innan tilldelningen på den andra raden sker.

c)`cout >>` ”Hejsan”; ändras till
`cout <<` ”Hejsan”; så att det blir en utskrift. Detta eftersom ”Hejsan” inte är ett giltigt variabelnamn.

4a) Det är lämpligt att spara de två ”komponterna” nodnamn samt antalet grannar tillsammans i tex en struct/klass eller ett `std::pair`. Den senare skulle kunna se ut enligt `std::pair<std::string, int>`.

b) Det finns bra argument både för `std::list` och `std::vector`. Båda har fördelen att man kan lägga till nya element sist (när man går igenom trädet) med `push_back`-funktionen. När man sedan ska lagra antalet barn så är det en fördel om man kan indexera elementet direkt, vilket med man med en `vector` men inte med `list`.

Eftersom trädet kan bli väldigt stort kan det vara en nackdel att `vector` dubblar minnesutrymmet så snart det tar slut eftersom minnet faktiskt kan ta slut. Om man vet antalet noder kan man använda `reserve`-funktionen som vector har. `std::list` har inte `reserve`-funktionen, men allokerar bara minne när ett nytt element skapas, vilket minskar problemet.

Möjligt datastrukturer skulle då vara
`std::list<std::pair<std::string, int> >` eller
`std::vector<Node_int_pair*>` där `Node_int_pair` är en klass/struct som man själv definierar.

Det är naturligtvis möjligt att ha en separat datastruktur för varje de olika komponterna, men man måste fortfarande på något sätt förknippa noden i den ena datastrukturen men antalet barn i den andra datastrukturen.

Eftersom `std::map` har pair som grundläggande element så är det naturligtvis mäjligt att även använda map som datastruktur om man använder nodens id som key och antalet barn som value. `std::multimap` är inte lämpligt eftersom den tillåter flera element med samma key. `map` har inte någon reserve-funktion, och den allokerar minne på liknande sätt som `list`. Eftersom `map` sorterar elementen automatiskt när de läggs till så blir prestandan sämre, och man har heller ingen nytta av sorteringen.

c)

Om antalet ändringar är stort och/eller stora delar av trädet ska flyttas är det lämpligt att använda `std::list` eftersom det finns färdiga `splice`-funktioner vilken flyttar dellistor inom samma eller olika listor. De stora fördelarna med `vector` (snabb iteration eftersom elementen lägger efter varandra i minnet samt omedelbar access genom []-operatorn) har man inte stor nytta av. Dock, om man själv skriver en effektiv funktion till `vector` som flyttar på element så kan man mycket väl använda `vector`. I det här fallet är det en stor nackdel att `map` sorterar elementen automatiskt, eftersom träden blir stora och bra prestanda krävs enligt uppgiften. Observera att dessa överväganden ska beaktas tillsammans med de från deluppgift a och b). Även här finns det bra argument för båda, det är motiveringarna som är viktiga.

Lösningsförslag praktisk del

1.

```
#include <string>
#include <algorithm>     //För deluppgift a.
#include <cctype>        //För deluppgift b.
```

a)

```
/*Oberoende av stora och små bokstäver.
== operatorn för std::string sköter sånt automatiskt.
*/
bool IsPalindrome(std::string indata)
{
     std::string reversedata = indata;
     reverse(reversedata.begin(), reversedata.end());

     return indata == reversedata;
}
```

b)

```
/* Iterera både framifrån och bakifrån samtidigt,
i varje steg jämförs de element iteratorerna refererar
till.
Om elementen är olika, så är det inte ett palindrom.
isupper() eller islower() fungerar lika bra.
*/
bool IsPalindromeCaseDependent(std::string indata)
{
     std::string::iterator forward = indata.begin();
     std::string::reverse_iterator reverse =
indata.rbegin();

     for(; forward != indata.end(); ++forward, ++reverse)
     {
          if( isupper(*forward) != isupper(*reverse) )
```

```
            {
                return false;
            }
        }
        return true;
}
```

2

a)

```
int Pyramid::NoBlocksNormal(int levels)
{
        int sum = 0;            /*Total number of blocks in the
                                pyramid so far. */
        int block_per_side; //The number of blocks per side.
        int i;                  //Iteration variable
        if(levels == 0)
        {
            return 0;
        }

        /* The i < levels +1 is necessary as the pyramid
        should contain exactly levels number of levels.
        For every increase in level, the number of blocks per
        side increases by 2. */
        for(i = 1, block_per_side = 1;
            i < levels +1;
            i++, block_per_side +=2)
        {
            sum += block_per_side * block_per_side;
        }
        return sum;
}
```

b) // Use the function from a) twice

```
int Pyramid::NoBlocksCapped(int upper_limit, int
lower_limit)
{
        if(upper_limit > lower_limit)
        {
            return 0;
        }
        return NoBlocksNormal(lower_limit) -
        NoBlocksNormal(upper_limit);
}

/* To see the diffent cases, use for example the following
test cases:
```

```
no_blocks = 10 (gives 2 levels as the levels require 1+9
blocks.
no_blocks = 11 (gives 2 levels as the third level requires
25 blocks).
*/
int Pyramid::MaxLevels(int no_blocks)
{
     int sum = 0;                    //Total number of blocks
for the pyramid so far.
     int block_per_side = 1;         //The number of blocks
per side.
     int level = 1;                  //Iteration variable
(current level)
     bool finished = false;

     if(no_blocks == 0)
     {
          return 0;
     }

     while(finished == false)
     {
          sum += block_per_side*block_per_side;

          if(sum == no_blocks)
          {
               //This pyramid required exactly noBlocks
blocks.
               finished = true;
          }
          else if(sum > no_blocks)
          {
               /*The last level used too many blocks.
               Remove the last level. */
               level--;
               finished = true;
          }
          else
          {
               //More levels can be added
               level++;
               block_per_side +=2;
          }
     }
     return level;
}
```

3. Exempel på saker att rätta till: ge rätt tillgänglighet till funktioner. Tänk igenom vilka variabler som böra var publika (antagligen inga i det här fallet). För de variabler som blir protected eller private, tänk igenom hur värden på dessa ska sättas. Behövs get-funktioner? Gör en egen klass Thermometer med ett väl definierat interface för att öka återanvändbarheten och flexibiliteten. Ändra hanteringen av gamla temperaturer så att inte minne behöver allokeras med `new`. Lägg till destruktor som deallokerar det minne som eventuellt har allokerats med new. Tänk igenom vilka funktioner/variabler ska vara publik/protected/private med tanke på framtida utbyggnade genom arv.