Laboration 1

Imperativ programmering

Laboration 1 omfattar fem mindre programmeringsuppgifter. Avsikten med dessa är att ge övning på i huvudsak så kallad imperativ programmering (procedurorienterad programmering), där variabler, tilldelning, repetitions- och villkorssatser utgör grunden. Uppgift 5 har även inslag objektorienterad/objektbaserad karaktär. Uppgifterna täcker bl.a. följande:

- grundläggande datatyper, pekare, std::vector (en fältliknande datastruktur), **struct** och **class**, länkad lista
- uttryck och satser
- funktioner, parameteröverföring
- grundläggande objektorienterade konstruktioner: klass, datamedlem, medlemsfunktion, konstruktor och destruktor
- in- och utmatning, interaktivt och från/till fil
- uppdelning av kod på flera filer, inkluderings- och implementeringsfil, hjälpmedlet make, make-filer
- konstruktion av testdata
- söka information via literatur, internet, direkthjälp på datorsystemet (man-sidor) och andra källor

Läs igenom specifikationerna för uppgifterna noga. Uppgifterna får anses vara väl specificerade och det finns i de flesta fall exempel på hur indata ska ges och hur utdata ska skrivas ut. I uppgiften att konstruera programmen ingår, helt naturligt, även egna beslut om sådant som inte uttryckligen specificeras, men följ specifikationen. *Fråga alltid din assistent om du känner dig osäker om en uppgifts innebörd!*

Laboration 1.2-1.3 är mycket enkla uppgifter. ***I och med uppgift 1.4 höjs svårighetsgraden märkbart.***

Uppgifterna i Laboration 1 motsvarar vad som normalt ingår i en grundkurs i programmering och utgör själva basen för programmering i de allra flesta imperativa och objektorienterade programmeringsspråk.

Endast en kort översikt ges på föreläsning (Fö 1–2) inför denna laborationsserie och mycket av stoffet lämnas till självstudier (repetition av förkunskaper och konvertering till grundläggande C++).

Problemlösningsfärdighet, dvs att kunna analysera ett problem och ta fram en programmerbar lösning, tioll exempel genom så kallad stegvis programförfining, förutsätts vara inhämtat i någon grundkurs.

Stilgranskning av Lab 1.1-1.3

För att snabbt få in en god programmeringsstil har vi s.k. *stilgranskning* av Laboration 1.1-1.3. Dessa uppgifter ingår *inte* i slutredovisningen av Lab 1 och alltså inte i betygssättningen. Det är dock obligatoriskt att lämna in Laboration 1.1-1.3 för granskning och det bör göras så snart möjligt och med alla filer på en gång. Varje uppgift får lämnas in för granskning en gång. På den lektion 2 ges möjlighet att diskutera frågor med anledning av erfarenheterna från de inledande laborationerna.

Slutredovisning av Laboration 1

Slutredovisning av Laboration 1 ska göras i form av *en* samtidig redovisning av uppgift 1.4 och 1.5. Koden ska se snygg ut även på papper, utan exempelvis långa rader som antingen klipps av eller vars slut skrivs ut med felaktigt indrag på en ny rad.

Redovisningar ska skickas till din assistent via ett skript, ~TDDC76/send_lab, och ett ifyllt IDA-omslag för laborationer ska lämnas in. Se anvisningarna för detta som finns i de allmänna anvisningarna till laborationerna.

Komplettering? Görs på hemarbetstid! Ska lämnas in senast 8 dagar efter att komplettering begärts!

Betygssättning

Laboration 1.4 och 1.5 betygssätts endast U/G/VG. För att kunna erhålla betyget **G** på Laboration 1 måste uppgifterna vara lösta på ett tillfredsställande sätt, inklusive programmeringsstil. Komplettering kan bli aktuellt om den första versionen inte uppfyller specifikationen och stilkrav. För att kunna få **VG** på laboration 1 ska deadline för inlämning strikt hållas och det ges endast en möjlighet att komplettera till godkänt och en sådan komplettering ska lämnas in senast inom 8 kalenderdager.

Tidplanering

Två tillfällen är avsatta för laboration 1.1-1.3, ett för laboration 1.4 och två för laboration 1.5 Se kursens schema på webben för närmare detaljer och deadline.

Kommer du efter ska du slutföra på hemarbetstid, inte på labbtillfällen avsedda för efterföljande labbar!

Testdata

Länkar till testdata finns på kursens laborationssida på webben. Observera att det i flera uppgifter ingår att du själv tar fram adekvata testdata.

Filnamn

Namnge källkodsfiler i enlighet med vad som anges för respektive uppgift, det är en förutsättning för att kunna redovisa.

Tips för framgång

Tänk bl.a. på följande då du skriver programmen:

- Var noga med att välja bra namn på variabler, funktioner, parametrar, etc., redan från början. Det ökar läsbarheten och minskar behovet av (onödiga) kommentarer.
- Var noga med den textuella utformningen: uppdelning på rader, insättning av tomma rader, insättning av mellanrum på rader, användning av parenteser, korrekt indrag från vänstermarginalen, etc.
- Fundera över vilken datatyp som förefaller lämpligast för t.ex. en variabel eller en funktionsparameter, då det finns valmöjlighet.
- Försök hålla ner antalet variabler i olika sammanhang, utan att för den skull tumma på logik och läsbarhet! Använd t.ex. inte en variabel för flera olika syften.
- Tänk på hur du skriver sammansatta uttryck, så att deluttrycken kommer i en logiskt riktig ordning och, i vissa fall, en säker ordning.
- Fundera över vilken typ av sats som förfaller lämpligast i ett visst sammanhang, t.ex. då du ska konstruera en repetition (passar **for**, **while** eller **do** bäst i just detta fall?) eller ett val (**if** eller **switch**?).
- Välj funktionsparametertyper med omsorg med tanke på semantik: inparameter, utparameter, in- och utparameter?
- Var ”**const**-korrekt”, dvs allt som är konstant ska deklareras med **const**.

Det finns en stilguide tillgängliga på kursens laborationssida på webben och de kodexempel som används på föreläsningar och annat är naturligtvis stilbildande.

1. In- och utmatning av heltal (stilgranskas)

Skriv programmet på en fil med namnet **lab1-1.cc**.

Programmet ska läsa in ett heltal, större än eller lika med 1, och skriva ut en tabell med tre kolumner, där alla heltal från 1 till det inlästa talet anges i decimal, oktal respektive hexadecimal form.

Kontroll av att inmatat värde är större än eller lika med 1 ska göras och om så ej är fallet ska ett felmeddelande skrivas ut och programmet ska begära ett nytt värde. Detta ska upprepas till dess ett korrekt värde ges, varefter tabellen skrivs ut. Programmet ska avslutas efter att tabellen skrivits ut.

Nedan visas ett körexempel där först ett felaktigt värde (-5) matats in och sedan ett korrekt värde (16):

```
Ge önskat slutvärde (minst 1): -5
Felaktigt värde, försök igen!
Ge önskat slutvärde för tabellen: 16

  DEC        OKT        HEX

    1          1          1
    2          2          2
    3          3          3
    4          4          4
    5          5          5
    6          6          6
    7          7          7
    8         10          8
    9         11          9
   10         12          a
   11         13          b
   12         14          c
   13         15          d
   14         16          e
   15         17          f
   16         20         10
```

De tre kolumnerna kan ha en enhetlig bredd, t.ex. 10 tecken.

Det är tillåtet att förutsätta att enbart heltalsvärden ges som indata (ingen komplicerad felhantering krävs alltså). Testa programmet noga för olika heltalsvärden.

Frivilligt övning: Intervall

Skriv programmet på en fil med namnet **lab1-1-extra.cc**.

Programmet ska läsa in även en undre gräns för tabellen. Den undre gränsen ska vara ett heltal större än eller lika med 1, vilket programmet ska kontrollera. Inläsningen ska ej gå vidare till inläsning av den övre gränsen förrän en korrekt undre gräns har givits. Programmet ska sedan läsa in och kontrollera att den övre gränsen är större än eller lika med den undre gränsen och inte gå vidare till utskrift förrän detta uppfyllts.

```
Ge önskat startvärde (minst 1): -5
Felaktigt värde, försök igen!
Ge önskat startvärde (minst 1): 5
Ge önskat slutvärde för tabellen (minst 5): 16

  DEC        OKT        HEX

    5          5          5
    6          6          6
    …          …          …
   15         17          f
   16         20         10
```

2. In- och utmatning av reella tal (stilgranskas)

Skriv programmet på en fil med namnet **lab1-2.cc**.

Programmet ska läsa in ett reellt tal (**double**) som anger en temperatur i *Kelvin* och sedan skriva ut motsvarande temperaturvärden i grader *Celsius* (Kelvin - 273.15) och grader *Fahrenheit* (1.8·Kelvin - 459.67). Detta ska upprepas till dess talet 0 matas in.

Om ett inmatat värde är mindre än 0, ska en felutskrift göras och en ny inmatning begäras.

Körexempel:

```
Ge en temperatur i Kelvin: 273.15
273.15 Kelvin motsvarar 0.00 grader Celsius eller 32.00 grader Fahrenheit.

Ge en temperatur i Kelvin: -1
Negativa kelvinvärden är ej tillåtna!

Ge en temperatur i Kelvin: 373.15
373.15 Kelvin motsvarar 100.00 grader Celsius eller 212.00 grader Fahrenheit.

Ge en temperatur i Kelvin: 0
0.00 Kelvin motsvarar -273.15 grader Celsius eller -459.67 grader Fahrenheit.

Slut.
```

Utskrivna värden ska ha två decimaler. Det är tillåtet att förutsätta att endast reella värden matas in.

Frivillig övning: Robust inmatning

Skriv programmet på en fil med namnet **lab1-2-robust.cc**.

Om man försöker läsa något från en inström som inte kan tolkas som ett värde av samma typ som variabeln man ska läsa in till, kommer inströmmen att försättas i ett feltillstånd och ”frysas”. Det innebär att strömmens läsposition blir kvar på första felaktiga tecken i indata och efterföljande läsförsök kommer att misslyckas på grund av att feltillståndet leder till omedelbar retur från läsoperationen. Feltillstånd kan upptäckas med ström-operationerna bad(), fail() och good(). Om cin.bad() returnerar **true** är det riktigt illa och strömmen kan inte användas mer. I annat fall kan feltillståndet hävas, felaktiga indata läsas bort och därefter ska strömmen kunna användas igen för läsning. Inför sådan felkontroll i programmet.

3. Teckenhantering (stilgranskas)

Skriv programmet på en fil med namnet **lab1-3.cc**. Det finns en webbsida med teckenhanteringsfunktioner länkad från kursens sida för laborationer, "Teckenhanteringsfunktioner":

Programmet ska läsa text från standard inmatningsström `cin` och räkna vilka olika slags tecken som finns i texten. Använd teckenhanteringsfunktioner ur standardbiblioteket (isalpha(), isdigit(), ispunct() och isspace()). Programmet ska skriva ut resultatet i enlighet med följande exempel:

```
Indata innehöll:

 553 alfabetiska tecken
  11 siffertecken
 228 interpunktionstecken
 352 vita tecken
1237 tecken totalt
```

Observera, att det totala antalet tecken inte behöver vara summan av antalet tecken i de fyra kategorierna som ska räknas, och att en text kan utgöras av ett godtyckligt antal rader.

Det finns en uppsättning standardfunktioner för allmän teckenhantering som man får tillgång till genom inkluderingen <cctype>. Ge kommandot ’man -s 3c ctype’ för information om dessa. Då det gäller bokstavshantering fungerar dessa funktioner endast säkert för engelskt alfabet (A-Z). Det är tillåtet att anta att indata endast innehåller bokstäver i det engelska alfabetet.

4. Ordlista

Skriv programmet på en fil med namnet **lab1-4.cc**.

Programmet ska läs ord från standard inström (cin) och lagra orden i en ordlista. Indata kan utgöras av godtyckligt många, godtyckligt långa rader med ord. Läsning ska ske till dess filslut nås i inströmmen.

Ett ord är en godtycklig följd av alfabetiska tecken. Ord separeras i indata med vita tecken (mellanrum, tab-tecken, radslut). Ingen skillnad ska göras mellan stora och små bokstäver; ”ord”, ”ORD” och ”Ord” anses alltså vara samma ord. I ordlistan ska ord lagras med små bokstäver. Du får förutsätta att indata enbart består av alfabetiska tecken och vita tecken, dvs indata innehåller endast ord. De svenska problembarnen å, ä och ö behöver inte hanteras korrekt då det gäller att göra om till små bokstäver eller då det gäller sorteringsord.

Varje unikt ord och hur många gånger ordet förekommit i indata ska lagras. Definiera en **struct** med namnet word_entry, för att lagra ett ord och det antal gånger ordet förekommit.

Ordlistan ska utgöras av en std::vector med en word_entry-struct för varje ord.Orden ska lagras i bokstavsordning i vectorn och varje nytt ord ska direkt sättas in på rätt plats i vectorn. Om ordet inte finns sedan tidigare ska en ny word_entry-struct sättas in med ordet och dess räknare satt till 1, i annat fall ska enbart räknaren i den befintliga struct-en för ordet stegas upp. Utskriften ska vara en tabell enligt följande:

```
 Ord  Antal

 det  1
  en  2
gång  2
 var  1
```

Dela upp i funktioner. Alla data ska överföras via parametrar. Det kan t.ex. vara lämpligt att införa en funktion `insert`(*ordlista*, *ord*), som sätter in *ord* i *ordlista*, och en funktion `print`(*ordlista*) som skriver ut *ordlista* enligt exemplet ovan.

5. Länkad lista

Uppgiften förbereds på lektion 2. Det finns ett givet skelett för ett testprogram på filen **lab1-5.cc**, där det framgår vad testprogrammet (minst) ska göra och vilka utdata som ska erhållas.

De funktioner som ska finnas anges nedan. En av dessa är given i föreläsningsmaterialet och på lektion 2 tas ytterligare någon/några funktioner fram. Det kan finnas fler funktioner än de nedan angivna som är naturliga för en lista eller kan vara bra att ha för att implementera andra funktioner; sådana kan läggas till. Funktioner kan vara *iterativa*, dvs bygga på repetitionssatser (**for**, **while**, **do**), eller *rekursiva*. Implementera funktionerna iterativt, utom i eventuella fall då rekursion medför en uppenbar fördel. Tänk på att funktionerna ska kunna hantera specialfall, till exempel en tom lista. Följande funktioner *ska* finnas:

- **append**() (*given*) ska ta en lista, ett namn och en ålder som argument och sätter in en ny nod med namn och ålder *sist* i listan.
- **insert**() ska ta en lista, ett namn och en ålder som argument och sätta in en ny nod med namn/ålder *först* i listan.
- **clear**() ska ta en lista som argument och radera alla noder i listan; resultatet ska vara en tom lista.
- **empty**() ska ta en lista som argument och returnera **true** om listan är tom, annars **false**.
- **copy**() ska ta en lista som argument, skapa en kopia och returnera en pekare till den första noden i kopian.
- **reverse**() ska ta en lista som argument och vända ordningen på noderna. Det är inte tillåtet att skapa några nya noder och den ursprungliga listan får bara stegas igenom *en* gång i samband med vändningen. Tänk först, koda sedan!
- **print**() (*given*) ska ta en lista och en utström som argument och för varje nod skriva ut namnet, ett mellanrumstecken, åldern inom parentes och sedan ny rad.
- **print_reversed**() ska skriva ut på samma sätt som print() men i omvänd ordning (utan att ändra listan!).
- **swap**() ska ta två listor som argument och byta innehåll på listorna.

Dela upp på filer

Datatyper och funktionsdeklarationer som hör till listan ska skrivas på en inkluderingsfil **List.h** och tillhörande definitioner ska finnas på en fil **List.cc**. Programmet för att testa listan ska finnas på den givna filen **lab1-5.cc**.

Skriv en enkel make-fil för att kompilera programmet. Det finns en kort introduktion till make i kursmaterialet och make gås igenom på lektion 2.

Testdata

En testdatafil 29damer.alfa med 29 namn och tillhörande åldersuppgift finns i mappen för given kod för laboration 5. Gör också egna indatafiler som testar programmet för specialfall, t.ex. tom indatafil.

Frivilliga hemarbetsuppgifter

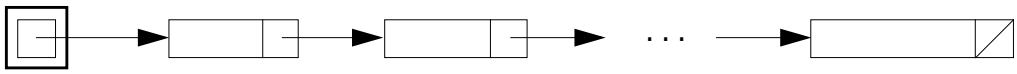
Implementera funktionen reverse() rekursivt.

Följande förutsätter stoff från Fö 3.

Lägg till en rekursiv destruktor och anpassa clear() för detta.

Lägg till en eller flera konstruktorer, eventuellt även NSDMI för någon/några datamedlemmar. Förutsätt att namn och ålder alltid ska anges då en ny listnod skapas med **new**. Anpassa koden där nya listnoder skapas.

För att garantera korrekt initiering, destruering och minneshantering kan en klass List införas. Syftet med en sådan klass är att dölja pekaren till den första noden i en listan för användaren och säkerställa korrekt initiering av en ny lista (att pekaren alltid sätts till **nullptr**) och korrekt destruering av listnoderna när en lista försvinner (destruktorn ska se till att alla listnoder destrueras). I figuren nedan representerar rektangeln som omsluter pekaren till den första noden den kapsling som klassen List ska ge.



Typen List_Node blir genom detta en implementeringstyp för List, dvs inget som en användare av klassen List ska behöva känna till.

Operationerna på listan görs om till medlemsfunktioner i klassen List. Den påtagliga skillnaden är att den lista som operationerna ska operera på är implicit (**this**-objektet).

Eftersom List kommer att vara en icke-trivial klass (vi har en pekare till dynamiskt minne) är det viktigt att ha egna versioner av *kopieringskonstruktor, kopieringstilldelningsoperatorn* och *destruktorn* (de kompilator-genererade duger inte). Även flyttsemantik är intressant för en klass som List, så *flyttkonstruktor* (*move constructor*) och *flyttilldelningsoperator* (*move assignment operator)* bör finnas.

Klassen List kommer att ha påtagliga likheter med klassen String som används som föreläsningsexempel.