Kurs 1DV434

Laborations-PM steg4_3

Ett lexikon.

Förkunskaper som krävs för laborationen

1. För det första måste du klara av att arbeta bekvämt med pekare. Om du har svårigheter med dem, så kommer den här laborationen att bli ett nästan oöverstigligt problem. Klarar man av den här labben så *kan* man hantera pekare, annars kan det vara dags att ta ett varv till med läroboken …
2. Laborationen innehåller filhantering, dvs. att läsa och skriva från en fil. Är du osäker på detta, bör du friska upp dina kunskaper på *fstream*, dvs. hur man öppnar en fil, hur man skriver till den och hur man stänger den.
3. Även stränghantering finns med i laborationen (den ”gamla” sortens *char**). Du bör ta en titt på funktionen *strtok*, som är svår att förstå när man läser om den i bruksanvisningen, men egentligen är suverän att ha tillhands när man skall plocka ord ur en textsträng.
4. Vi ska bygga en någorlunda komplex pekarstruktur för att skapa ett Engelskt-Svenskt lexikon. Själva pekarstrukturen finns beskriven senare. Det är viktigt att du förstår hur den fungerar. Vi har kapslat in pekarna i två olika klasser, *WList* och *TList*. Klassdefinitionerna finner du också nedan i laborationsbeskrivningen. Läs igenom dem och sätt dig in i hur klasserna är tänkta att fungera.
5. Vi kommer att använda *statiska* medlemsfunktioner och medlemsvariabler. Du måste ha insikt om hur sådana fungerar innan du ger dig i kast med laborationen. Därför diskuterar vi nu lite om dessa klassmedlemmar. Om du förstår det som står i avsnittet nedan, så går laborationen mycket smidigare.

Statiska klassmedlemmar

Innan vi påbörjar själva laborationen, ska vi först ta en diskussion om det där med statiska medlemmar i en klass (funktioner och variabler). Ett sätt att tänka sig det hela är att införa ett **klassobjekt**. Vi är ju vana att arbeta med klasser och att skapa objekt från klasserna. Nu betraktar vi klassen själv som om den är ett objekt, det s.k. *klassobjektet*. Varje klass som vi använder har alltså ett samhörande klassobjekt, som också går att använda. Hittills har vi inte utnyttjat detta klassobjekt, utom när vi skapar ett nytt objekt via *new*-anropet. Även om det har varit ”osynligt” för oss, är det nämligen på klassobjektet som vi anropar metoden *new*. Det kan ju inte gärna vara på något vanligt objekt, för något sådant har ju inte hunnit skapas ännu.

Ett klassobjekt kan i princip betraktas som ett vanligt objekt, som har metoder och datamedlemmar precis som vilket annat objekt som helst. Normalt sett innehåller klassobjektet metoder för att skapa vanliga objekt från klassen (dvs. med operator *new*), men vi kan även bestämma andra metoder och datamedlemmar som ska finnas i det. Detta gör man genom att i klassdefinitionen deklarera vissa variabler och funktioner som *static*. Dessa kommer då att hamna på klassobjektet istället för på de vanliga objekt som instansieras från klassen. Det är viktigt att förstå att de då bara finns i en ”upplaga”, dvs. blir gemensamma för alla objekt som skapas från klassen.

Om vi nu tittar på klassen *WList*, vars definition finns längre fram i laborationstexten, så kan du se att vi har använt *static*-deklarationen, för att lägga flera av metoderna uppe på klassobjektet. Detta klassobjekt kommer att tjänstgöra som själva lexikonet. Det är klassobjektet vi vänder oss till när vi vill lägga in nya ord eller när vi vill ta bort ett tidigare sparat ord. Likaså är det klassobjektet som vi ska vända oss till när vi vill ha ett ord översatt.

När vi vill anropa en metod på klassobjektet för klassen *WList* så använder vi instruktionen

WList::metodnamn(parametrar);

Om man inifrån någon av metoderna, i ett objekt som är instansierat från klassen *WList*, vill anropa samma metod som ovan, så kan man även där skriva instruktionen ovan (den fungerar alltid), men man kan även anropa funktionen på samma sätt som när man anropar vanliga medlemsfunktioner på det egna objektet, dvs. enligt följande.

metodnamn(parametrar);

Ett objekt, som är instansierat från en viss klass, har alltså tillgång även till klassobjektets data och medlemsfunktioner, på samma sätt som den har tillgång till sina egna. När vi utifrån vill anropa en metod på ett objekt så använder vi ju, som du vet denna instruktion:

objekt.metodnamn(parametrar);

Här ser vi att C++ gör skillnad på klassobjekt och vanliga objekt, genom att vi måste använda dubbelkolon när vi utifrån sänder ett meddelande till klassobjektet. När vi refererar från ett vanligt objekt, används istället punktnotation. Då utnyttjar vi det faktum att ett objekt, som har instansierats från en viss klass, automatiskt har tillgång till både metoderna och datamedlemmarna hos sitt klassobjekt.

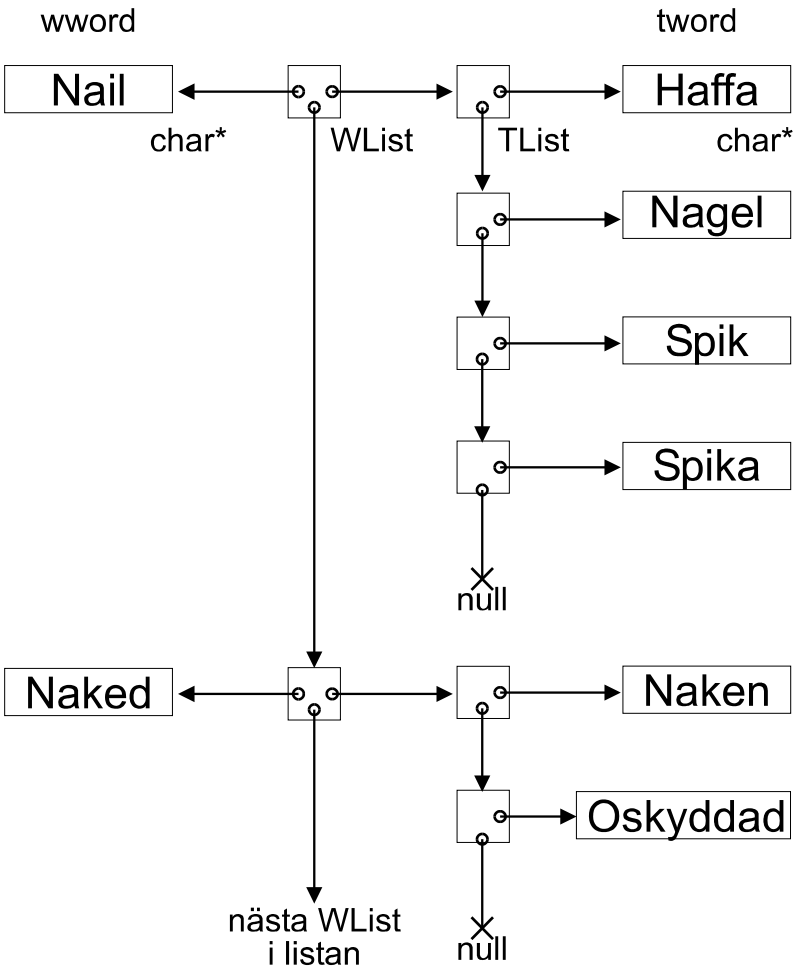
I Java användc punktnotation i båda fallen, så där görs mindre skillnad mellan vanliga objekt och klassobjekt. I Java används också klassobjekten mycket flitigare än i C++, så det är bra att du sätter dig in i konceptet, om du vill lära dig Java-språket senare.

Detta avslutar förberedelseuppgiften.
Nu går vi vidare till själva laborationen!

Kravspecifikation för laborationen

Under laborationen ska du skriva två klasser (*WList* och *TList*) med vars hjälp du ska realisera ett Engelskt-Svenskt lexikon. Med hjälp av detta lexikon ska man utifrån ett engelskt ord kunna finna dess svenska översättning. Eftersom det kan finnas flera översättningsord, så ska vi också kunna få flera svenska ord som svar, när vi ber om översättning av ett engelskt ord.

Det finns ett huvudprogram färdigskrivet, som du ska använda för att testa dina klasser. När du kör detta, ser du på skärmen vad som förväntas av klasserna. Det kan vara bra att i förväg läsa igenom huvudprogrammet och ser hur det fungerar. Nu ska vi dock först titta närmare på de båda klasser som du ska skriva. För att göra sökning och lagring så effektiv som möjligt, så ska vi använda oss av länkade listor enligt nedanstående figur:



Som synes innehåller strukturen ovan ett antal engelska ord (här kallade *wword*), vilka hålls samman i bokstavsordning av en lista (den s.k. ordlistan). Listan innehåller ett antal noder från klassen *WList*. Varje nod har en pekare till själva engelska ordet samt en pekare till den *WList* som representerar efterföljande engelska ord i listan. Vi ser också att varje listnod i ordlistan innehåller en pekare till en *TList*, dvs. en pekare till första noden i den översättningslista som tillhör detta engelska ord. Denna översättningslista innehåller alltså alla de svenska ord som ska tjänstgöra som översättningar till det engelska ordet.

En *TList* innehåller endast två pekare, en som pekar ut det svenska ordet och en som pekar ut den *Tlist*-nod som representerar nästa (i bokstavsordning) svenska ord i listan. Generellt sett brukar man alltid kalla den pekare, som pekar ut efterföljande nod i en lista, för *next*-pekaren. Således har vi i klassdefinitionerna för våra båda listklasser (*WList* och *TList*) använt det här

namnet på denna pekare. När man traverserar (=stegar igenom) en lista baserad på *next*-pekare så måste man hitta den första noden i listan på något sätt, sedan fortsätter man att stega sig igenom listan med hjälp av *next*-pekarna tills man kommer till den sista noden i listan, som inte har någon efterföljare (dvs. *next* = null).

Hur hittar vi då första noderna i de båda listorna? Ja, för varje översättningslista så finns det ju en pekare i den samhörande *WList*-noden som pekar till dess första element, så det är bara att gå via *WList*-noden så hittar man den första *Tlist*-noden i den samhörande översättningslistan. Men hur får man då tag i första *Wlist*-noden i själva ordlistan? Jo, vi lagrar en pekare till den som en statisk datamedlem i *WList*-klassen! Den ligger alltså i klassobjektet för klassen *WList*. Vi har kallat den *whead*, och den är alltså av typen *WList**.

Nu är vi mogna att ge oss i kast med själva klassdefinitionerna. Som du kan se så har vi lagt båda definitionerna i samma headerfil och kallat filen för *List.h*. Du kan göra på samma sätt om du så vill, men du kan även skriva dem i separata headerfiler om det passar dig bättre.

```
#ifndef LIST_H
#define LIST_H

class TList {
friend class WList;
public:
    TList( char* tword, TList* tnext );
    virtual ~TList();
    const char* getWord() const { return word; }
    const TList* successor() const { return next; }
private:
    char*   word;
    TList*  next;
};

class WList {
public:
  //----- instance methods
    WList( char* wword, char* tword, WList* wnext );
    ~WList();
    bool insertTword( char* tword );
    const char* getWord() const { return word; }
  //----- class methods belonging to the class object
    static  WList* insert( char* wword, char* tword );
    static  bool remove( char* wword, char* tword );
    static  void killWlist();
    static  void showWlist();
    static  const TList* translate ( char* wword );
    static  bool save( char* filename );
    static  bool load( char* filename );
private:
    static  WList* whead;
    char*   word;
    TList*  thead;
    WList*  next;
};

#endif
```

Klassdefinitionerna ovan visar författarens design, som inte behöver överensstämma med din. Det gäller dock att de fetstilsmarkerade static-metoderna i klassen *WList* måste se ut exakt som de gör i ovanstående klassdefinition, eftersom de används av det huvudprogram som du ska köra för att testa dina klasser. I övrigt har du full frihet att göra din design som du vill, under förutsättning att den uppfyller följande tre krav:

1. Det ska gå att köra huvudprogrammet i befintligt skick, och de utskrifter som ges från programmet skall överensstämma exakt med det facit som skrivs ut vid körningen.
2. D ska bygga upp den struktur av länkar som visas i figuren ovan, dvs. det skall finnas en ordlista och en översättningslista som båda skall vara sorterade i bokstavsordning på sitt innehåll.
3. De statiska metoderna i *WList* ska ha följande funktion:

   `static WList* insert( char* wword, char* tword );`

   Vid anropet ska *wword* innehålla det engelska ordet och *tword* det svenska. Vid anropet förs dessa båda ord in i bokstavsordning i lexikonet. Om operationen lyckas, så returnerar funktionen en pekare till den *Wlist*-nod som motsvarar det engelska ordet, annars returneras NULL. Om t.ex. den angivna översättningen redan finns registrerad, så förs den inte in igen, utan operationen misslyckas och returnerar då NULL. Att vi returnerar en pekar istället för en *bool* för att tala om hur det gick har sina randiga skäl. Om man direkt vill registrera flera svenska översättningsord för ett visst engelskt ord så kan man anropa *insert* för det första ordet, men sedan anropa *insertTword* på den *WList* som man får i retur, för de övriga svenska orden. Detta blir mycket effektivare och går snabbare eftersom man inte behöver leta rätt på *Wlist*-noden varje gång.

   `static bool remove( char* wword, char* tword );`

   Vid anropet skickas det engelska ordet i *wword* och det svenska i *tword*. Vid anropet tas det svenska ordet bort ur översättningslistan för det engelska ordet. Om översättningslistan därmed blir tom, så tas även det engelska ordet bort ur ordlistan.

   `static void killWlist();`

   Anropet tar bort hela ordlistan och alla översättningslistor. Även de lagrade orden (både engelska och svenska) tas bort. Därmed blir allt dynamiskt allokerat minne återlämnat till systemet.

`static void showWlist();`

Ger en komplett utskrift av innehållet i lexikonet. Det ska se ut så här:

```
Contents of Registry
--------------------
house : hus     koja    logi
pen :   penna
set :   apparat ställa
weird : dum     konstig
```

Efter rubriken skrivs de engelska orden på var sin rad, åtföljda av ett blanktecken och ett kolon. Därefter skrivs på samma rad alla svenska ord som är översättningar för det engelska ordet. Varje sådant svenskt ord inleds med ett tab-tecken.

`static const TList* translate ( char* wword );`

Returnerar en pekare till den första noden i översättningslistan för det engelska ord som skickats som parameter.

`static bool save( char* filename );`

Sparar hela lexikonets innehåll på filen vars namn anges i parametern. Du får själv bestämma hur filens innehåll ska se ut. Huvudsaken är att du senare kan läsa in filen och återskapa lexikonet exakt som det såg ut när det sparades.

`static bool load( char* filename );`

Läser tillbaka det innehåll som tidigare sparats i filen, vars namn ges som parameter. Det innehåll som vid anropet finns i lexikonet raderas först, innan laddningen från fil påbörjas. Efter anropet ska alltså lexikonet se ut på exakt samma sätt som det gjorde när det sparades undan i filen.

När du är klar med dina klasser, provkör du dem med testprogrammet *Lexikontest.cpp* som återfinns i *Filer till lab 3* på kurswebben. Se till att du också får med dig filen *MemoryLeakDetector.h* till din projektkatalog, den inkluderas nämligen av testprogrammet. Denna klass har till uppgift att se till att det görs utskrifter på bildskärmen, om det visat sig att du vid programmets slut inte har gjort *delete* på allt, som du tidigare har gjort *new* på. Det är ju viktigt att vi upptäcker om det finns ”minnesläckor” i programmet!

Vid körningen av testprogrammet ges en serie bilder på konsolen, som ser ut enligt följande:

```
Contents of Registry
--------------------
house : hus     koja    logi
pen :   penna
set :   apparat ställa
weird : dum     konstig
============================================================
===      Texten ovan skall vara samma som texten nedan !   ===
============================================================
Contents of Registry
--------------------
house : hus     koja    logi
pen :   penna
set :   apparat ställa
weird : dum     konstig

Tryck på Retur så går vi vidare !
```

För att få godkänt på laborationen ska det, som skrivs ut ovanför mittbandet, överensstämma exakt med det som skrivs ut under mittbandet (det undre är facit).

Ledtrådar

För dig som vill ha lite ledtrådar, bifogas här de kommentarer som finns till medlemsfunktionerna i vårt eget förslag till implementeringsfilen *List.cpp*. Om du är lite rådvill, så kanske dessa kommentarer kan ge dig den hjälp du behöver. Detta är en ganska komplex laboration, så jag hoppas att du kan hitta några kurskamrater att diskutera laborationen med. Sådana diskussioner är mycket nyttiga! Observera att vi inte gör anspråk på att ha gjort ett perfekt program. Du kanske gör ett som är mycket bättre! Därför bör du ta de här ledtrådarna mer som ett diskussionsunderlag än ett facit!

```
/*********************************************************************
 *                      TList constructor
 *                      -----------------
 * This constructor will link itself into the tlist before the node
 * pointed to by the second parameter. It makes a copy of the word
 * given in first parameter, and sets its word pointer to point to it.
 * The predecessor in the tlist will not be updated by this code !!!
 *
 *********************************************************************/

/*********************************************************************
 *                      TList destructor
 *                      ----------------
 * This destructor will just delete the word it owns.
 *
 *********************************************************************/

/*********************************************************************
 *                      WList constructor
 *                      -----------------
 * This constructor will create a new tlist containing a single node
 * and set its thead pointer to point to it. The TList node is sent
 * the tword parameter as parameter to its constructor.
 * It creates a copy of the wword parameter and sets it word pointer
 * to it. Finally it sets its next pointer to the same value as the
 * last parameter.
 *
 *********************************************************************/

/*********************************************************************
 *                      WList destructor
 *                      ----------------
 * This destructor will delete the word it owns and also kill its
 * entire tlist and the words connected to that tlist.
 *
 *********************************************************************/

/*********************************************************************
 *                      bool WList::insertTword( char* tword )
 *                      --------------------------------------
 * This method will insert a translation word to an already existing
 * Wlist node, which means that the node's tlist will be extended by
 * another TList element pointing to the word specified in parameter.
 * Method will return true if a new word has been inserted, otherwise
 * false. A false return normally means that the same word already has
 * been registered, and that no duplicate was inserted.
 *
 *********************************************************************/
```

```
/*****************************************************************
 *         static WList* WList::insert( char* wword, char* tword )
 *         -------------------------------------------------------
 * If wword is not found in wlist, this class method will insert a new
 * WList node at the proper position in the wlist. It will also create
 * a tlist entry for tword. If wword is already in wlist, its tlist
 * will be extended with a new Tlist node containing tword.
 * Method will return a pointer to the affected WList if successful,
 * otherwise it will return NULL meaning that the tword already has
 * been registered with this wword or that some other error occurred.
 *
 *****************************************************************/

/*****************************************************************
 *         static bool WList::remove( char* wword, char* tword )
 *         -----------------------------------------------------
 * This class method will find the wlist node for wword, and then
 * traverse its tlist until tword is found. It will then remove that
 * TList node. If this operation makes the tlist empty, the method
 * will also remove the WList node.
 * Method will return true if operation was successful, otherwise false.
 * It will normally return false only if tword is not found for wword.
 *
 *****************************************************************/

/*****************************************************************
 *                      static void WList::showWlist()
 *                      ------------------------------
 * This class method will produce a printout of the entire registry.
 *
 *****************************************************************/

/*****************************************************************
 *             static TList* WList::translate( char* wword)
 *             --------------------------------------------
 * This class method will return the tlist for the WList node
 * corresponding to parameter wword.
 *
 *****************************************************************/

/*****************************************************************
 *                      static void WList::killWlist()
 *                      ------------------------------
 * This class method will delete all links and words stored in dictionary.
 *
 *****************************************************************/

/*****************************************************************
 *             static void WList::save( char* filename)
 *             ----------------------------------------
 * This class method will save the entire contents of the registry to
 * the file specified by parameter. The format agrees with the printout
 * produced by ShowList (except that no headers are written now).
 *
 *****************************************************************/

/*****************************************************************
 *             static void WList::load( char* filename)
 *             ----------------------------------------
 * This method will first erase the current content of the dictionary
 * and then load new content from the file specified in the parameter.
 * The file must previously have been written by a call to save method.
 *
 *****************************************************************/
```