A first step toward an Object Oriented Design is to identify application concepts or entities that may be useful to represent by programmer defined data types.

For kwic333, some of these are alphabetically ordered list, line, word appearance, listing, (key) word.

A word appearance has both a line (from the input) and a position within that line at which a word (as defined in the specification) appears. One word and one line may belong to many different word appearances.
Some operations concerning word appearances:

1. Build a partially initialized word appearance from a line.
2. Find one or more places to complete a word appearance.
3. Store word appearances in a set, ready for sorting.
4. Tell if one word is alphabetically $\leq$ another. (Separate concern from finding the word place).
5. Sort a set of word appearance to construct a sorted sequence of word appearances
6. Print a word appearance with the specified indentation and truncation rules to make the key word begin at col. 40.

Data created by strread:

One line:

```
word ;
```

One word appearance:

```
pch
pline
```

The word appearance creation problem:

Given one line plus either the "j" value from a previous word appearance (with that line) or j=0, FIND the next word at or after j if there is any.
Algorithm: Finite State Automaton (or scanner).
Inputs:
- pch
- initial position (0 or previous j)

Variables:
- const char * pline = pch;
- int c = initial position //locates NEXT char to process
- enum state_t state = Ready;
- int i, j; // to report position of found word.

Reference: Enumeration type (Strou. sec 4.8)
Initialize: current char. position;
and CURRENT STATE=“Start State”

Finite state scanner loop:

1. Retrieve and classify the current character.

2. Depending on the CURRENT STATE and CURRENT CHAR. CLASS:
   
   (a) Possibly take an action.
   
   (b) Choose a new state (may be the same as current).

3. Exit if new state is “DONE”; otherwise make CURRENT STATE be the new state and make the current character be the next.

#include "classify.h"
enum state_t { Ready, Not, Inside, DONE };
state_t state; // current STATE
int c; // position of NEXT char to analyze.
chclass_t chclass; // class of NEXT char
const char *pline = ...; // line to scan
int i; // i will record the index where the
     // word being found starts,
     // i=-1 to report none found.
int j; // j will record the index just
     // after the word’s end.

// Code in a loop that prepares for scanning
i = -1;
state = Ready;
c = ...; //either 0 or previous value of j
// classify.h Classifier for kwic333
enum chclass_t { Space, Letter, Other, Null }; external chclass_t classify( char ch );

// classify.cpp Classifier for kwic333
#include <ctype.h>
#include "classify.h"
chclass_t classify( char ch )
{
    if ( ch == ' ' ) return Space;
    if( isalpha( ch ) ) return Letter;
    if( ch == '\' ) return Null;
    return Other;
}
while (state != DONE) {
    chclass = classify( pline[c] );
    switch( state ) {
        case Ready:
            switch( chclass ) {
                case Space:
                    c++; state = Ready; break;
                case Other:
                    c++; state = Not; break;
                case Letter:
                    i = c;
                    c++; state = Inside; break;
                case Null:
                    // You figure out!!
            }
            break; // end of state==Ready case

        case Inside:  //in switch(state) block
case Inside: // in switch(state) block
switch( chclass ) {
    case Space:
        // You figure out!!
        // more chclass cases
    }
    break; // end of state==Inside case

    case Not:   // in switch(state) block
        // Handle state==Not chclass cases.
        // More state cases.
    } // end of switch( state )
} // end of while
if( i < 0 )
{
    // finish with this line
}
// Create a new word appearance
// Continue a loop to seek another.
Sequences of **word appearances** will be sorted.  

**Data Structure Alternatives:**

1. Linked list.

2. Array of either **word appearances** or pointers to them.

Linked list disadvantages: Space needed for links (pointers), less convenient for sorting.  
Array disadvantage: Number of word appearances is unpredictable.  
Solution: **Extendable Array**

```
struct ArrayWA {          // Extendable Array
    WA *pWA;          // Pointer to actual Word Appearance array.
    int nused;        // Number of entries used, which is
    // the next available index if nused < length
    int length;       // Number of entries available at *pWA
} myArray;
```

Whenever a new entry is needed, check if nused==length If so,

```
WA *pnewWA = new WA[ newlen = 2 * myArray.length ];
for( int i = 0; i < myArray.nused; i++ )
    pnewWA[i] = myArray.pWA[i];  //Copies entire structure!
delete myArray.pWA;     //Copy first, before deleting!
myArray.pWA = pnewWA;
myArray.length = newlen;
```

It is a good idea to **double** the array size whenever it needs extension.  
Initialize everything: Put initial array in free-store.
Remember:
myArray.pWA contains the pointer to (i.e., address of) the actual array
myArray.pWA + i is address of element i
*(myArray.pWA + i) refers to element i.
myArray.pWA[i] also refers to element i.
myArray.pWA[myArray.nused] is the NEXT available element, provided
myArray.nused<myArray.length.

Why it’s cool to multiply the array length by 2 when extending it:
If the length is multiplied by 2 whenever the array fills up, the total of all allocations is ALWAYS LESS THAN 4 TIMES the amount actually needed (if the initial allocation is used.)

Proof: Let InitLen = the initial length; NeededLen = the length needed. The total number of array elements allocated is

\[ InitLen + 2InitLen + 2^2InitLen + \ldots + 2^NInitLen \]

\[ 2^{(N-1)}InitLen < NeededLen \leq 2^NInitLen \]

\[ Total = (2^{(N+1)} - 1)InitLen < 4 \cdot 2^{N-1}InitLen < 4 \cdot NeededLen \]
More facts, proved similarly:

Number of copyings of elements is LESS THAN $2 \cdot NeededLen$.

Number allocations not freed is LESS THAN $2 \cdot NeededLen$.

Proofs use the formula for the sum of a geometric series:

$$A + RA + R^2A + \cdots + R^NA = \frac{R^{(N+1)} - 1}{R - 1}A$$

with $R = 2$.

One can choose $R$ with $1.0 < R < 2$ if space is expensive, to use less space but more time for allocation and copying. This is a **Space-Time Tradeoff**