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 333cxref, some of these are identifier, identifier appearance, line, position (in a line), line number, logical line and identifier report.

A id. appearance has both a line (from the input) and a position within that line at which an identifier (as defined in the specification) appears. One identifier and one line may belong to many different id. appearances!

Each id. report (what’s printed) has ONE identifier and the (line number:position) pairs for all identifier appearances with that identifier.
Some operations concerning **logical lines**:

1. Scan it to find a substring satisfying the lexical definition for an identifier. Then report the **line number** and the **position**. This operation must be repeatable to find all of them.

2. Get the next **logical line**, or tell that there are no more.

3. Obtain the next character (for use by the scanner) together with the **line number** and **position**.

---

Some operations concerning **identifiers**:

1. Given a string, construct an **identifier** object to store it, or tell that it is already stored. (We don’t want to store multiple copies of strings!)

2. Sort a set of **identifiers**. Sorting uses:

3. Compare two **identifiers** to tell which should be put before the other, according to the sorting rule for **THIS PROJECT**.

4. Add an **id. appearance** to the list of **id. appearances** of this **identifier**.

5. Retrieve the list of this **identifier**’s **appearances** (so they can be printed).
Some operations concerning **id. appearances**:

1. Construct an **identifier appearance**, from the **identifier**, **line number** and **position** found by the scanner.

2. Print the **line number** and **position**.

---

Sequences of **identifiers** or **identifier appearances** will be sorted.

**Data Structure Alternatives:**

1. Linked list.

2. Balanced search tree.

3. Array of them pointers to them.

Linked list disadvantages: Space needed for links (pointers), less convenient for sorting.

Array disadvantage: Number of items is unpredictable.

Solution: **Extendable Array**
Data Structure for `identifier`: index into a **String Table**

### String Table

```cpp
class StringTable {
    char *pstr;
    int tableSize;
    int nextFreeIndex;
    void grow(void);
public:
    StringTable(int initSize = 128);
    int insert( const char *);
    const char *p(int i){ return pstr+i; }
}
```

### StringTable::StringTable( int initSize )

```cpp
{ pstr = new char[initSize];
  tableSize = initSize;
  nextFreeIndex = 0;
}
int StringTable::insert( const char *pcc )
{
    int ret = nextFreeIndex;
    int strSize = strlen(pcc) + 1;
    while( nextFreeIndex + strSize > tableSize )
        grow();
    strcpy( pstr+nextFreeIndex, pcc );
    nextFreeIndex = nextFreeIndex + strSize;
    return ret;
}
```
Remember (within class StringTable scope):

- `pstr` contains the pointer to (i.e., address of) the actual array
- `pstr + i` is address of element `i`
- `*(pstr + i)` refers to element `i`
- `pstr[i]` also refers to element `i`
- `pstr[nextFreeIndex]` is the NEXT available element, provided `nextFreeIndex < tableSize`.

```cpp
void StringTable::grow(void)
{
    // Allocate a new char array of 2*tableSize.
    // Copy old chars from pstr[...] into new array.
    // Update pstr and tableSize.
    // Free old char array.
}
```
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

$$InitLen + 2InitLen + 2^2InitLen + \cdots + 2^NInitLen$$

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

$$Total = (2^{(N+1)} - 1)InitLen < 4 \cdot 2^{N-1}InitLen < 4 \cdot NeededLen$$

---

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:

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

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**.
The extendable array might be used in Project 4 for:

1. The string table (array of characters).

2. Array of identifiers:
   Each identifier has a string table index and list of identifier appearances

3. Array of identifier appearances: Each entry has (at least) the line number and position.

You also have to design data structure for rapidly:

1. Given a string, construct an identifier object to store it, or tell that it is already stored. (We don’t want to store multiple copies of strings!)

2. Sort a set of identifiers.

Alternatives:

1. Hash table for identifiers, array of identifiers, sort the identifier array after scanning all of input.

2. Balanced tree for identifiers, in-order tree traversal to print output.

Each identifier must refer to its list of appearances.
A very different alternative:

1. Input processing generates the set of identifier appearances.
2. Sort that set lexicographically, first on identifier, second line number, third position.
3. Print sorted set.