

## Exercise Word Statistics (`wordstats.c`)

Implement a tool for word statistics. For this, your program should allow the user to enter an arbitrary number of words with up to 63 characters. For each word it should be checked whether it already existed and if not, it should be stored. After “.” was entered, the program should print the number of distinct words that were entered and then terminate.

### Example

#### Input

```
banana
Apple
apple
banana
```

#### Output

The words `Apple` and `apple` are treated as different words, therefore there are 3 different words.

3

### Hints/Remarks

Make sure to also read the next exercise. It is similar in certain ways and you can save a lot of work if you reuse parts of this exercise.

#### Useful procedures

Use the procedures `scanf` for the input and `strcmp` to check whether two strings are identical.

You should implement more procedures yourself, otherwise your program might become unreadable (an efficient implementation should take around 50+ lines).

It is a good idea to store strings on the heap. `string.h` contains useful procedures for this, for instance `strdup`.

You will have to store words that are read in some datastructure (some kind of table). A possible way to handle this is to store the input in some local variable (of type `char[64]`). If this buffer contains a “.” (`strcmp`), you should terminate your program. Then, check whether the content of this local variable already exists inside the table of words. If it does not duplicate the local string variable (`strdup`) and store it in your table. Since we do not know the size of this table, it should also be stored on the heap and resized using `realloc` if necessary.

If you follow this strategy, the table is a pointer to multiple strings. Since the datatype for strings is also a pointer (to characters), the final datatype for this table would be `char **` (a pointer of pointers to characters).

#### Using `realloc`

The program must read an unlimited number of words. The simplest solution to solve this problem is to allocate a data structure on the heap and if it is too small use `realloc` to reallocate more memory.

In order to avoid too many reallocations it is a common strategy to always double the size of the data structure (this strategy is used in ArrayList in Java). See the following (incomplete) code snippet to see how this strategy works:

```
int size = 16; // elements reserved in dyn-array

... *dynarray = malloc(sizeof(...) * size);

len = 0; // actual number of elements used.

while(read more elements) {
    // in here we will write something into dynarray[len].

    if(len >= size) {
        // It seems that dynarray is too small
        size = 2 * size; // double its size
        dynarray = realloc(dynarray, sizeof(...) * size);
    }

    dynarray[len] = something;
    len++;
}
```

If you already have experience with linked lists, heaps or binary sorting trees you may of course also use these more complex data structures.

If you do not manage to implement a dynamic array but rather use one with static size, you will receive 50% of points, provided the rest of your program is correct.