1. **strncpy** is a rather dangerous function, since it may result in buffer overflow or an unterminated string. An implementation that would work similarly to the standard is shown below. For reference, here’s the description:
The `strncpy()` function copies no more than \( n \) bytes from the second string argument (\( s_{\text{from}} \)) to the first (\( s_{\text{to}} \)).
If there is no null character within these bytes, then \( s_{\text{to}} \) will not be null-terminated.
If \( s_{\text{from}} \) is shorter than \( n \), the remainder is padded with null characters.
If \( s_{\text{from}} \) and \( s_{\text{to}} \) overlap (e.g. `strncpy(str, str+2, 3)`), the function’s behavior is undefined.
If \( s_{\text{to}} \) is not large enough to receive the \( n \) characters, the function’s behavior is undefined.
There is no error checking for these two situations.
The return value is \( s_{\text{to}} \). There is no special return value (e.g. NULL) to indicate there was an error.

```c
unsigned int strlen(const char *str) {
    int i;
    for (i=0; str[i]!='$\null$'; i++) ;
    return i;
}

char *strncpy( char *s_to, const char *s_from, unsigned int count ) {
    int i, j;
    for (i=strlen(s_to), j=0; s_from[j]!='$\null$' && j<count; i++, j++) {
        s_to[i] = s_from[j];
    }
    while (j<count) {
        s_to[i++] = '$\null$';
        j++;
    }
    return s_to;
}
```

2. The original text is stored in a string which is very inflexible when it comes to modifying it. There are more practical implementations, such as linked lists of small arrays, or arrays with gaps or tables with character positioning information.

Now back to the current problem. If the replacement is shorter than the word, we will have no memory allocation problems. If it is longer, then we will need to allocate more space for the text. One solution would be to count all occurrences of the word in the text (through repeated calls to `strstr`), evaluate how much space we need, reallocate the text, then find every occurrence of the word again and replace it, making sure to make room for it by shifting elements over. Such a solution would take up a lot of time, since it would traverse the array several times (the shifting is particularly expensive).

If we don’t have any memory limitations, a faster solution would involve allocating enough memory for a new string to contain our text, then traverse the original copying it over to the new string, replacing the words as we go. In the end, discard the old text and use the new one.

Some comments regarding the replacement: `strstr` returns a pointer to the first occurrence of a substring. If you substract the beginning of the sting from that, you’ll get the distance between them
and you can use strncpy to copy that chunk. Then, use strcat to add the replacement, and repeat the process starting from the next character in the original text.