Exercise 9 – Dynamic Memory

Objective

This practical session is designed to consolidate your understanding of dynamic memory allocation and to illustrate a number of useful techniques that are often used when dealing with dynamic data structures.

Reference Material

This session is based on material in the Dynamic Memory Management chapter. This practical session is located in the following directory:

	Microsoft Windows	*Linux*
Directory:	c:\qacadv\dynmem	**~/qacadv/dynmem**
Solution directory:	c:\qacadv\dynmem\Solution	**~/qacadv/dynmem/Solution**

The debugging routines discussed in the chapter are available in the sub-directory **memdebug**. You may wish to use these if your routines don't seem to be working properly, or just to provide "peace of mind".

Overview

In this practical session you will write the core routines of an editor. First you will write a program that reads a single line of arbitrary length into dynamic memory (dynamic memory must be used since it cannot be known in advance how long the line will be). The next step is to write a program that reads an arbitrary number of lines into dynamic memory, each line of arbitrary length.

There are a number of text files provided for you to use. **short.txt** contains one line of about 240 characters, **long.txt** one line of about 10000 characters and **vlong.txt** one line of about 20000 characters.

On Microsoft Windows DO NOT read these last two files into Visual Studio because it can only cope with lines up to about 250 characters.

Practical Outline

1. **On Microsoft Windows** open **readline.sln, on Linux** change your current working directory.

   Take a look at **readline.c**. The program opens a file and calls the function `process()`. This returns a pointer, which at the moment is not used (since it is `NULL`). The idea is to read a single line from the file into a dynamically allocated array. Once this has been done the address of this array should be returned. Don't

forget to append the '\0' terminator onto the end of the array, otherwise `printf` will do horrible things.

For the first version of the function, read each character from the file and increment a counter, say `c`. Call `realloc()` to reallocate a block of `c` bytes of memory. Copy the character onto the end of this reallocated block (i.e. the position denoted by `c`). Keep on going until the newline character '\n' or EOF is reached.

For example: say the file contains "abc\n". You would read 'a' and increment the counter from 0 to 1. Call `realloc()` to enlarge a block of 0 bytes to 1 byte. Copy 'a' into the storage at position 1 - 1, i.e. 0. Next read 'b'. Increment the counter from 1 to 2. Call `realloc()` to enlarge the block of 1 byte to 2 bytes. Copy 'b' into the storage at position 2 - 1, ie 1. Then read 'c'. Increment the counter to 3, call `realloc()` to enlarge the block to 3 bytes. Copy 'c' into position 3 - 1, ie 2. Read the '\n'. Call `realloc()` to enlarge the block to 4 bytes, copy a '\0' into position 4. Return the address returned by `realloc()`.

A sample solution is available in the **Solution** sub-directory, called **readln1.c**.

2. Working in the same directory or workspace, consider the performance and efficiency of the above approach. The problem is that `realloc()` is called for every character in the file. If the file is large, `realloc()` will be called many thousands of times.

   Once your `process()` function is working, update it to maintain a second counter, say `allocated`, which counts the number of bytes actually allocated. It must always be larger than the counter `c` so there is always spare room in the array. For example, imagine the input line reads "hello world" - 11 characters. At the start `c` is 0 and `allocated` is 10. Call `realloc()` to allocate a block of 10 characters. Read 'h', increment `c`, since it (1) is less than `allocated` (10), no need to call `realloc()`. Store the character as before and continue. Read the 'e', the counter `c` is now 2 which is still less than `allocated`, again no need to call `realloc()`. Store the character and continue.

   This carries on through the characters "llo worl". By the time the character 'l' is reached the value of `c` has become equal to `allocated`. Thus `allocated` is increased by 10 to 20 and `realloc()` is called to allocate a block of 20 bytes. The 'd' and '\0' characters may be stored without further ado.

   Admittedly using this strategy the storage allocated is larger than that required, however you should notice the program running much faster than before. You may wish to spend a little time increasing the value by which `allocated` is incremented. The greater the amount the fewer times realloc is called, however, if it is made too large ridiculously large blocks will be allocated.

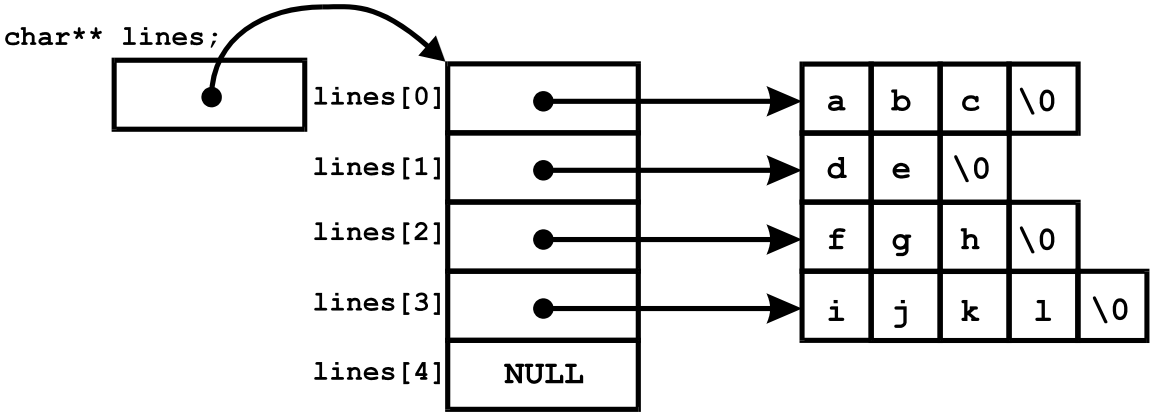
   You may want to incorporate the timer routine to check the timings.

A sample solution is available in the **Solution** sub-directory, called **readln2.c.**

3. **On Microsoft Windows** open **readfile.sln, on Linux** stay in the same working directory.

   Take a look at **readfile.c**. The function `process()` now reads in a whole file by calling the function `processLine()` until EOF causes it to return NULL.

   You will need to incorporate your `process()` function from the last exercise and rename it `processLine()`. Use the variable `lines` (declared as a pointer to a pointer) as follows:



This is the arrangement of the pointers if the file contains 4 lines. The blocks of memory containing “abc”, “de”, “fgh” etc. are those that will be allocated by your `processLine()` routine as it stands at the moment. This routine will need to be modified slightly in that when EOF is hit it must return NULL. Thus `process()`, as it is now called, will sit in a loop:

```
while((p = processLine(....)) != NULL)
      /* store line pointed to by p */
```

The NULL value needs to be placed at the end of the array so that `main()` knows it has reached the last line.

The algorithm for allocating the array of lines is much the same as that for allocating the array of characters.

If you need an input file to test, try the **readfile.c** file itself. The program opens files for reading and so no harm should come to your code.

A sample solution is available in the Solution sub-directory, called **readfile.c**.