Computation I 5EIA0
C Exercises for Week 5
(v0.3, August 2, 2022)
Solutions

1 Introduction

The solutions are surrounded by the following standard code.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
int main (void) {
  // code is inserted here
}
```

Note that the addresses that your code prints are most likely different from the ones shown in the example outputs. That is because compilers on different computers may place variables at different addresses in memory. This is why you should never use absolute addresses.

2 Malloc & Free Pitfalls

1. Write a program that declares variable p with type pointer to an integer. Do not initialise it. Print out its value (as a pointer). Then set initialise it to NULL and print its value. Then malloc 10 integers and assign the result to p; print its value. Then free the malloc'd space and print the value of p. The addresses that you see in your output are most likely different. Can you explain:

   - What the first value of p is?
   - What is the value 0x0?
   - Why are the last two values of p the same? Is this a potential pitfall?

   This is the output on my Mac:

   ```
   p=0x108910025
   p=0x0
   p=0x7f8b11405950
   p=0x7f8b11405950
   ```

   Hint: use the "%p" `printf` format string to print out a pointer.

   This is the output in the VM, where the unitialised value happens to be NULL:

   ```
   computation@computation-virtual-machine:~ gcc -Wall x.c && ./a.out
   x.c: In function 'main':
   x.c:6:3: warning: 'p' is used uninitialized in this function [-Wuninitialized]
      printf("p=%p\n",p);
      ^~~~~~~~~~~~~~~~~~
   p=(nil)
   p=(nil)
   p=0x55dd7abf3670
   p=0x55dd7abf3670
   ```

```
int *p;
printf("p=%p\n",p);
p = NULL;
printf("p=%p\n",p);
p = (int *) malloc (10*sizeof(int));
printf("p=%p\n",p);
free(p);
printf("p=%p\n",p);
```

- The first value of p is a random value since p has not been initialised. Any attempt to use it will likely result in a Segmentation Fault, i.e. an invalid memory access.
- The value 0x0 is zero in hexadecimal. It is equal to the NULL pointer which is just the value zero.
- The last two values of p are the same because the function free does not change its argument. Therefore, the pointer p still points to the same memory area *even after it has been freed*. This is a common pitfall, because you can still access this memory region even though you shouldn't. (See another question below.)

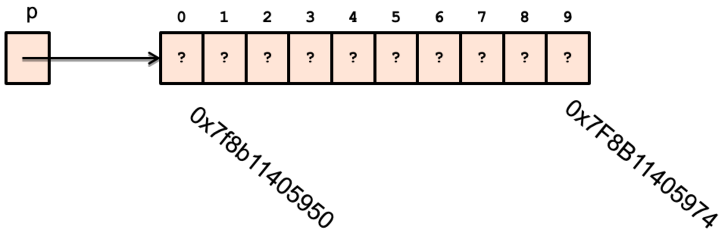

Figure 1: One pointer to a memory area.

2. Print the value at the address in p (the integer than p points to) *before it is initialised*. What happens? On my Mac it happens to not crash and print a random value:

```
p=0x1125e0025
*p=-125990072
p=0x0
p=0x7ff4d5405950
p=0x7ff4d5405950
```

while in the VM it crashes:

```
p=(nil)
Segmentation fault (core dumped)
```

Thus this program is wrong because it uses a variable before it has been initialised, thus uses an undefined value.

3. Now let's investigate what happens when we use space after it has been freed. Declare a new variable q that is also a pointer to an integer. After freeing p and printing its value, malloc 10 integers and assign the result to q. Print out the value of q. Can you explain this value?

```
p=0x7fff2c614250
p=(nil)
p=0x55a5dd864670
p=0x55a5dd864670
q=0x55a5dd864670
```

Most often what you will see is that the value of q is the same as the value of p. The reason is that what p points to is free space. This free space is big enough for the next malloc, the result of which we assign to q. In other words, q's malloc reuses p's space that was freed. Why is this a potential problem?

```
int *p, *q;
printf("p=%p\n",p);
p = NULL;
printf("p=%p\n",p);
p = (int *) malloc (10*sizeof(int));
printf("p=%p\n",p);
free(p);
printf("p=%p\n",p);
q = (int *) malloc (10*sizeof(int));
printf("q=%p\n",q);
```

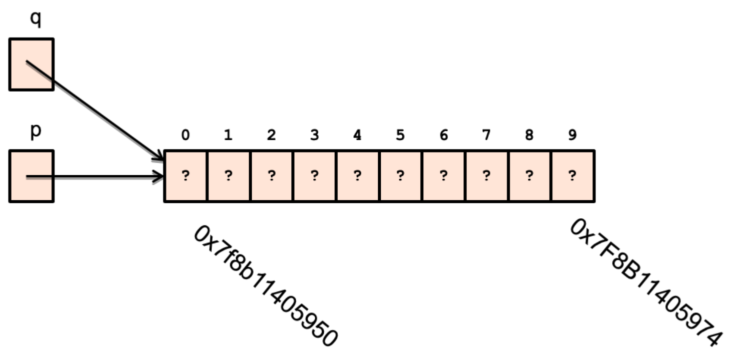

Figure 2: Two pointers to the same memory area.

4. The problem of the previous question is called "aliasing" because there are two variables pointing to the same memory location. This means that changing the memory via one pointer variable will also change the content that the other one points to. Sometimes this is what you want. But most of the time it is not and it can be very confusing. Let's investigate the problem in a few steps. First recall that p points to 10 integers. In other words, it is really an array of 10 integers. Before freeing p set the 10 array elements to the values 10, 20, ..., 100 and print them out.

```
p=0x7fffbf164810
p=(nil)
p=0x55a68d5db670
p[0]=10
p[1]=20
...
p[9]=100
p=0x55a68d5db670
q=0x55a68d5db670
```

```
int *p, *q;
printf("p=%p\n",p);
//printf("*p=%d\n",*p);
p = NULL;
printf("p=%p\n",p);
p = (int*) malloc (10*sizeof(int));
printf("p=%p\n",p);
for (int i=0; i < 10; i++) p[i] = 10*(i+1);
for (int i=0; i < 10; i++) printf("p[%d]=%d\n",i,p[i]);
free(p);
printf("p=%p\n",p);
q = (int*) malloc (10*sizeof(int));
printf("q=%p\n",q);
```

5. Next, after freeing p set the 10 array elements of q to the values -10, -20, ..., -100 and print them out. All should be as expected. Now also print out the 10 array elements of p. Explain what happened.

```
p=0x7fffbf164810
p=(nil)
p=0x55a68d5db670
p[0]=10
p[1]=20
...
p[9]=100
p=0x55a68d5db670
q=0x55a68d5db670
q[0]=-10
q[1]=-20
...
q[9]=-100
... what happens here? ...
```

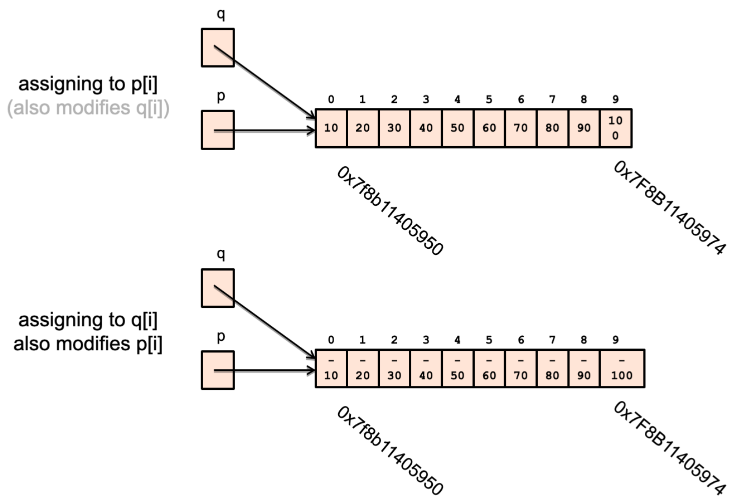

Figure 3: Aliasing: modifying the memory pointed to by p through another pointer q.

What you will (most likely) see is that the values of the array p have also changed. The reason is that p and q point the same memory region and thus to the same data. This data can be changed through both p and q. To avoid that this happens accidentally it is recommended to set pointers to NULL after freeing them. If then accidentally use it then you will get a Segmentation Fault, which is an immediate failure. In other words you know immediately that your program is wrong (and 'where' in the gdb debugger will tell you immediately where the problem is). That is much better than a program that starts to act strangely and is hard to debug.

If you cannot reproduce the reuse of the freed memory region then just replace the `q = (int*) malloc ...;` line by `q = p;` which achieves the same thing.

```
p=0x7fffbf164810
p=(nil)
p=0x55a68d5db670
p[0]=10
p[1]=20
...
p[9]=100
p=0x55a68d5db670
q=0x55a68d5db670
q[0]=-10
q[1]=-20
...
q[9]=-100
p[0]=-10
p[1]=-20
...
p[9]=-100
```

```
int *p, *q;
printf("p=%p\n",p);
//printf("*p=%d\n",*p);
p = NULL;
printf("p=%p\n",p);
p = (int*) malloc (10*sizeof(int));
printf("p=%p\n",p);
for (int i=0; i < 10; i++) p[i] = 10*(i+1);
for (int i=0; i < 10; i++) printf("p[%d]=%d\n",i,p[i]);
free(p);
printf("p=%p\n",p);
q = (int*) malloc (10*sizeof(int));
printf("q=%p\n",q);
for (int i=0; i < 10; i++) q[i] = -10*(i+1);
for (int i=0; i < 10; i++) printf("q[%d]=%d\n",i,q[i]);
for (int i=0; i < 10; i++) printf("p[%d]=%d\n",i,p[i]);
```

6. Sometimes we accidentally free the same memory region twice, which is not allowed. Try it! Free p twice. What happens?
   In the VM this is nicely reported before the program crashes.

```
p=0x7ffcdbc9a480
p=(nil)
p=0x55ab84257670
p[0]=10
p[1]=20
...
p[9]=100
free(): double free detected in tcache 2
Aborted (core dumped)
```

   On my Mac it gives a similar error message:

```
p=0x1069f1025
p=0x0
p=0x7fce27405950
p[0]=10
p[1]=20
...
p[9]=100
a.out(20805,0x106ac7e00) malloc: *** error for object 0x7fce27405950: pointer bei
a.out(20805,0x106ac7e00) malloc: *** set a breakpoint in malloc_error_break to de
Abort trap: 6
```

7. Let's investigate memory leaks. A memory leak occurs when we malloc some memory but lose access to it. As a result we cannot free it anymore. Can you write a (very small!) program to do this?

```
int *p = NULL;
printf("p=%p\n",p);
p = (int*) malloc (10*sizeof(int));
// this will do it:
p = (int*) malloc (10*sizeof(int));
// but this is even simpler:
p = NULL;
// in both cases we cannot free the original memory region that p pointed to any more
```

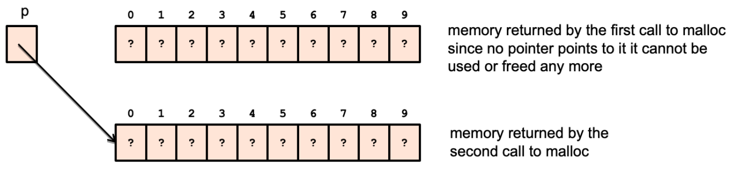
p
0 1 2 3 4 5 6 7 8 9
? ? ? ? ? ? ? ? ? ?
memory returned by the first call to malloc since no pointer points to it it cannot be used or freed any more
0 1 2 3 4 5 6 7 8 9
? ? ? ? ? ? ? ? ? ?
memory returned by the second call to malloc

Figure 4: A memory leak

3 Using minigrind to avoid memory leaks and multiple frees

Minigrind is a mini version of Valgrind, a program that detects memory problems (leaks, freeing multiple times, not freeing, etc.) However, Valgrind is not so easy to use, which is why I wrote minigrind as a simpler-to-use alternative. (Of course it also does a lot less than Valgrind.) Minigrind is available on Oncourse (it is planned to also have it during the exams), and it may be useful to know how to use it. The correctness of your memory management in homework and exams will be checked by using the `minigrind` library (after the homework or exam has closed).
To keep track of the space you've malloced and freed you can include `#include "minigrind.h"`. It replaces all `malloc` and `free` function calls by versions that keep track of whether malloc'd space has been freed or not. To help you debug, calling the `CheckMemory` function prints all space that you have malloced but not (yet) freed. You can just comment out this line when you don't want to use `minigrind`.
It is instructive to redo the previous exercises with minigrind included. In particular questions refleak and 6 shown below are good to look at.

1. Add `include "minigrind.h"` at the top of your memory-leak program of Question 7. You should see this output:

```
Malloc 40 bytes, entry 0
Malloc 40 bytes, entry 1
FinalCheckMemory: entry 0, 40 bytes not freed
FinalCheckMemory: entry 1, 40 bytes not freed
```

   The FinalCheckMemory is automatically run at the end of the program. Notice that it reports *two* leaks. Why is that?

   Can you make it produce the following output, where there is no memory leak?

```
Malloc 40 bytes, entry 0
Free 40 bytes, entry 0
Malloc 40 bytes, entry 1
Free 40 bytes, entry 1
```

   Hint: The simple memory leak program did two mallocs and did not free both of them.

2. Add `include "minigrind.h"` at the top of your double-free program of Question 6. You should see this output:

```
p=0x7ffcdbc9a480
p=(nil)
Malloc 40 bytes, entry 0
p=0x55ab84257670
p[0]=10
p[1]=20
...
p[9]=100
Free 40 bytes, entry 0
Free: entry 0, 0 bytes, address 0x562515d18670
Free: error: re-freeing entry 0, 0 bytes, address 0x562515d18670
```

4 Two-Dimensional Arrays

1. Declare a a two-dimensional array of integers called a of 3 rows and 4 columns. Using loops, set each element row, column to the value row+column. Then print all elements.

```
a[0][0]=0
a[0][1]=1
a[0][2]=2
a[0][3]=3
a[1][0]=1
a[1][1]=2
a[1][2]=3
a[1][3]=4
a[2][0]=2
a[2][1]=3
a[2][2]=4
a[2][3]=5
```

Two-dimensional arrays are declared and addressed as `array[row][column]`. This is the same as done in e.g. Matlab and matrix indexing in general.

```
#define ROWS 3
#define COLUMNS 4
  int a[ROWS][COLUMNS];
  for (int r=0; r < ROWS; r++) {
    for (int c=0; c < COLUMNS; c++) {
      a[r][c] = c+r;
    }
  }
  for (int r=0; r < ROWS; r++) {
    for (int c=0; c < COLUMNS; c++) {
      printf("a[%d][%d]=%d\n",r,c,a[r][c]);
    }
  }
```

2. Now also print out the addresses of each element of the array. Can you explain the addresses? (As always with addresses; yours may be different.)

```
a[0][0]=0 address 0x7ffeead0e8e0
a[0][1]=1 address 0x7ffeead0e8e4
a[0][2]=2 address 0x7ffeead0e8e8
a[0][3]=3 address 0x7ffeead0e8ec
a[1][0]=1 address 0x7ffeead0e8f0
a[1][1]=2 address 0x7ffeead0e8f4
a[1][2]=3 address 0x7ffeead0e8f8
a[1][3]=4 address 0x7ffeead0e8fc
a[2][0]=2 address 0x7ffeead0e900
a[2][1]=3 address 0x7ffeead0e904
a[2][2]=4 address 0x7ffeead0e908
a[2][3]=5 address 0x7ffeead0e90c
```

See the next question for a detailed explanation.

```
printf("a[%d][%d]=%d address %p\n",r,c,a[r][c],&a[r][c]);
```

3. Now also print out the difference of each address with the start of the array. Explain the offsets.

```
a[0][0]=0 address 0x7ffeead0e8e0, offset 0
a[0][1]=1 address 0x7ffeead0e8e4, offset 1
a[0][2]=2 address 0x7ffeead0e8e8, offset 2
a[0][3]=3 address 0x7ffeead0e8ec, offset 3
a[1][0]=1 address 0x7ffeead0e8f0, offset 4
a[1][1]=2 address 0x7ffeead0e8f4, offset 5
a[1][2]=3 address 0x7ffeead0e8f8, offset 6
a[1][3]=4 address 0x7ffeead0e8fc, offset 7
a[2][0]=2 address 0x7ffeead0e900, offset 8
a[2][1]=3 address 0x7ffeead0e904, offset 9
a[2][2]=4 address 0x7ffeead0e908, offset 10
a[2][3]=5 address 0x7ffeead0e90c, offset 11
```

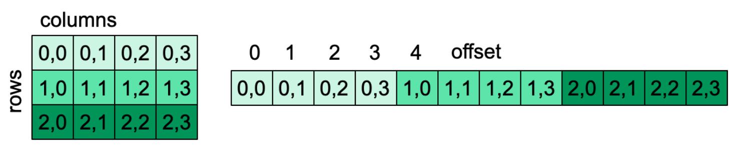

Figure 5: Two-dimensional array

In C you add or subtract an integer from a pointer, e.g. a+3 which is equivalent to array indexing a[3]. But it is also allowed to subtract two pointers *as long as they point into the same array*. The result is then the difference in terms of number of elements. For example, &a[3]-&a[0] is the same as as (a+3)-(a+0), i.e. 3. It is important to note that all this is true independently of the type of the array elements. Array elements are always successive in memory.

Considering two-dimensional arrays, nothing changes. However, the difference of (e.g.) &a[2][3]-&a[0][0] is now (a+2*COLUMNS+1)-(a+0*COLUMNS+0) is 2*COLUMNS+1 i.e. 9. Thus element 2,1 is stored in memory 9 integers after element 0,0. Array elements are still always successive in memory. But additionally the memory layout is *row major*, i.e. we first get all elements of row 0, then all elements of row 1, etc. See the lecture notes and book for some figures.

```
printf("a[%d][%d]=%d address %p, offset %ld\n",r,c,a[r][c],&a[r][c]-&a[0][0]);
```

(The format for the offset is %ld instead of %d because subtracting two pointers results in a long integer.)

4. Now print the offset of each element in terms of bytes.

```
a[0][0]=0 address 0x7ffee75238e0, offset 0 bytes
a[0][1]=1 address 0x7ffee75238e4, offset 4 bytes
a[0][2]=2 address 0x7ffee75238e8, offset 8 bytes
a[0][3]=3 address 0x7ffee75238ec, offset 12 bytes
a[1][0]=1 address 0x7ffee75238f0, offset 16 bytes
a[1][1]=2 address 0x7ffee75238f4, offset 20 bytes
a[1][2]=3 address 0x7ffee75238f8, offset 24 bytes
a[1][3]=4 address 0x7ffee75238fc, offset 28 bytes
a[2][0]=2 address 0x7ffee7523900, offset 32 bytes
a[2][1]=3 address 0x7ffee7523904, offset 36 bytes
a[2][2]=4 address 0x7ffee7523908, offset 40 bytes
a[2][3]=5 address 0x7ffee752390c, offset 44 bytes
```

All you need to do is to multiply the index by the size of the element of the array.

```
printf("a[%d][%d]=%d address %p, offset %lu bytes\n",
       r,c,a[r][c],&a[r][c], sizeof(int)*(&a[r][c]-&a[0][0]));
```

5. Instead of printing row by row, now print the array column by column. Why are the offsets no longer increasing by four every time?

```
a[0][0]=0 address 0x7ffee58578e0, offset 0 bytes
a[1][0]=1 address 0x7ffee58578f0, offset 16 bytes
a[2][0]=2 address 0x7ffee5857900, offset 32 bytes
a[0][1]=1 address 0x7ffee58578e4, offset 4 bytes
a[1][1]=2 address 0x7ffee58578f4, offset 20 bytes
a[2][1]=3 address 0x7ffee5857904, offset 36 bytes
a[0][2]=2 address 0x7ffee58578e8, offset 8 bytes
a[1][2]=3 address 0x7ffee58578f8, offset 24 bytes
a[2][2]=4 address 0x7ffee5857908, offset 40 bytes
a[0][3]=3 address 0x7ffee58578ec, offset 12 bytes
a[1][3]=4 address 0x7ffee58578fc, offset 28 bytes
a[2][3]=5 address 0x7ffee585790c, offset 44 bytes
```

Most importantly, *nothing has changed in the memory layout of the two-dimensional array!* Previously we printed elements (0,0), (0,1), (0,2), (0,3), (0,4), (1,0), etc. which were successive in memory. However, now we print elements (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), etc. which are *not successive* in memory.

```
printf("a[%d][%d]=%d address %p, offset %ld\n",r,c,a[r][c],&a[r][c]-&a[0][0]);
```

5 Arrays of pointers

1. We want to create a new program that implements a lower triangular array. This is an array in which row i has i elements (starting in the first column). A lower triangular matrix of 4 rows and 4 columns thus has elements (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3). The remaining elements are zero and do not need to be stored.

   How would you do this?

   A simple two-dimensional matrix is square and stores all 16 elements, which is not what we want. The section heading clearly gave the answer. We can declare an array of 4 (rows) pointers. We then malloc an array of r integers for each row i. Do this for the next exercise.

2. Create a lower triangular array of size 4x4 with element (i,j) set to 10*i+j. Print out the array with the offset of each element with respect to the first element of its row using a pointer calculation (as before). Use the solution to the previous question. Can you explain the offsets?

```
a[0][0]=0 offset 0
a[1][0]=10 offset 0
a[1][1]=11 offset 1
a[2][0]=20 offset 0
a[2][1]=21 offset 1
a[2][2]=22 offset 2
a[3][0]=30 offset 0
a[3][1]=31 offset 1
a[3][2]=32 offset 2
a[3][3]=33 offset 3
```

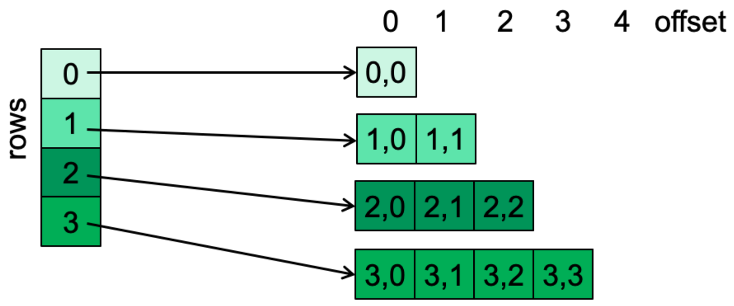

Figure 6: An array of pointers to malloc'd arrays (rows).

Since each row is a consecutive number of integers (malloc'ed per row) the indices are consecutive. Note that in the program below we can still use the notation `a[r][c]` to index. The reason is that `a[r]` is a pointer to a malloc'd memory region, which we can interpret as an array. Thus we can index `a[r]` like this `a[r][c]`. If that confuses you, then you can use an intermediate variable instead: `int *row = a[r]; ... row[c] ....`

```
#define ROWS 4
  int *a[ROWS];
  for (int r=0; r < ROWS; r++) {
    a[r] = (int *) malloc((r+1) * sizeof(int));
    for (int c=0; c <= r; c++) {
      a[r][c] = 10*r +c;
    }
  }
  for (int r=0; r < ROWS; r++) {
    for (int c=0; c <= r; c++) {
      printf("a[%d][%d]=%d offset %ld\n",r,c,a[r][c], a[r][c]-a[r][0]);
    }
  }
}
```

3. Now also print out the address of each element. Can you explain the addresses?

```
a[0][0]=0 address 0x7fcc77405950
a[1][0]=10 address 0x7fcc77405980
a[1][1]=11 address 0x7fcc77405984
a[2][0]=20 address 0x7fcc774059c0
a[2][1]=21 address 0x7fcc774059c4
a[2][2]=22 address 0x7fcc774059c8
a[3][0]=30 address 0x7fcc77405a20
a[3][1]=31 address 0x7fcc77405a24
a[3][2]=32 address 0x7fcc77405a28
a[3][3]=33 address 0x7fcc77405a2c
```

The addresses are no longer consecutive from element 0,0:

```
a[0][0]=0 address 0x7fcc77405950, offset 0 bytes
a[1][0]=10 address 0x7fcc77405980, offset 48 bytes
a[1][1]=11 address 0x7fcc77405984, offset 52 bytes
a[2][0]=20 address 0x7fcc774059c0, offset 112 bytes
a[2][1]=21 address 0x7fcc774059c4, offset 116 bytes
a[2][2]=22 address 0x7fcc774059c8, offset 120 bytes
a[3][0]=30 address 0x7fcc77405a20, offset 208 bytes
a[3][1]=31 address 0x7fcc77405a24, offset 212 bytes
a[3][2]=32 address 0x7fcc77405a28, offset 216 bytes
a[3][3]=33 address 0x7fcc77405a2c, offset 220 bytes
```

However, you can see that *in each row the elements are consecutive* because they are in the same malloc'd region. But from row to row the start address can vary because the malloc for each row can be placed anywhere in memory. (In your output you'll probably see addresses that are almost consecutive because you're making successive mallocs. But that cannot be relied upon! So I did some random mallocs between to force different addresses. Moreover, it is not valid C to compute the difference between the address of `[0][0]` and `[i][j]` because (for $i, j > 0$) row 0 and row j are not the same array because they are malloced separately.)

```
   int *a[ROWS];
   for (int r=0; r < ROWS; r++) {
     a[r] = (int *) malloc((r+1) * sizeof(int));
     for (int c=0; c <= r; c++) {
       a[r][c] = 10*r +c;
     }
   }
   for (int r=0; r < ROWS; r++) {
     for (int c=0; c <= r; c++) {
       printf("a[%d][%d]=%d address %p\n",r,c,a[r][c],&a[r][c]);
     }
   }
 }
```

4. We will now write a program that implements the Unix "history" function, i.e. allows you to go back to the previous commands that you typed. Write a new program that declares an array of 6 pointers to characters and initialise the entries to NULL. Make a while loop and ask for a command. A command is a word (i.e. can be read with %s). Any command can be typed and you don't need to do anything with it yet, except that "quit" exits the program.

```
Command? hgjf
Command? nothing
Command? quit
Bye!
```

```
#define HISTORY 10
  char *history[HISTORY] = { NULL };
  char cmd[100];
  do {
    printf("Command? ");
    scanf("%s",cmd);
  } while (strcmp(cmd, "quit"));
  printf("Bye!\n");
```

5. Every time a command is entered, remove the oldest command from the history, and add the new command. The latest command is index 0 and the oldest command is index HISTORY-1. When the command that is typed is "history" then you should print the history (oldest first).

```
Command? cd
Command? ls
Command? mkdir
Command? cd
Command? ls
Command? history
5 cd
4 ls
3 mkdir
2 cd
1 ls
0 history
Command? diff
Command? cd
Command? history
5 cd
4 ls
3 history
2 diff
1 cd
0 history
Command? quit
Bye!
```

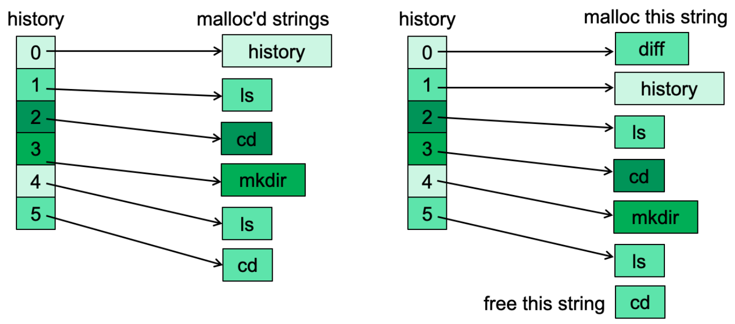

Figure 7: Left: the history array and strings after the first "history" command. Right: after the "diff" command.

```
#define HISTORY 6
  char *history[HISTORY] = { NULL };
  char cmd[100];
  do {
    printf("Command? ");
    scanf("%s",cmd);
    // if history is full, free the oldest one
    if (history[HISTORY-1] != NULL) free(history[HISTORY-1]);
    // shift everyone up by one place
    for (int i=HISTORY-2; i >= 0; i--) history[i+1] = history[i];
    // insert the latest command at the start of the history
    history[0] = (char *) malloc (strlen(cmd));
    strcpy(history[0],cmd);
    // print the history if asked
    if (!strcmp(cmd, "history")) {
      for (int i=HISTORY-1; i >= 0; i--)
        if (history[i] != NULL) printf("%d %s\n", i, history[i]);
    }
  } while (strcmp(cmd, "quit"));
  printf("Bye!\n");
```

6. Now implement the '!command' functionality. When the command starts with an exclamation mark then lookup the latest command in the history with that prefix (i.e has the same starting characters). If the command is found then use it as the newly entered command.

```
Command? hello
Command? there
Command? how
Command? !he
2 hello
Command? history
4 hello
3 there
2 how
1 hello
0 history
Command? !what
Not found in history
Command? history
5 hello
4 there
3 how
2 hello
1 history
0 history
Command? quit
Bye!
```

```
#define HISTORY 6
  char *history[HISTORY] = { NULL };
  char cmd[100];
  do {
    printf("Command? ");
    scanf("%s",cmd);
    if (cmd[0] == '!') {
      for (int i=0; i < HISTORY; i++) {
        if (history[i] != NULL && strncmp(&cmd[1], history[i], strlen(cmd)-1) == 0) {
          printf("%d %s\n", i, history[i]);
          strcpy(cmd, history[i]);
          break;
        }
      }
      if (cmd[0] == '!') {
        printf("Not found in history\n");
        continue;
      }
    }
    if (history[HISTORY-1] != NULL) free(history[HISTORY-1]);
    for (int i=HISTORY-2; i >= 0; i--) history[i+1] = history[i];
    history[0] = (char *) malloc (strlen(cmd));
    strcpy(history[0],cmd);
    if (!strcmp(cmd, "history")) {
      for (int i=HISTORY-1; i >= 0; i--)
        if (history[i] != NULL) printf("%d %s\n", i, history[i]);
    }
  } while (strcmp(cmd, "quit"));
  printf("Bye!\n");
```

- **22/7 v0.1** First version.
- **1/10 v0.2** Added 2D arrays, arrays of pointers.