Dynamic memory allocation

We know how to work with **static** arrays, i.e. such that their size is known at the time of compilation. Now we need a method to deal with situations where size is unknown at compilation, and is going to be fixed at runtime.

Dynamic memory allocation can be done with **malloc()** function, which is declared in **stdlib.h**. Memory is deallocated with **free()**.

- **#include <stdlib.h>**
- **void * malloc(unsigned int size)**
  - allocates a memory block of **size** bytes
  - returns an address of that memory block
  - it is up to the programmer to free the memory
    - **free()**

Note that **void * malloc()** returns an addres of the memory block that have been allocated as a pointer to **void** (i.e. **void ***). The programmer needs to *cast* the returned pointer to an appropriate data type: **int *** for an array of **int**, **char *** for an array of char, and so on.

1D arrays

We will start with 1D arrays. Those wirk very much the same as the static counterpart. The only difference is in array declaration (memory allocation) and the need to destroy it at the end. Everything else works the same.

The first program allocates a block of 24 bytes and interprets it as an array of characters (how many?)

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    char *tab = (char *)malloc(24); // allocate memory

    for(int i=0; i<24; ++i) // fill the array with characters
        tab[i] = 50 + i;

    for(int i=0; i<24; ++i) // print the content as characters
        printf("%c ", tab[i]);

    free(tab); // Deallocate memory
}
```

Providing the number of bytes to allocate is very inconvinient. The size of variables might vary in between architectures or simply we might not know the size of the data type[1]. The easy solution is to use the **sizeof()** function to estimate the size of a data type.

[1] Up to now we have been using only simple data types (int, double ...) with well defined size. The **composite data type** are constructed from simple as well as other composites types and their size might be difficult to estimate. Also composites can be modified during program development changing their size.

This program reads an integer *n* from standard input and allocates an array of *n* integers.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int n; // Variable to store size
    //scanf("%d", &n); // Read size from keyboard
    n = 10;

    //int tab[10];
    //int tab[n]; <- negative points
    int *tab = (int *)malloc(n*sizeof(int)); // allocate memory for an array of int

    for(int i=0; i<n; ++i) // Fill array with numbers
        tab[i] = i;

    for(int i=0; i<n; ++i) // Print
        printf("%d ", tab[i]);

    free(tab); // Deallocate memory
}
```

In specific problems size of the actual data array can result from different operations. It could be read from keyboard, file, or other input (e.g. a message over network in client-server configuration).

In this example we will read data from a file *data1.dat*. The file contains data defining position (x and y coordinates) of *n* points. The structure of the file is as follows:

n <- number of points

x1 y1 <- coordinates of the first point x2 y2

.. ..

xn yn <- coordinates of the last point

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    FILE *f = fopen("./samples/data1.dat", "r");

    int n;
    fscanf(f, "%d", &n);
    printf("%d\n", n);
    double *x = (double *)malloc(n*sizeof(double));
    double *y = (double *)malloc(n*sizeof(double));

    for(int i=0; i<n; ++i)
    {
        fscanf(f, "%lf %lf", &x[i], &y[i]);
        printf("%lf %lf\n", x[i], y[i]);
    }

    free(y);
    free(x);
    fclose(f);
}
```

Functions and 1D arrays

1D dynamic arrays work the same as static ones as arguments to functions

In this example we will develop functions.

1. Filling an array with data
2. Normalizing the data - squeeze to 0-1 range
3. printing the content of an array.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void fill(double *tab, int n);
void print(double *tab, int n);
void normalize(double tab[], int n);


int main()
{
    srand(time(NULL));
    int n = 10; //scanf()
    double *x = (double *)malloc(n*sizeof(double));

    fill(x, n);
    print(x, n);
    normalize(x, n);
    print(x, n);

    free(x);
}

void fill(double *tab, int n)
{
    for(int i=0; i<n; ++i)
    {
        tab[i] = (double)rand() / RAND_MAX;
    }
}

void print(double *tab, int n)
{
    for(int i=0; i<n; ++i)
    {
        printf("%lf ", tab[i]);
    }
    printf("\n");
}

void normalize(double tab[], int n)
{
    // find min and max
    double max = tab[0], min = tab[0];
    for(int i=1; i<n; ++i)
    {
        if(max < tab[i]) max = tab[i];
        if(min > tab[i]) min = tab[i];
    }
    // Now using max and min scale the values so the max is 1 and min is zero
    for(int i=1; i<n; ++i)
    {
        tab[i] -= min; // (max - min) // we have to use the formula for a linear function
    }
}
```

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int **p = (int **)malloc(2 * sizeof(int*));
    p[0] = (int*)malloc(5 * sizeof(int));
    p[1] = (int*)malloc(5 * sizeof(int));

    // use with p[i][j];

    free(p);
    free(p[0]);
    free(p[1]);
}
```

Lets have a Christmas tree

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand(time(NULL));

    int rows = 10;
    for(int i=0; i<rows; ++i)
    {
        for(int j=0; j<rows-i; ++j)
        {
            printf(" ");
        }
        for(int j=0; j<2*i+1; ++j)
        {
            if(rand()%10 == 0)
                printf("o");
            else
                printf("*");
        }
        printf("\n");
    }

}
```

2D arrays

Contrary to 1D dynamic arrays, dynamically allocated 2D arrays (or any higher dimensional constructs) **differ** from their static counterparts. The source of the differences lies in the way data is stored in memory. In the case of static arrays the memory is guaranteed to be *continously* occupied. In the dynamic case memory distribution is up to the programmer.

The main feature we need from a 2D array is the ability to acces data with two indices, i.e.: A[i][j]. Since a single square bracket operator was infact an **indirection** operator * applied to an array (pointer) the double square brackets would correspond to a double indirection operator **, or in other words the 2D array would be equivalent to a pointer to a pointer (a double pointer) (i.e. **int ** A**).

tab[i] -> *(tab + i)
tab[i][j] -> *(tab[i] + j) -> *(*(tab + j) + j)

We start with a 1D array of 6 integers, that we would like to interpret as a 2x3 2D array.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int n = 6; // we fix the number of elements to 6

    int *tab = (int *)malloc(n*sizeof(int)); // allocate

    tab[0] = 11; tab[1] = 12; tab[2] = 13; // first row to be
    tab[3] = 21; tab[4] = 22; tab[5] = 23; // second row to be

    printf("%d %d %d\n", tab[0], tab[1], tab[2]);
    printf("%d %d %d\n", tab[3], tab[4], tab[5]);

    free(tab);
}
```

This is not a 2D array, i.e. we can not acces elements as tab[1][2] to get 23. We modify the program such that rows of our intended 2D structure are referenced by different pointers.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int n = 6;

    int *tab = (int *)malloc(n*sizeof(int));

    tab[0] = 11; tab[1] = 12; tab[2] = 13;
    tab[3] = 21; tab[4] = 22; tab[5] = 23;

    int *p0 = &tab[0]; // First row
    int *p1 = &tab[3]; // Second row

    printf("%d %d %d\n", p0[0], p0[1], p0[2]);
    printf("%d %d %d\n", p1[0], p1[1], p1[2]);

    free(tab);
}
```

So we can acces data stored in tab as two separate arrays, a bit better but still not a 2D array. We modify the program further and replace pointers *p0* and *p1* with a static 1D arrays of pointers to integers (**int ***).

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int n = 6;

    int *tab = (int *)malloc(n*sizeof(int));
```

```
    tab[0] = 11; tab[1] = 12; tab[2] = 13;
    tab[3] = 21; tab[4] = 22; tab[5] = 23;

    int *A[2]; // Static 1D array of int *
    A[0] = &tab[0]; // First row
    A[1] = &tab[3]; // Second row

    //We can now access the elements with [][] !!
    printf("%d ", A[0][0]); printf("%d ", A[0][1]); printf("%d\n", A[0][2]);
    printf("%d ", A[1][0]); printf("%d ", A[1][1]); printf("%d\n", A[1][2]);

    free(tab);
}
```

Now we can access the data with double square brackets, in other words data can be interpreted as a 2D array. A is a 1D array, so A[i] returns the i'th element of A. The type stored in A is **int *** so A[i] is a pointer to which we can apply square brackets. Finally A[i][j] returns an integer.

Our final modification is to make A a dynamically allocates array, note that A stores **int ***, so the type we need for dynamic allocation is **int ****.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int n = 6;

    int *tab = (int *)malloc(n*sizeof(int));

    tab[0] = 11; tab[1] = 12; tab[2] = 13;
    tab[3] = 21; tab[4] = 22; tab[5] = 23;

    int **A = (int **)malloc(2 * sizeof(int *)); // Dynamic allocation of A
    A[0] = &tab[0]; // Assign address of the first row
    A[1] = &tab[3]; // Address of the second row

    //We can now access the elements with [][] !!
    printf("%d ", A[0][0]); printf("%d ", A[0][1]); printf("%d\n", A[0][2]);
    printf("%d ", A[1][0]); printf("%d ", A[1][1]); printf("%d\n", A[1][2]);

    free(A); // Deallocate A
    free(tab);// Deallocate tab
}
```

Working with dynamic 2D arrays differs from what we did with static 2D arrays. The main difference is in the way of passing arrays to functions. In the case of static 2D arrays we did it with a type, variable name and two square brackets, the number of columns needed to be passed as well. E.g.:
int A[][M] for a 2D array of integers with M (known at compilation) collumns. A dynamic array will not work with such a function (can you explain why?), instead we need to pass the pointer-to-pointer variable, i.e. **int ****.

In this example we will develop a function that prints values storred in a n x m 2D array. The array is created based on values stored in a file **data2.dat**. The structure of the file is as follows:

n m <- number of rows, collumns

a11 a12 ... a1m <- first row

a21 a22 ... a2m <- second row

... ...

an1 an2 ... anm <- the last row

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

// Prints the content of a 2D array od integers
// n - number of rows
// m - number of collumns
void print(int **A, int n, int m)
{
    printf("\nThe content of a 2D array:\n");
    for(int i=0; i<n; ++i) // all rows
    {
        printf("["); // a nice bracket
        for(int j=0; j<m; ++j) // all collumns
        {
            printf("%d, ", A[i][j]);
        }
        printf("\b\b]\n"); // two backspaces and a nice bracket
    }
}

int main(){
    FILE *f = fopen("data2.dat", "r"); // Open a file
    int n, m; // rows and collumns
    fscanf(f, "%d", &n); // read number of rows
    fscanf(f, "%d", &m); // read number of collumns
    printf("The array is %d x %d\n\n", n, m);

    int **A = (int **)malloc(n * sizeof(int *)); // Allocate A
    int *p = (int *)malloc(n * m * sizeof(int)); // Allocate space for data
    for(int i=0; i<n; ++i) // Assign addreses to elements of A
    {
        // p is the begining of the memory segment,
        // m is the number of elelents in a single row
        A[i] = p + i * m;
        //Print addresses of rows
        printf("Address of %d row is %p \n", i, A[i]);
    }

    for(int i=0; i<n; ++i) // Read data from a file
    {
        for(int j=0; j<m; ++j)
        {
            fscanf(f, "%d", &A[i][j]);
        }
    }

    print(A, n, m); // Use function print

    // Deallocate memory and close the file
    free(p);
    free(A);
    fclose(f);
}
```

The array in the example above is 3x2, the number of elements in a row is 2, the elements are integers so spacing betwin rows is 8 bytes. Verify the addresses printed above. Also have a look at line 34 where adreses are assigned to elements of A.

In [ ]:

```
#include <stdio.h>
```

```c
#include <stdlib.h>

#define MAX_SIZE 10

void fill(int A[][MAX_SIZE], int r, int c)
{
    for(int i=0; i<r; ++i)
    {
       for(int j=0; j<c; ++j)
       {
           A[i][j] = i + j + 1;
       }
    }
}
void print(int A[][MAX_SIZE], int r, int c)
{
    for(int i=0; i<r; ++i)
    {
       for(int j=0; j<c; ++j)
       {
           printf("%d ", A[i][j]);
       }
       printf("\n");
    }
}

void copydiag(int A[][MAX_SIZE], int r, int c, int d[])
{
    for(int i=0; i<r; ++i)
    {
        d[i] = A[i][i];
    }
}
void copyrow(int A[][MAX_SIZE], int r, int ri, int d[])
{
    for(int i=0; i<r; ++i)
    {
        d[i] = A[ri][i];
    }
}
void insertrow(int A[][MAX_SIZE], int r, int ri, int d[])
{
    for(int i=0; i<r; ++i)
    {
         A[ri][i] = d[i];
    }
}

int main(){
    int tab[MAX_SIZE][MAX_SIZE];
    fill(tab, 3, 3);
    print(tab, 3, 3);

    printf("---\n");

    //int d[MAX_SIZE];
    int *d = (int *)malloc(3 * sizeof(int));
    copyrow(tab, 3, 1, d);
    for(int i=0; i<3; ++i)
        printf("%d ", d[i]);

    free(d);
}
```

aaa

In [7]:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n = 3;
    int m = 5; // rows
    double *p = (double*)malloc(n*m*sizeof(double));
    double **tab = (double**)malloc(m*sizeof(double*));

    //p[i][j] *(*(p+1)+num_col*j)
    //tab[i][j]
    //tab[0] = p;
    //tab[1] = p+n;
    for(int i=0; i<m; ++i)
    {
        tab[i] = p+i*n;
    }
    tab[0][0] = 10;
    tab[0][1] = 11;
    tab[0][2] = 12;

    tab[3][0] = 30;
    tab[3][1] = 31;
    tab[3][2] = 32;

    for(int i=0; i<m; ++i)
    {
        for(int j=0; j<n; ++j)
            printf("%lf ", tab[i][j]);
        printf("\n");
    }

    for(int i=0; i<n*m; ++i)
        printf("%lf ", p[i]);

    free(p);
    free(tab);
}
```

```
10.000000 11.000000 12.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
30.000000 31.000000 32.000000
0.000000 0.000000 0.000000
10.000000 11.000000 12.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 30.000
000 31.000000 32.000000 0.000000 0.000000 0.000000
```

In [ ]: