Background
Understanding C Programs

J.R Herrero

Spring 2024

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Contents

Homework 1

Understanding C programs

1.1 Introduction

Plenty of resources can be found on the Internet. We have compiled a set of entries for your perusal. They can be reached in Aula-ESCI at:
`https://aula.esci.upf.edu/mod/page/view.php?id=242998`

In addition, we have prepared this document and a set of examples. The goal of this material is to provide the necessary background for understanding codes written in C since this is the programming language which we will be using during the term in order to learn parallel programming, follow the course examples and exercises, and do most of the laboratory assignments in the course. Depending on your background on programming and data structures this material can be very necessary, or mostly redundant with your previous knowledge. Therefore, **this work is optional** and you will not be required to deliver any report. **However, please make sure to master all the concepts presented in this preliminary assignment in order to be able to follow the course:**
using simple programs we will illustrate how one can access a *matrix*, i.e. a *2 dimensional array*, by rows or by columns. We will start using *row-wise* storage, also called *row-major* order, for the matrix since this is the standard in C and C++ when using a matrix allocated *statically* in memory (memory space reserved at compilation time). Later, we will illustrate how to use *pointers* and how *pointer arithmetic* works. Afterwards, we will see the equivalent matrix codes using *dynamic memory allocation*, i.e. memory allocated at execution time. Also, we will see how the 2D indices can be translated into a single index (1D). Finally, we will see how one can change the row-major order into *column-major* order. **If you master all this topics listed above, then you can fully skip this material**.

This short introduction is accompanied by some source codes. You can learn by inspecting each code and observing the output of its execution. In order to use them, you need to extract the files from file `lab0.tar.gz`. You can unpack the files with the following command line: `"tar -zxvf lab0.tar.gz"`. It will create a subdirectory `lab0` containing the source files together with a `Makefile` for compiling the C and C++ files. This is simply achieved by changing the current working directory via the execution of command `"cd lab0"` followed by the execution of the command `"make"`.

Instructions assume you boot your PC/terminal with Linux and have `gcc`, `g++` and `make` installed. For the Python codes, we will assume that the interpreter can be invoked as `python3`[1]. Execution of the python codes can be done by typing `"python3 basename.py"`, where `basename` must be replaced by the *basename* (file name appearing before the `.py`) in each file name. The `Makefile` is prepared to generate executable files for the C++ codes ending in `_cpp`, i.e. the basename is appended so that the executable generated is called `basename_cpp`. Similarly, the executables generated from C codes will end in `_c`. You can execute them as `"./basename_cpp"` and `"./basename_c"` respectively. You can avoid prepending the `"./"` each time if you add the current working directory "." to the `PATH` environment variable. Assuming we use the `bash` shell this can be done as follows: `export PATH=.:$PATH`. Instead of typing it each time, one can automate this configuration of the behavior of the shell by adding this line within file `.bashrc`. Future sessions will load this file if it exists and configure your execution environment automatically.

[1]You can find the way to do the equivalent in systems booted with Windows or MacOS X.

1.2 Accessing 2 Dimensional Arrays (Matrices)

A *matrix* is an object typically used in mathematics and many areas of science and engineering. In words, it is defined as a set of numbers arranged in rows and columns so as to form a rectangular array. The numbers are called the *elements*, or *entries*, of the matrix.

Using simple programs[2] written in Python, C++ and C, we will illustrate how one can access a matrix, i.e. a 2 dimensional array, by rows or by columns. We will start using *row-wise* storage, also called *row-major* order, for the matrix since this is the standard in C and C++ when using a matrix allocated statically in memory (memory space reserved at compilation time).

Afterwards, we will see the equivalent code using *dynamic memory allocation*, i.e. memory allocated at execution time. Also, we will see how the 2D indices can be translated into a single index (1D). Later, we will see how one can change the row-major order into *column-major* order.

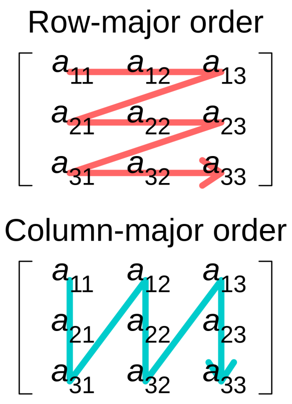
Row-major order
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Figure 1.1: A 3 by 3 matrix stored by rows or by columns (source Wikipedia).

1.2.1 Row-major storage and access by rows

The following code snippets demonstrate how to access a matrix by rows in Python, C++, and C.

The python code available in file `matrix_by_rows.py` directly iterates by rows, printing them at once.

```
# Assuming matrix is a 2D list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing matrix by rows
for row in matrix:
    print(row)
```

[2]These programs can be found in directory `lab0`. C and C++ codes can be automatically compiled by executing the command `make`.

C++ can also provide some high level abstraction to handle the access row by row, and element by element within each row. The code is available in file `matrix_by_rows.cpp`:

```
#include <iostream>
#include <vector>

int main() {
    // Assuming matrix is a vector of vectors
    std::vector<std::vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Accessing matrix by rows
    for (const auto &row : matrix) {
        for (int element : row) {
            std::cout << element << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

However, in C we must write code to explicitly traverse the data structure accessing each value one at a time. In this example, variable `matrix` is a 2 dimensional array representing a matrix. The appearances of `matrix[i][j]` in the code access element `i,j` within the matrix, being `i` the variable used to indicate the row and `j` the variable used to indicate the column within the matrix. Note that `i` indicates the **row** since it is used as **the** 1^{st} **index** when accessing the 2D array, e.g. `matrix[i][.]`; similarly `j` indicates the **column** since it is used as **the** 2^{nd} **index**, e.g. `matrix[.][j]`. Loop `i` is used to traverse the rows; loop `j` is used to traverse the columns. The following code has loop `j` as the inner loop. This means that, given a row indicated by the outer loop `i`, all the elements in that row, i.e. all the columns within that row, are traversed. Only after the whole inner loop has been traversed, the next iteration of the outer loop will be executed, allowing the access to the next row. The code is available in file `matrix_by_rows.c`:

```
#include <stdio.h>

int main() {
    // Assuming matrix is a 2D array
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Accessing matrix by rows
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

1.2.2 Row-major storage and access by columns

Files with basename `matrix_by_cols` illustrate how to **access a matrix by columns**. Again, we provide versions coded in Python, C++, and C respectively. Note that these codes continue to **store the matrix by rows**.

The Python code is available in file `matrix_by_cols.py`:

```
# Assuming matrix is a 2D list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing matrix by columns
for col in zip(*matrix):
    print(col)
```

Note that, **in order to access the matrix by columns, in the C and C++ codes loop `i` becomes the inner loop**. Consequently, loop `j`, which traverses the columns, becomes the outer loop. This can be observed by highlighting the differences between the versions traversing rows and columns.

In C++:

```
diff matrix_by_rows.cpp matrix_by_cols.cpp
  ...
<     // Accessing matrix by rows
<     for (const auto &row : matrix) {
<         for (int element : row) {
<             std::cout << element << " ";
---
>     // Accessing matrix by columns
>     for (size_t j = 0; j < matrix[0].size(); ++j) {
>         for (size_t i = 0; i < matrix.size(); ++i) {
>             std::cout << matrix[i][j] << " ";
```

In C:

```
diff matrix_by_rows.c matrix_by_cols.c
  ...
<     // Accessing matrix by rows
<     for (int i = 0; i < 3; ++i) {
<         for (int j = 0; j < 3; ++j) {
---
>     // Accessing matrix by columns
>     for (int j = 0; j < 3; ++j) {
>         for (int i = 0; i < 3; ++i) {
```

The previous C and C++ codes used a matrix allocated statically at compilation time. However, in practise, most times it is necessary to request memory space once the problem size is provided by the user. When we ask for memory dynamically, at execution time, the system will return a memory address which we can subsequently use for storing the data. Therefore, we need to know how to handle memory addresses.

1.3 Pointers

In C, **pointers** act like *signposts* pointing to memory locations. They **store memory addresses**, not data themselves. Pointers are powerful for memory management and working with data structures in C. However, one needs to handle them with care as they can be tricky if not used properly. Here's the gist:

Declaring a Pointer: Let's assume we want to have a variable named *iptr* for pointing to an integer, or more precisely, for pointing to (keeping the address of) the memory area where the integer is stored. The code will contain a statement such as "`int* iptr;`". Please, note the `*` *after* int. This means *iptr* can hold memory addresses of some integer. Such an statement makes the compiler reserve space for keeping an address. The amount of memory in bytes needed for the pointer can be retrieved via "`sizeof(int*)`", which typically returns an 8 in modern 64 bit systems. In contrast, "`sizeof(int)`" typically returns a 4. It is **very important** to note that *pointer declaration saves space for the pointer; however, it neither reserves space for an integer nor initializes the pointer to point to a meaningful address in memory.* Uninitialized pointers are sometimes referred to as *wild pointers.*

Getting an Address: The & (address-of) operator retrieves the memory address of a variable. For example, `&num;` gives the address where a variable named *num* is stored.

Pointing the Pointer: We can assign the address obtained from & to a pointer variable. e.g. `iptr = &num;` Now *iptr* points to the memory location where variable `num` is stored. Similarly, we can use the pointer variable to save a memory address obtained dynamically at execution time. This will be presented in section 1.4.

Following the Pointer: The $*$ (*dereference*) operator is meant for looking at what's stored at the address a pointer points to. Contrary to the pointer declaration, the $*$ used as dereference operator is placed *in front* of the variable of pointer type. We must understand $*$*iptr* as "the value at the location pointed to by *iptr*". Therefore, given the previous definition `iptr = &num;`, if we have an integer variable `int newnum;` we can then write `newnum = *iptr;`. A question arises, is this statement equivalent to `newnum = num;`? The answer depends on the data type of `num` (see *Pitfalls* below).

Modification Through Pointers: One can change the value at the memory location a pointer points to using $*$*iptr* = newValue .

Example: File `pointers.c` illustrates the usage of pointers to access a variable of data type `int`. Observe how a variable can be modified through a pointer defined to point to it.

Pitfalls: Note that the assignment `newnum = *iptr;` will copy 4 bytes, i.e. the size of an `int`, since `iptr` is defined as `int*`. This will happen regardless of the data type of variable `num` to which it is pointing after the assignment `iptr = &num;`. Thus, those 4 bytes being copied will only make sense as an integer in case `num` has data type `int` (or an array of integers, from which the 1^{st} element will be retrieved). Otherwise, we will most likely be interpreting erroneously as an integer the new contents of variable `newnum` consisting of 4 contiguous bytes copied from the initial address of `&num;` followed possibly by the contents of other variable(s) stored there. For instance, if we have defined an an array of characters `char[10] num;` then we would be copying the first 4 characters in the array, interpreting the pack of those 4 bytes as an integer in variable `newnum`, which most likely does not make any sense.

File `pointers_pitfalls.c` illustrates this kind of error. Beware that, at most, the compiler could warn us with a message:

```
> make pointers_pitfalls_c
gcc       -o pointers_pitfalls_c pointers_pitfalls.c
pointers_pitfalls.c: In function main:
pointers_pitfalls.c:17:7: warning: assignment to int * from incompatible pointer type char (*)[10]
                                   [-Wincompatible-pointer-types]
   17 |   ptr = &num;
      |       ^
```

However, the executable is still created and can be run. Obviously, with an erroneous behavior:

```
> pointers_pitfalls_c
Value of the variable 'num': 123456789
Memory address stored in 'ptr': 0x7ffc4e34311e
Value accessed through the pointer 'ptr': 875770417
Changing the value through the pointer 'ptr' to: -43210
Value of the variable 'num' after modification through pointer: 6Wÿÿ56789
```

Note that variable `num` holds a sequence of characters from 1 to 9. However, they are not numbers, but characters. Each of them occupies 1 byte and is coded using some convention (ASCII). When we try to print the value pointed by `ptr`, it takes the first four bytes '1', '2', '3' and '4' in the initial positions of array `num` as a single pack of 4 bytes and interprets them as an integer coded in the internal representation of the machine. Thus, it prints number 875770417. When we write a -43210 through the pointer we overwrite the first 4 bytes of the character array. Then, the initial '1234' is replaced by the '6Wÿÿ' before the '56789' which remain unchanged. Thus, now the character array `num` stores the characters 6Wÿÿ56789. This illustrates that sometimes errors appear somewhere but the error is due to a pointer being misused somewhere else in the program. We must always check for the proper initialization of pointers and the data types they point to.

In addition, when a pointer stores an address of memory allocated dynamically (see section 1.4), using that memory is not valid after releasing that memory (using `delete` in C++ and `free` in C). Note, however, that the pointer continues to store the very same address unless we overwrite it with another address. Therefore, we should stop using the pointer until we change it to point to another valid address in the logical address space of the process. A pointer pointing to a memory location that has been freed (deleted) is called a *dangling* or a *stray* pointer. Such a situation can lead to unexpected behavior in the program and serve as a source of bugs in C and C++ programs.

1.3.1 Pointer Arithmetic in C

Pointers in C don't just store memory addresses; they can perform basic arithmetic on those addresses. This allows us to efficiently navigate memory blocks and access elements within arrays.

Concept: Imagine an array `arr` of integers. Each element in the array occupies a consecutive memory location. Let's assume we have an integer pointer `int* ptr` pointing to the beginning of the array (i.e. `ptr = &arr[0]`). Then, adding an integer `n` to a pointer moves it `n` elements forward in memory. So, `ptr+n` points to the n^{th} element (`&arr[n]`). Let's see an example:

```
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  int *ptr = arr; // ptr points to &arr[0]

  printf("Value at arr[0] through *ptr: %d\n", *ptr); // Access value using dereference
  printf("Value at arr[2] through *(ptr + 2) using pointer arithmetic: %d\n", *(ptr + 2));

  return 0;
}
```

This code produces the following output:

```
Value at arr[0] through *ptr: 1
Value at arr[2] through *(ptr + 2) using pointer arithmetic: 3
```

The code above declares and initializes an integer array `arr`; and a pointer `ptr` that is initialized to point to the first element (note that `arr` is equivalent to `&arr[0]`). The first `printf` accesses the value at `arr[0]` using the dereference operator `*ptr`. The second `printf` demonstrates *pointer arithmetic.* Adding 2 to `ptr` provides the address two elements forward in memory, effectively pointing to `&arr[2]`. Thus, the value at that memory location is accessed using `*(ptr + 2)`.

Notes:

- Pointer arithmetic does not check that the resulting memory address stays within the valid bounds of the memory block the pointer points to. Be cautious to avoid accessing memory outside the allocated array.
- Implicitly, we are saying that pointer+1 points to the next element. For this, to work, the pointer must have the proper data type. This leads us to the next section of this document.

Pointer arithmetic of pointers to different data types

File `pointer_arithmetic.c` provides C code illustrating pointer arithmetic for integers and characters with comments and explanations.

Integer Array and Pointer Arithmetic The code demonstrates pointer arithmetic for an integer array `int_arr` and a pointer `int_ptr` of type `int*`.

Initial Addresses: In general, if we have a variable called `var` which is not an array, then `var` returns the contents of the variable, while `&var` is the only way to obtain its address in memory. However, when we work with arrays the behavior is different. Consider `pointer_arithmetic.c`: `&int_arr` provides the address of `&int_arr[0]`, the first element in the array. Assigning this address to `int_ptr makes` it point to that memory location. The code also illustrates that `int_arr`, `&int_arr` and `&int_arr[0]` provide the initial address of the array. Note that this only works for arrays. In addition, when we have a 2 dimensional array, i.e., a matrix such as `int matrix[nrows][ncols]` note that the use of matrix[i] is equivalent to writing `&matrix[i][0]`.

Accessing Values: Dereferencing `*int_ptr` allows accessing the value stored at the memory location pointed to by `int_ptr`.

Pointer Arithmetic: Adding 2 to `int_ptr` (`int_ptr += 2`) moves the pointer 2 steps forward in memory. Since integers occupy 4 bytes each, this moves the pointer to point to `&int_arr[2]`.

Difference in Addresses: The difference between the addresses after pointer arithmetic computed as `((long)int_ptr - (long)&int_arr)` represents the number of bytes the pointer moved. This difference should be `2 * sizeof(int)`, which is typically $2 \times 4 = 8$ bytes on most machines. *Note*: Casting the addresses to long before subtraction avoids potential integer overflow issues.

Character Array and Pointer Arithmetic The code showcases pointer arithmetic for a character array `char_arr` and a pointer `char_ptr` of type `char*`.

Similar to Integers: The initial steps with `&char_arr` and dereferencing `*char_ptr` work similarly to the integer case, but they access characters (typically 1 byte each).

Character-Specific Arithmetic: Moving the pointer 2 steps forward (`char_ptr += 2`) advances it by 2 bytes due to the smaller size of characters. This points `char_ptr to &char_arr[2]`.

Difference in Addresses: The address difference calculation `((long)char_ptr - (long)&char_arr)` represents the byte movement. This difference should be `2 * sizeof(char)`, which is typically $2 \times 1 = 2$ bytes on most machines.

Summary of Key Points concerning Pointer Arithmetic:

- Pointer arithmetic movement depends on the data type the pointer points to.
- The difference in addresses after pointer arithmetic reflects the number of bytes moved and the data type size.
- Data types matter!

1.4 Dynamic Memory Allocation

The two codes shown in this section present the equivalent versions of the C and C++ codes which access the matrix by rows (section 1.2.1), but using memory allocated dynamically for the matrix: they allocate space using `new` in C++ and `malloc`[3] in C. In both codes memory allocation is done in two stages. There is an initial request for holding a vector of pointers[4] with `rows` positions. Subsequently, a loop with `rows` iterations performs `rows` requests to allocate memory for each of the rows.

More specifically, in these two codes variable `matrix` is used to access a vector of vectors. Specifically, `matrix` is a *pointer to a vector of pointers to integers*. Is this a confusing long description? It will help to read the statement `int** matrix` from right to left since that will convey the meaning: `matrix` is a pointer (`*`) to a pointer (`*`) to an integer (`int`). Note the singular in the previous sentence. However, this does not mean that other pointers to integers may follow the initial one! Variable `matrix` will keep their initial address, regardless of how many.

Thus, rephrasing and elaborating the details, the initial memory allocation request will return the address in memory where it has reserved space for storing the vector of `int*` with `rows` elements, as requested in the program. This address is then stored in variable `matrix`. Thus, `matrix` must be a pointer and we see a "*". As explained in section 1.3.1, it is fundamental to specify the data type to which the pointer points to. In this case, we are allocating space for a vector of pointers to integers (`int*`). This motivates the appearance of 2 wildcards "**" since we have a pointer "*" to the beginning of a memory area where `int*` will be stored.

Next, in a second stage, a loop is used to allocate space for each of the rows, keeping the initial address of row `i` in position `matrix[i]`. Note the data type `int` when memory space is requested for each row, namely a vector of integers with `cols` elements, because here we allocate space for the numerical values, the integers conforming the matrix rows. Compare this to the initial stage with the allocation of matrix which had to store a vector of pointers to integers `int*`.

C++ codes use the `new` and `delete` clauses to request and free memory.

```
/* File: matrix_by_rows_dyn_2ind.cpp */

int main() {
    // Specify matrix dimensions
    const int rows = 3;
    const int cols = 3;

    // Dynamically allocate memory for the matrix
    int** matrix = new int*[rows];                  // Stage 1
    for (int i = 0; i < rows; ++i) {                // Stage 2
        matrix[i] = new int[cols];
    }

    // Assign values to the matrix
    int count = 1;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i][j] = count++;
        }
    }

    // Accessing matrix by rows
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            std::cout << matrix[i][j] << " ";
        }
```

[3]Alternatively we can use `calloc`. Unlike `malloc`, it initializes the memory to zeros.

[4]A pointer holds a memory address and can be used to access data stored in that memory address (see section 1.3).

```
        std::cout << std::endl;
    }

    // Deallocate dynamically allocated memory
    for (int i = 0; i < rows; ++i) {
        delete[] matrix[i];                  // Free memory of each row
    }
    delete[] matrix;                         // Free vector of pointers

    return 0;
}
```

The C code uses functions `malloc` and `free` from the standard C library to request and free memory:

```
/* File: matrix_by_rows_dyn_2ind.c */

int main() {
    // Specify matrix dimensions
    const int rows = 3;
    const int cols = 3;

    // Dynamically allocate memory for the matrix
    int** matrix = (int**)malloc(rows * sizeof(int*));      // Stage 1
    for (int i = 0; i < rows; ++i) {                        // Stage 2
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }

    // Assign values to the matrix
    int count = 1;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i][j] = count++;
        }
    }

    // Accessing matrix by rows
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Deallocate dynamically allocated memory
    for (int i = 0; i < rows; ++i) {
        free(matrix[i]);                  // Free memory of each row
    }
    free(matrix);                         // Free vector of pointers

    return 0;
}
```

These versions use dynamic memory allocation for matrices in both C++ and C. **Make sure to deallocate the memory properly to avoid memory leaks**.

1.5 Linearizing 2D accesses into 1D

Below you can find the equivalent versions of the C and C++ codes without using an additional vector `int** matrix`. Instead, the 2D access through indices `i` and `j` is linearized into a single index. The linearized index is calculated as `i*cols + j`. Note that `i*cols` computes the *offset* from the beginning of the array to the beginning of row i, being row 0 the first row. Then, within row `i`, we advance to the proper column by adding `j`. Note that, contrary to the codes in section 1.4 which required `nrows+1` requests to allocate memory, the codes in this section solve this issue with a single memory allocation request. An additional advantage is that the memory allocated is guaranteed to be contiguous in memory.

```
    /* File: matrix_by_rows_dyn_1ind.cpp */

    ...
    // Dynamically allocate memory for the matrix
    int* matrix = new int[rows * cols];

    // Assign values to the matrix
    int count = 1;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i * cols + j] = count++;
        }
    }

    // Accessing matrix by rows
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            std::cout << matrix[i * cols + j] << " ";
        }
        std::cout << std::endl;
    }

    // Deallocate dynamically allocated memory
    delete[] matrix;
    ...
}
```

In the following C code, the matrix is also linearized using the formula `i*cols + j`. Note that this determines that the storage follows a *row-major order*. However, storage order does not determine how memory accesses are performed: accesses can be performed by rows, as seen in the initialization loops where `i` appears in the outer loop; or by columns, as in the subsequent printing of the matrix values for which `j` appears in the outer loop.

```
    /* File: matrix_by_rows_dyn_1ind.c */

    ...
    // Dynamically allocate memory for the matrix
    int* matrix = (int*)malloc(rows * cols * sizeof(int));

    // Matrix initialization: row-wise traversal and row-major storage
    int count = 1;
    for (int i = 0; i < rows; ++i) {        // Select row
        for (int j = 0; j < cols; ++j) {    // Traverse all columns in row
            matrix[i * cols + j] = count++;
        }
    }
    // Accessing by columns a matrix stored in row-major order
    for (int j = 0; j < cols; ++j) {        // Select column
        for (int i = 0; i < rows; ++i) {    // Traverse all rows in column
            printf("%d ", matrix[i * cols + j]);
        }
        printf("\n");
    }

    // Deallocate dynamically allocated memory
    free(matrix);
    ...
}
```

Therefore, **the order of the loops determines the order of accesses** to the matrix. And the **expression used to compute the index determines the storage** order. Note however, that traversing the data in the same order in which it is stored is far more efficient since accessing data stored close by exploits *spatial locality* and results in faster accesses from the memory hierarchy, which typically consists of several levels of smaller, but faster, cache memories.

1.6 Column-wise storage in C

Even when C stores matrices in *row-major* order by default, the linearization introduced in the previous section allows us to use *column-major* order. For this we need to change the expression used to compute the index. Note that the matrix access is now linearized into 1D with the expression `j*rows + i`, instead of `i*cols + j` as it was done assuming row-wise storage. This is illustrated in the following code excerpt taken from file `matrix-col-major.c`:

```
    ...
    // Dynamically allocate memory for the matrix stored by columns
    int* matrix = (int*)malloc(rows * cols * sizeof(int));

    // Matrix initialization: column-wise traversal and column-major storage
    int count = 1;
    for (int j = 0; j < cols; ++j) {        // Select column
        for (int i = 0; i < rows; ++i) {    // Traverse all rows in column
            matrix[j * rows + i] = count++;
        }
    }

    // Accessing by columns a matrix stored in column-major order
    for (int j = 0; j < cols; ++j) {
        for (int i = 0; i < rows; ++i) {
            printf("%d ", matrix[j * rows + i]);
        }
        printf("\n");
    }

    // Deallocate dynamically allocated memory
    free(matrix);
    ...
}
```

In this code, the matrix is stored by columns, and the access order is adjusted to iterate over columns first and then rows for both assigning values and printing the matrix. Make sure to deallocate the memory properly to avoid memory leaks.

1.7 Final considerations

Allocating contiguous space for a matrix is usually convenient, both in terms of exploitation of locality in memory access and reduction in calls to `malloc`. On the other hand, one can prefer to read accesses to a 2 dimensional array with 2 indices instead of 1 as was done in the previous two sections. The following code illustrates how both can be achieved for a matrix stored row-wise. The requirement is using an additional vector of pointers properly initialized to have each element point to the beginning of a row:

```
...
// Dynamically allocate memory for the matrix
int*  matrix = (int*)  malloc(rows * cols * sizeof(int ) ); // Matrix of integers  (1D access)
int**    mat = (int**) malloc(rows *        sizeof(int*) ); // Access as mat[i][j] (2D access)

/* Mount mat so that we can use 2 indices to access element mat[row][col] */
for (i = 0; i < rows; i++)
  mat[i] = matrix + i*cols ; /* Let mat[i] point to the beginning of row i of matrix */

/* From here we can use mat[i][j] as an alias of matrix[i*cols + j] */

// Matrix initialization: row-wise traversal and row-major storage
int count = 1;
for (int i = 0; i < rows; ++i) {        // Select row
    for (int j = 0; j < cols; ++j) {    // Traverse all columns in row
        mat[i][j] = count++;            // Equivalent to matrix[i*cols + j] = count++;
    }
}

...

// Deallocate dynamically allocated memory
free(mat);
free(matrix);
...
```

This code allocates `rows*cols` numerical (integer) values contiguous in memory with the initial call to `malloc`, keeping the address in variable `matrix`. We can access to the elements in `matrix` using 1 index computed as explained in section 1.5. In addition, a second array `mat` is allocated. It consists of `rows` pointers to integers which are then initialized to point to the beginning of each row using pointer arithmetic[5]: `mat[i] = matrix + i*cols;` . This allows us to access the matrix using 2 indices through variable `mat`: `mat[i][j]` is equivalent to `matrix[i*cols + j]`.

[5]Pointer arithmetic is explained in section 1.3.1.