A few ways to create matrices in C

Thomas Ericsson
Chalmers, Mathematics
2004

In Fortran (dense) matrices are stored in the same way in (almost) all programs. This is because the matrix is a builtin type in Fortran and the language has a lot of support for matrix computations. This is not the case in C, and so the are several possible data structures for storing matrices. It is important to pick the proper data structure if the matrix should be passed as an argument to a Fortran routine or used together with the Sunperf library. Another issue is how we would like to access the elements in the matrix. Is it important to be able to write `A[row][col]` or will `*(A + row * n + col)` do?

Here comes a short description of some alternative data structures. Suppose we would like to store the matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

The most obvious way is illustrated by the following short program.

```
#include <stdio.h>
int main()
{
  double A[2][3], elem = 0;
  int    row, col;

  for(row = 0; row < 2; row++)
    for(col = 0; col < 3; col++)
      A[row][col] = ++elem;

  return 0;
}
```

This way to create matrices is rather limited. We would at like to have a more dynamic choice of dimensions. The first step would be something like:

```
#include <stdio.h>
int main()
{
  const int m = 2, n = 3;
  double A[m][n];
    ...
```

Some compilers accept such constructions, but not all. The following is allowed, but a bit clumsy:

```
#include <stdio.h>
#define _M 2
#define _N 3
int main()
{
  const int m = _M, n = _N;
  double A[_M][_N];
    ...
```

Such a matrix can be passed as a parameter to a Fortran program.

Dynamic memory allocation

Some assignments in the course require that tests should be performed for a sequence of matrices of increasing sizes. It is inconvenient having to edit the program, changing the dimensions, recompiling etc. This leads us to dynamic memory allocation. So first a few words about that.

The C-library routines `malloc` and `free` are used to allocate memory and to return it. `stdlib.h` contains the prototypes. In C++ we have `new` and `delete`. Java has garbage collection, so only `new` is necessary. Fortran90 has `allocate` and `deallocate`.
We will concentrate on C from now on. `ptr = malloc(size)` returns a pointer, `ptr`, to a block of data at least `size` bytes suitably aligned for any use. If there is not enough available memory `ptr` will be a null pointer.
`free(ptr)` will return the memory to the application, though not to the system. Memory is returned to the system only upon termination of the application. If `ptr` is a null pointer, no action occurs. It is illegal to free the same memory more than once, to try to use freed memory and to free using a pointer not obtained from `malloc`.

Here is a typical piece of code where we allocate 100 double precision numbers. Note the use of `sizeof` and the check on the pointer value. We then store some values in the memory. The first loops uses pointer arithmetic and the second uses vector notation. Note that `vec` is a pointer and not a vector but it is allowed to mix the notation.
There are differences between vectors and pointers though. If we have the declaration:

```
double *vec, vector[100];
```

`vec` can point to something else but `vector` cannot. We need space for the pointer variable, `vec`, but `vector` itself takes no space,

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

int main()
{
  double *vec;       /* vec is a pointer to double */
  int n = 100, k;

  if( (vec = malloc(n * sizeof(double))) == NULL ) {
    printf("malloc of vec failed.\n");
    exit(1);
  }

  for(k = 0; k < n; k++)
   *(vec + k) = k;        /* pointer notation   */

  for(k = 0; k < n; k++)
   vec[k] = k;            /* vector notation    */

  free(vec);              /* release the memory */

  return 0;
}
```

What I would like to do is to allocate memory for an $m \times n$-matrix `A`, using `malloc`, and then pass `A` as an argument to a function, receiving `A` as an $m \times n$-matrix so that I can use matrix-indexing `A[row][col]` inside the function. This can be done with some trickery (and with some compilers), but I do not know how to do it in a completely legal way (following the C-standard) so I will not pursue this topic further.

Here is another alternative. We know that a matrix is stored by rows in C. So if `A` is the address of the `[0][0]`-element, `A[row][col]` has address `A + n * row + col` where `n` is the number of elements in a row. We can use vector indexing instead of using pointer arithmetic. Here is an example (to make the code shorter I will not check that `malloc` succeeded, a bad programming practice). I have added a function to show how the parameter could be passed.

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

double sum_elements(double *A, int m, int n);

int main()
{
  double *A;
  int m = 2, n = 3, k;

  A = malloc(m * n * sizeof(double)); /* Allocate memory for the m x n-matrix
                                       */

  for(k = 0; k < m * n; k++)
    A[k] = k + 1;                     /* This is ONE way to access the elements
                                       */

  printf("result = %e\n", sum_elements(A, m, n));

  free(A);
  return 0;
}

double sum_elements(double *A, int m, int n)
{
  double sum = 0;
  int row, col;

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      sum += A[n * row + col];        /* This simulates A[row][col]-access.
                                         We could use pointer notation.
                                       */
  return sum;
}
```

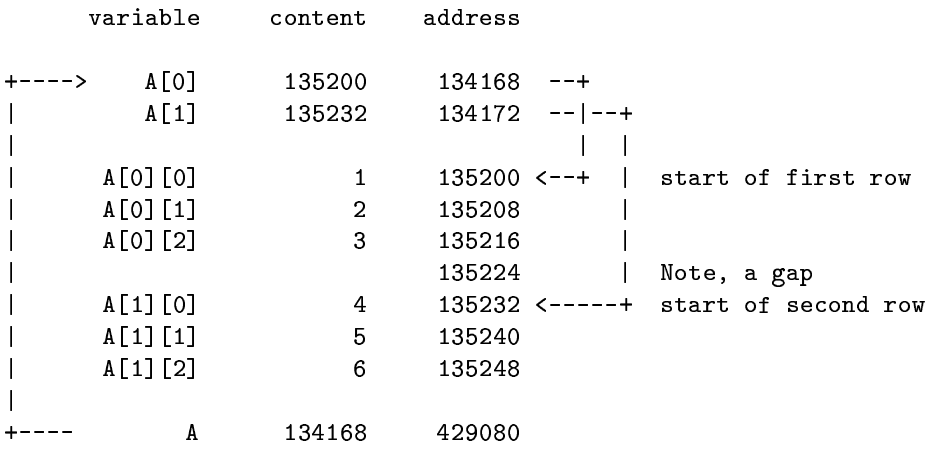
One advantage of this approach is that it easy to pass the array as an argument to a Fortran routine (and it is easy to store the matrix by columns instead).

This will not work with Fortran

Two fairly common ways to store a matrix will be described below. The first method does **not work** together with Fortran though, but the other does. Both methods support `A[row][col]`-indexing.
Here comes the first example:

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

double sum_elements(double **A, int m, int n);

int main()
{
  double **A, elem = 0;                    /* Note **  */
  int m = 2, n = 3, row, col;

  A = malloc(m * sizeof(double *));        /* Allocate space for row pointers.
                                              Note double * .
                                           */

  for(row = 0; row < m; row++)
    A[row] = malloc(n * sizeof(double));   /* Allocate space for elements in a row.
                                              Note double.
                                           */

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      A[row][col] = ++elem;                /* Note A[row][col] */

  printf("result = %e\n", sum_elements(A, m, n));

  for(row = 0; row < m; row++)             /* free */
    free(A[row]);

  free(A);                                 /* free again,
                                              Note the order of the calls to free.
                                           */

  return 0;
}

double sum_elements(double **A, int m, int n)
{
  double sum = 0;
  int row, col;

  for(row = 0; row < m; row++)
    for(col = 0; col < n; col++)
      sum += A[row][col];

  return sum;
}
```

The memory layout may look something like this after we have initialised the matrix. The arrows show how the addresses point.



`A` points to `A[0]` which in turn points to `A[0][0]`, the first element in the first row. `A[1]` points to the beginning of the second row, i.e. `A[1][0]`. The first `malloc` allocates space for `A[0]` and `A[1]` (`m` row pointers). Then comes a loop with `m` calls to `malloc` where each one allocates memory for storing the `n` elements in row.

We note that `sizeof(double *)` is four since `A[0]` and `A[1]` are four bytes apart (134172-134168=4). The double precision numbers are eight bytes apart, except between `A[0][2]` and `A[1][0]` where the gap is 16 bytes. This is the reason this data structure cannot be used when calling Fortran routines, the elements are **not** contiguous in memory.
This storage cannot be used when calling routines in the Sunperf library either (for the same reason).
One advantage with this data structure is that all the rows need not have the same length.
Note also that this storage requires more memory than the usual matrix data structure (we need extra space for the row pointers). That is true with the next method as well, but it has the advantage of giving contiguous elements, making it possible to pass the array to a Fortran routine.

This will work with Fortran

```
...
 double **A;

 A = malloc(m * sizeof(double *));       /* Allocate space for row pointers.
                                            Note double * .
                                         */

 A[0] = malloc(m * n * sizeof(double));  /* Allocate space for the elements in the matrix.
                                            Note that we get contiguous elements.
                                         */
 for(row = 1; row < m; row++)
   A[row] = A[0] + row * n;              /* Give the row pointers their values, i.e.
                                            find out where each row starts.
                                            There are n elements in each row.
                                         */

 for(row = 0; row < m; row++)
   for(col = 0; col < n; col++)
     A[row][col] = ++elem;
...
```

The memory layout may look something like this after we have initialised the matrix. The arrows show how the addresses point.

```
         variable     content    address

   +---->    A[0]      135768     134744  --+
   |         A[1]      135792     134748  --|--+
   |                                        |  |
   |      A[0][0]           1     135768 <--+  |  start of first row
   |      A[0][1]           2     135776       |
   |      A[0][2]           3     135784       |
   |      A[1][0]           4     135792 <-----+  start of second row
   |      A[1][1]           5     135800
   |      A[1][2]           6     135808
   |
   +----        A      134744     429080
```

To pass the array to Fortran we use the parameter `&A[0][0]`, `A[0]` or `*A`.

For more details about this and other topics, see the C-FAQ:
`http://www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c.html`