Arrays and pointers in C
==========

(Thanks a lot to Steve Summit for the very illuminating comments,
thanks to Ilana Zommer for the excellent comments)

This is a short text on arrays and pointers in C with an emphasis on using multi-dimensional arrays. The seemingly unrelated C rules are explained as an attempt to unify arrays and pointers, replacing arrays and the basic array equation by a new notation and special rules (see below).

The purpose of the following text is threefold:

1) Help those who mix C modules in their programs to understand the pointer notation of C and pass arrays between FORTRAN and C.

2) Show possible workarounds for the lack of adjustable arrays, one of C technical shortcomings.

3) Show an interesting method of array implementation.

The C language as a "portable assembler"

Older operating systems and other system software were written in assembly language. CPUs were slow and memories very small, and only assembly language could generate the tight code needed.

However, assembly is difficult to maintain and by definition not portable, so the advantages of a High Level Language designed for system programming were clear. Improvements in hardware and compiler technology made C a success.

Pointers and pointer operators

FORTRAN imposes a major restriction on the programmer, you can reference only named memory locations, i.e. Fortran variables. Pointers make it possible, like in assembly, to reference in a useful way any memory location.

A pointer is a variable suitable for keeping memory addresses of other variables, the values you assign to a pointer are memory addresses of other variables (or other pointers).

How useful are pointers for scientific programming? Probably much less than C fans think, few algorithms used in scientific require pointers. It is well-known that having unrestricted pointers in a programming language makes it difficult for the compiler to generate efficient code.

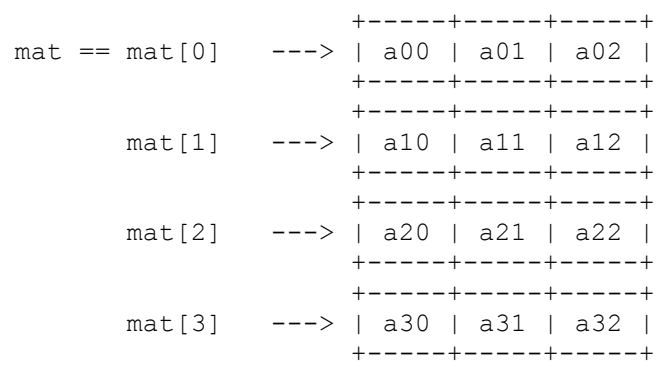
C pointers are characterized by their value and data-type. The value is the address of the memory location the pointer points to, the type determines how the pointer will be incremented/decremented in pointer (or subscript) arithmetic (see below).

Arrays in C and the array equation

We will use 2D arrays in the following text instead of general N-dimensional arrays, they can illustrate the subtle points involved with using arrays and pointers in C, and the arithmetic will be more manageable.

A 2D array in C is treated as a 1D array whose elements are 1D arrays (the rows).

For example, a 4x3 array of T (where "T" is some data type) may be declared by: "T mat[4][3]", and described by the following scheme:



The array elements are stored in memory row after row, so the array equation for element "mat[m][n]" of type T is:

```
address(mat[i][j]) = address(mat[0][0]) + (i * n + j) * size(T)

address(mat[i][j]) = address(mat[0][0]) +
                     i * n * size(T)    +
                     j * size(T)

address(mat[i][j]) = address(mat[0][0]) +
                     i * size(row of T) +
                     j * size(T)
```

A few remarks:

1) The array equation is important, it is the connection between the abstract data-type and its implementation. In Fortran (and other languages) it is "hidden" from the programmer, the compiler automatically "plants" the necessary code whenever an array reference is made.

2) For higher-dimensional arrays the equation gets more and more complicated. In some programming languages an arbitrary limit on the dimension is imposed, e.g. Fortran arrays can be 7D at most.

3) Note that it's more efficient to compute the array equation "iteratively" - not using the distributive law to eliminate the parentheses (just count the arithmetical operations in the first two versions of the array equation above). The K&R method (see below) works iteratively.

   It reminds one of Horner's Rule for computing a polynomial iterativly, e.g.

   ```
   a * x**2 + b * x + c = (a * x + b) * x + c
   ```

computing the powers of x is eliminated in this way.

4) The number of rows doesn't enter into the array equation, you don't need it to compute the address of an element. That is the reason you don't have to specify the first dimension in a routine that is being passed a 2D array, just like in Fortran's assumed-size arrays.

The K&R method of reducing arrays to pointers

K&R tried to create a unified treatment of arrays and pointers, one that would expose rather than hide the array equation in the compiler's code. They found an elegant solution, albeit a bit complicated. The "ugly" array equation is replaced in their formulation by four rules:

1) An array of dimension N is a 1D array with elements that are arrays of dimension N-1.

2) Pointer addition is defined by:

```
ptr # n = ptr + n * size(type-pointed-into)
```

"#" denotes here pointer addition to avoid confusion with ordinary addition.
The function "size()" returns object's sizes.

3) The famous "decay convention": an array is treated as a pointer that points to the first element of the array.

The decay convention shouldn't be applied more than once to the same object.

4) Taking a subscript with value i is equivalent to the operation: "pointer-add i and then type-dereference the sum", i.e.

```
xxx[i] = *(xxx # i)
```

When rule #4 + rule #3 are applied recursively (this is the case of a multi-dimensional array), only the data type is dereferenced and not the pointer's value, except on the last step.

K&R rules imply the array equation

We will show now that the array equation is a consequence of the above rules (applied recursively) in the case of a 2D array:

```
mat[i] = *(mat # i)                    (rule 4)

mat[i][j] = *(*(mat # i) # j)          (rule 4)
```

"mat" is clearly a "2D array of T" and decays by rule #3 into a "pointer to a row of T". So we get the first two terms of the array equation.

```
   mat[i][j] = *(*(mat + i * sizeof(row)) # j)
                  ^^^^^^^^^^^^^^^^^^^^^
                   Pointer to row of T
```

Dereferencing the type of "(mat # i)" we get a "row of T".

```
   mat[i][j] = *((mat + i * sizeof(row)) # j)
                 ^^^^^^^^^^^^^^^^^^^^^^
                       Row of T
```

We have now one pointer addition left, using again the "decay convention",
the 1D array "row of T" becomes a pointer to its first element, i.e.
"pointer to T". We perform the pointer addition, and get the third term
of the array equation:

```
   mat[i][j] = *(mat + i * sizeof(row) + j * sizeof(T))
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                            Pointer to T

   address(mat[i][j]) = mat + i * sizeof(row) + j * sizeof(T)
```

Remember that "mat" actually points to the first element of the array,
so we can write:

```
   address(mat[i][j]) = address(mat[0][0]) +
                        i * sizeof(row)    +
                        j * sizeof(T)
```

This is exactly the array equation. QED

Why a double pointer can't be used as a 2D array?

This is a good example, although the compiler may not complain,
it is wrong to declare: "int **mat" and then use "mat" as a 2D array.
These are two very different data-types and using them you access
different locations in memory. On a good machine (e.g. VAX/VMS) this
mistake aborts the program with a "memory access violation" error.

This mistake is common because it is easy to forget that the decay
convention mustn't be applied recursively (more than once) to the
same array, so a 2D array is NOT equivalent to a double pointer.

A "pointer to pointer of T" can't serve as a "2D array of T".
The 2D array is "equivalent" to a "pointer to row of T", and this
is very different from "pointer to pointer of T".

When a double pointer that points to the first element of an array,
is used with subscript notation "ptr[0][0]", it is fully dereferenced
two times (see rule #5). After two full dereferencings the resulting
object will have an address equal to whatever value was found INSIDE
the first element of the array. Since the first element contains
our data, we would have wild memory accesses.

We could take care of the extra dereferencing by having an intermediary
"pointer to T":

```
   type   mat[m][n], *ptr1, **ptr2;

   ptr2 = &ptr1;
```

```
   ptr1 = (type *)mat;
```

but that wouldn't work either, the information on the array "width" (n), is lost, and we would get right only the first row, then we will have again wild memory accesses.

A possible way to make a double pointer work with a 2D array notation is having an auxiliary array of pointers, each of them points to a row of the original matrix.

```
   type   mat[m][n], *aux[m], **ptr2;

   ptr2 = (type **)aux;
   for (i = 0 ; i < m ; i++)
       aux[i] = (type *)mat + i * n;
```

Of course the auxiliary array could be dynamic.

An example program:

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

main()
        {
        long    mat[5][5], **ptr;

        mat[0][0] = 3;
        ptr = (long **)mat;

        printf("  mat          %p \n",  mat);
        printf("  ptr          %p \n",  ptr);
        printf("  mat[0][0]    %d \n",  mat[0][0]);
        printf(" &mat[0][0]    %p \n", &mat[0][0]);
        printf(" &ptr[0][0]    %p \n", &ptr[0][0]);

        return;
        }
```

The output on VAX/VMS is:

```
         mat          7FDF6310
         ptr          7FDF6310
         mat[0][0]    3
        &mat[0][0]    7FDF6310
        &ptr[0][0]    3
```

We can see that "mat[0][0]" and "ptr[0][0]" are different objects (they have different addresses), although "mat" and "ptr" have the same value.

What methods for passing a 2D array to a subroutine are allowed?

Following are 5 alternative ways to handle in C an array passed from a Fortran procedure or another c routine.

Various ways to declare and use such an array are presented by examples with an array made of 3x3 shorts (INTEGER*2). All 5 methods work on

a VAX/VMS machine with DECC.

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

int func1();
int func2();
int func3();
int func4();
int func5();

main()
{
        short mat[3][3],i,j;

        for(i = 0 ; i < 3 ; i++)
                for(j = 0 ; j < 3 ; j++)
                {
                        mat[i][j] = i*10 + j;
                }

        printf(" Initialized data to: ");
        for(i = 0 ; i < 3 ; i++)
        {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                        printf("%5.2d", mat[i][j]);
                }
        }
        printf("\n");

        func1(mat);
        func2(mat);
        func3(mat);
        func4(mat);
        func5(mat);
}

 /*
 Method #1 (No tricks, just an array with empty first dimension)
 ================================================================
 You don't have to specify the first dimension!
 */

int func1(short mat[][3])
{
        register short i, j;

        printf(" Declare as matrix, explicitly specify second dimension: ");
        for(i = 0 ; i < 3 ; i++)
                {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                        printf("%5.2d", mat[i][j]);
                }
        }
        printf("\n");

        return;
```

```
}

 /*
 Method #2 (pointer to array, second dimension is explicitly specified)
 ======================================================================
 */

int func2(short (*mat)[3])
        {
        register short i, j;

        printf(" Declare as pointer to column, explicitly specify 2nd dim: ");
        for(i = 0 ; i < 3 ; i++)
                {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                        printf("%5.2d", mat[i][j]);
                }
        }
        printf("\n");

        return;
}

 /*
 Method #3 (Using a single pointer, the array is "flattened")
 ============================================================
 With this method you can create general-purpose routines.
 The dimensions doesn't appear in any declaration, so you
 can add them to the formal argument list.

 The manual array indexing will probably slow down execution.
 */

int func3(short *mat)
        {
        register short i, j;

        printf(" Declare as single-pointer, manual offset computation: ");
        for(i = 0 ; i < 3 ; i++)
                {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                        printf("%5.2d", *(mat + 3*i + j));
                }
        }
        printf("\n");

        return;
}

 /*
 Method #4 (double pointer, using an auxiliary array of pointers)
 ================================================================
 With this method you can create general-purpose routines,
 if you allocate "index" at run-time.

 Add the dimensions to the formal argument list.
 */
```

```
int func4(short **mat)
        {
        short    i, j, *index[3];

        for (i = 0 ; i < 3 ; i++)
                index[i] = (short *)mat + 3*i;

        printf(" Declare as double-pointer, use auxiliary pointer array: ");
        for(i = 0 ; i < 3 ; i++)
                {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                        printf("%5.2d", index[i][j]);
                }
        }
        printf("\n");

        return;
}

 /*
 Method #5 (single pointer, using an auxiliary array of pointers)
 ================================================================
 */

int func5(short *mat[3])
        {
        short i, j, *index[3];
        for (i = 0 ; i < 3 ; i++)
                index[i] = (short *)mat + 3*i;

        printf(" Declare as single-pointer, use auxiliary pointer array: ");
        for(i = 0 ; i < 3 ; i++)
                {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                        printf("%5.2d", index[i][j]);
                }
        }
        printf("\n");
        return;
}
```