P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #9

(Static) Arrays & Strings

static arrays, bracket operator, memory mapping, strings, '\0'

Why Arrays?

- so far we were concerned with variables for storing single data items
- for numbers this corresponds to scalar quantities
- mathematics also deals with higher dimensional quantities like e.g. vectors or matrices
- would be cumbersome to store each sampled value in a recorded seismogram in a single variable

$\Longrightarrow$ there is a need for a data structure that allows to treat collections of related data items as a single entity

What is an Array?

Characteristics

- data structure consisting of a group of elements
- individual elements are accessed by indexing
- an index is an integral number
- all elements are of the same intrinsic data type
- array elements are stored in a contiguous memory area

For a static array its size is known at compile time of the program, while for a dynamic array it is set during runtime.

Static Arrays in C (1/2)

- Static arrays in C are defined in the following fashion:

```
int iVec[10];    // array of 10 ints
double dVec[5];  // array of 5 doubles

typedef struct { char red; char green; char blue; } rgbValue;
rgbValue colors[128];  // array of 128 color definitions
```

- The number of elements can also be specified differently, as long as it is known at compile time:

Static Arrays in C (1/2)

- Static arrays in C are defined in the following fashion:

```
int iVec[10];    // array of 10 ints
double dVec[5];  // array of 5 doubles

typedef struct { char red; char green; char blue; } rgbValue;
rgbValue colors[128];  // array of 128 color definitions
```

- The number of elements can also be specified differently, as long as it is known at compile time:

```
const int dim = 100;  // named constant
double v[dim];

int size = 100;
double z[size];       // uses current value of size
```

Static Arrays in C (2/2)

- Arrays can be initialised as part of their definition:

```
int arr[3] = { -2, 3, 17 };
float vec[3] = { 1.0f, -3.7f, 5.1f };
```

- We can let the compiler determine array size form the length of the initialiser list:

```
double xx[] = { 1.0, -5.7, 9.8 };
```

- If list is too short, remaining entries are set to zero:

```
short s[5] = { 1, 2, 4 };
```

Static Arrays in C (3/3)

- Individual array elements are accessed using the bracket operator []

```
int arr[3] = { -2, 3, 17 }, elem;
arr[0] = 1;        // write access
elem = arr[2];     // read access
```

- Indexing of static arrays in C always starts at 0.
- The permissible index range is $[0, \ldots, \text{length} - 1]$.
- Accessing an array with other indices is called out-of-bounds access and can lead to
  - hard to detect errors
  - data corruption
  - program crash (segmentation violation)

Example: Inner Product

The following codelet computes the inner product of two vectors a and $b \in \mathbb{R}^{100}$

```
const int dim = 100;

double a[dim];
double b[dim];
double innerProduct;

/* fill vectors a and b with values */

innerProduct = 0.0;
for( int k = 0; k < dim; k++ ) {
  innerProduct += a[k] * b[k];
}
```

Multi-dimensional Arrays (1/2)

- C allows to define multi-dimensional arrays, e.g.

```
int aMat[20][30];
double tn3D[20][15][100];
```

- The number of dimensions is called the rank of the array.
- The size of the (complete) array is the product of the extent in the individual dimensions

$$\#\text{ elements of } \texttt{aMat} = 20 \cdot 30 = 600$$

$$\#\text{ elements of } \texttt{tn3D} = 20 \cdot 15 \cdot 100 = 30\,000$$

- Its memory footprint is the product of its size and the memory needed for datatype of the elements:

```
printf( " memory footprint = %lu bytes\n", sizeof( tn3D ) );
// memory footprint = 240000 bytes
```

Multi-dimensional Arrays (2/2)

- To access a single element of a rank n array we need an n-tuple of indices $(i_1, i_2, \ldots, i_n)$:

```
int mat[10][10];
mat[5][3] = 1.0;
```

- Internally we will need a mapping that assigns to each element a 1D address (to store it in computer memory).

Example: Matrix-Vector-Product (1/2)

Let $A \in \mathbb{R}^{n \times n}$ be a matrix and $x \in \mathbb{R}^n$ a vector. From linear algebra we know that the matrix-vector-product $y = Ax$ is given by

$$Ax = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{n} a_{1,j}\, x_j \\ \vdots \\ \sum_{j=1}^{n} a_{n,j}\, x_j \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = y$$

Example: Matrix-Vector-Product (2/2)

Let A, x and y be represented by

```
double x[n], y[n], A[n][n];
```

then the matrix-vector product $y = Ax$ may be computed by

```
for( int i = 0; i < n; i++ ) {
   y[i] = 0.0;
   for( int j = 0; j < n; j++ ) {
      y[i] += A[i][j] * x[j];
   }
}
```

Performance Test (1/2)

- assume we want to compute $A = B + C$ with matrices $A, B, C \in \mathbb{R}^{n \times n}$
- we compare two versions

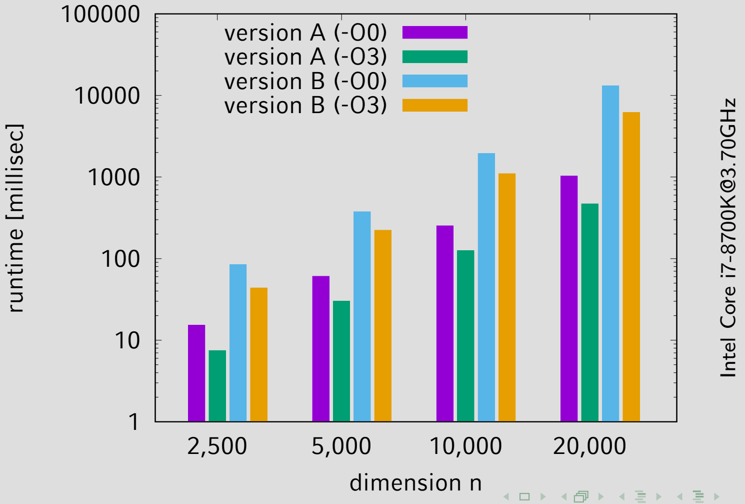
Variant A

```
for( int i = 0; i < dim; i++ )
  for( int j = 0; j < dim; j++ )
    matA[i][j] = matB[i][j] + matC[i][j];
```

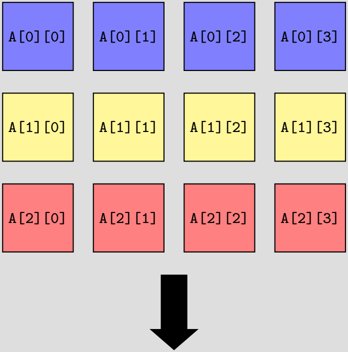
Variant B

```
for( int j = 0; j < dim; j++ )
  for( int i = 0; i < dim; i++ )
    matA[i][j] = matB[i][j] + matC[i][j];
```

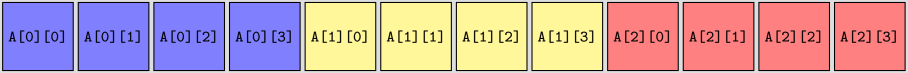
Performance Test (2/2)



C Array Layout



- C uses row-major ordering
- a 2D array (matrix) is stored one row after the other
- general: if two array elements differ in the last index only, they will be next to each other in memory
- right most indices run fastest, when traversing the array in memory



Fortran Array Layout

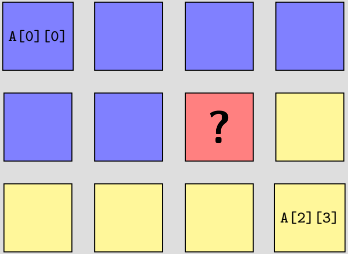
A(1,1)	A(1,2)	A(1,3)	A(1,4)
A(2,1)	A(2,2)	A(2,3)	A(2,4)
A(3,1)	A(3,2)	A(3,3)	A(3,4)

$\Downarrow$

A(1,1)	A(2,1)	A(3,1)	A(1,2)	A(2,2)	A(3,2)	A(1,3)	A(2,3)	A(3,3)	A(1,4)	A(2,4)	A(3,4)

- Fortran uses column-major ordering
- a 2D array (matrix) is stored one column after the other
- in general: if two array elements differ in the first index only, they will be next to each other in memory
- the left most indices run fastest, when traversing the array in memory

Mapping Function (1/3)



Example:

consider the 2D array on the left

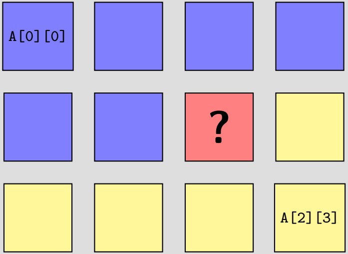
Question:

if memory locations are indexed $0, 1, 2, \ldots, 11$, at which position is the marked element stored?

Setting $s_2 = 4$ (extent in 2nd dimension), we get as memory position

$$\mathcal{M}\left(\texttt{A[1][2]}\right) = 1 \cdot s_2 + 2 = 6$$

Mapping Function (1/3)



Example:

consider the 2D array on the left

Question:

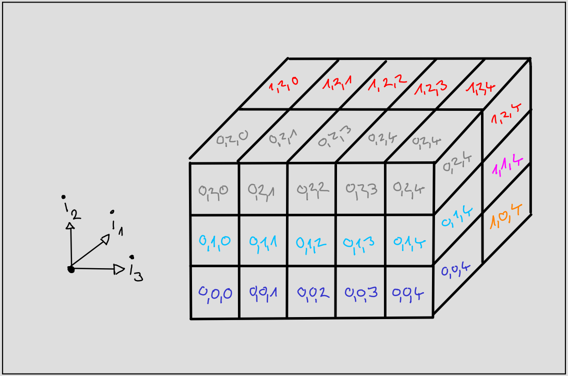
if memory locations are indexed $0, 1, 2, \ldots, 11$, at which position is the marked element stored?

Setting $s_2 = 4$ (extent in 2nd dimension), we get as memory position

$$\mathcal{M}\left(\texttt{A[1][2]}\right) = 1 \cdot s_2 + 2 = 6$$

general case: $\mathcal{M}\left(\texttt{A[i][j]}\right) = i \cdot s_2 + j$

Mapping Function (2/3)



For a 3D array we will get:

$$\mathcal{M}\left(\texttt{A[i][j][k]}\right) = i \cdot s_2 s_3 + j \cdot s_3 + k$$

Mapping Function (3/3)

- assume that we have an n–dimensional array and
- $i_1, i_2, \ldots, i_n$ are indices of an array element
- s_m is the extent in the m-th dimension

then $\mathcal{M}(i_1, i_2, \ldots, i_n)$ is given by

$$i_1 s_2 \cdots s_n + i_2 s_3 \cdots s_n + \ldots + i_{n-1} s_n + i_n$$

or concisely

$$\boxed{\mathcal{M}(i_1, i_2, \ldots, i_n) = \sum_{k=1}^{n} i_k \left(\prod_{m=k+1}^{n} s_m \right)}$$

Representation of Strings (1/3)

- String literals are enclosed in quotation marks.
- Strings are arrays of characters.

```
char mesg[] = "Hello World!";

printf( "length(mesg) = %lu characters\n", strlen(mesg) );
printf( "sizeof(mesg) = %lu bytes\n", sizeof(mesg) );
```

Representation of Strings (1/3)

- String literals are enclosed in quotation marks.
- Strings are arrays of characters.

```
char mesg[] = "Hello World!";

printf( "length(mesg) = %lu characters\n", strlen(mesg) );
printf( "sizeof(mesg) = %lu bytes\n", sizeof(mesg) );
```

- The code above prints

```
length(mesg) = 12 characters
sizeof(mesg) = 13 bytes
```

Representation of Strings (2/3)

- C use an extra null character ('\0') to mark the end of a string inside the character array.
- The following codelet prints strings do match!

```
char sayHi[] = "Hello";
char greet[] = { 'H', 'e', 'l', 'l', 'o', '\0' } ;

if( strcmp( greet, sayHi ) == 0 )
  printf( "strings do match!\n" );
```

- Operations on strings need to make sure there is enough space for the '\0' and that it is preserved.

Representation of Strings (3/3)

- We can directly access/manipulate single characters of a string

```
char str[] = "steamboat";

for( int k = 0; k < strlen( str ); k++ ) {
  printf( "%c", str[k] );
}

str[5] = '\0';
printf( "s use %s\n", str );
```

Representation of Strings (3/3)

- We can directly access/manipulate single characters of a string

```
char str[] = "steamboat";

for( int k = 0; k < strlen( str ); k++ ) {
  printf( "%c", str[k] );
}

str[5] = '\0';
printf( "s use %s\n", str );
```

- The code above prints

```
steamboats use steam
```

String Functions

- C does not allow us to directly operate on a string as a whole
  (such as compare two strings, assign one string variable to another, . . .)
- Instead functions are provided for this purpose
  (use requires to include `string.h` header file)
- The most important ones are

`strcpy()`	copy strings
`strncpy()`	copy n bytes of string
`strcat()`	concatenate strings
`strcmp()`	compare strings
`strlen()`	get length of string

String Copying (1/2)

- strcpy() copies complete string (including \0) in source buffer to destination buffer:

```
char strA[] = "Donaudampfschiff";
char strB[ strlen(strA) + 1];

strcpy( strB, strA );
printf( "copied string: '%s'\n", strB );
```

- So we get

```
copied string:  'Donaudampfschiff'
```

- Important: Destination buffer must be large enough!

String Copying (2/2)

- `strncpy()` copies maximally the first n bytes from source buffer to destination buffer:

```
char srcBuf[] = "abc";
char dstBuf[] = "123456789";
strncpy( dstBuf, srcBuf, 3 );
printf( "dstBuf: '%s'\n", dstBuf );
strncpy( dstBuf, srcBuf, 6 );
printf( "dstBuf: '%s'\n", dstBuf );
```

- details depend on n and length s of the string in the source buffer

$n \leqslant s$	copies first n characters only
$n = s + 1$	copies string including \0
$n > s + 1$	copies and adds $[n - (s + 1)]$ null chars '\0'

String Copying (2/2)

- `strncpy()` copies maximally the first *n* bytes from source buffer to destination buffer:

```
char srcBuf[] = "abc";
char dstBuf[] = "123456789";
strncpy( dstBuf, srcBuf, 3 );
printf( "dstBuf: '%s'\n", dstBuf );
strncpy( dstBuf, srcBuf, 6 );
printf( "dstBuf: '%s'\n", dstBuf );
```

- we get

```
dstBuf: 'abc456789'
dstBuf: 'abc'          <-- [ abc\0\0\0789\0 ]
```

Concatenation

`strcat()` can be used to append one string to another:

```
char w1[] = "dampfschiff";
char w2[] = "fahrtsgesellschaft";
char longWord[1000] = "Donau";
printf( "%s\n", longWord );
strcat( longWord, w1 );
printf( "%s\n", longWord );
strcat( longWord, w2 );
printf( "%s\n", longWord );
strcat( longWord, "skapitänstochter" );
```

```
Donau
Donaudampfschiff
Donaudampfschifffahrtsgesellschaft
Donaudampfschifffahrtsgesellschaftskapitänstochter
```

Safety (1/2)

- None of the functions
  - strcpy()
  - strncpy()
  - strcat()

  checks whether there is enough space in the destination buffer!
- If too many characters are copied one can overwrite the following memory area!

Safety (2/2)

- This codelet

```
struct { char dst[2]; char iVal; } test = { "A", 3 };
char src[] = "BC";
printf( "iVal (before) = %d\n", test.iVal );
strcat( test.dst, src );
printf( "iVal (after) = %d\n", test.iVal );
```

- will output

```
iVal (before) = 3
iVal (after) = 67
```

Comparing Strings

- strcmp() can be used to compare two strings
- if `s1` is the first argument and `s2` the second, it will return an int value:

< 0	if `s1` is lexicographically smaller than `s2`
$= 0$	of both strings are lexicographically equal
> 0	if `s1` is lexicographically larger than `s2`

- example:

```
strcmp( "abcde", "abCde" )   --> might return +1
strcmp( "abcde", "abcde" )   --> will return 0
strcmp( "abcd" , "abcde" )   --> might return -1
```

Length of String

strlen() returns the length of a string (in characters) without the terminating \0.

Remark

We will be able to do more sophisticated string manipulation once we understand the concept of pointers.