Array Representation

Troels Henriksen

Arrays

Definition

An array is a multidimensional sequence of objects of the *same type and size*.

- Arrays often used to represent mathematical objects such as *vectors*, *matrices*, and *tensors*.
- Probably the most common data structure for scientific data.
- The arrays we will cover in this lab (and course) are
  - **Regular:** all “rows” of a multi-dimensional array have the same size.
  - **Homogeneous:** all elements have the same type.

Regular arrays

In Python and F#, we can have lists of lists with irregular shapes:

```
>>> a = [[1,2,3], [4]]
>>> len(a)
2
>>> len(a[0])
3
>>> len(a[1])
1
```

- Such structures are **irregular**, and outside today's topic.
- What we will discuss is more similar to NumPy arrays.

So what's wrong with C arrays

We can declare an $n \times m$ array as

```
double A[n][m];
```

And then we can index it with for example `A[1][2]`. Easy!

So what's wrong with C arrays

We can declare an $n \times m$ array as

```
double A[n][m];
```

And then we can index it with for example `A[1][2]`. Easy!

But there are many problems with built-in arrays:

- They decay to pointers in many situations.
- They cannot be passed to a function without losing their size.
- They cannot be returned from a function at all.
- **They are not values!**

Let's build our own arrays

- C as a *language* does not have useful dynamic arrays.
- But C does have useful support for *dynamic memory allocation*.
- So let's build our own arrays!

Constructing a dynamic array in C

- Use `malloc()` or `calloc()` to obtain a block of memory with room for enough *bytes* to fit the array we need.
- We can view these functions as allocating an "array of bytes", which we can then *interpret* as arrays of some other type.

Let’s build our own arrays

- C as a *language* does not have useful dynamic arrays.
- But C does have useful support for *dynamic memory allocation*.
- So let’s build our own arrays!

Constructing a dynamic array in C

- Use `malloc()` or `calloc()` to obtain a block of memory with room for enough *bytes* to fit the array we need.
- We can view these functions as allocating an “array of bytes”, which we can then *interpret* as arrays of some other type.

Questions:

- **How much memory do we allocate?** An *x*-element array needs *x*`*sizeof(t)` bytes, where `t` is the element type (`int`, `double`, etc).

Let’s build our own arrays

- C as a *language* does not have useful dynamic arrays.
- But C does have useful support for *dynamic memory allocation*.
- So let’s build our own arrays!

Constructing a dynamic array in C

- Use `malloc()` or `calloc()` to obtain a block of memory with room for enough *bytes* to fit the array we need.
- We can view these functions as allocating an “array of bytes”, which we can then *interpret* as arrays of some other type.

Questions:

- **How much memory do we allocate?** An *x*-element array needs *x*`*sizeof(t)` bytes, where `t` is the element type (`int`, `double`, etc).
- **How do we lay out the array in memory?** That’s a more open question...

The idea

```
int* arr = malloc(12); // reserve 12 bytes
```

- Suppose `malloc()` returns the address 1000.
- When we do `arr[i]`, C computes the address $1000 + \texttt{i} \times \texttt{sizeof(int)}$ and reads an `int` (four bytes) from that address.
  - `&arr[0]`: 1000
  - `&arr[1]`: 1004
  - `&arr[2]`: 1008
- (Recall that `&x` means "the address of `x`".)

One-dimensional arrays in C

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

int main() {
  int size = 10;
  int *arr = malloc(size * sizeof(int));

  printf("&arr: %p\n", (void*)&arr);
  printf("arr:  %p\n", (void*)arr);

  for (int i = 0; i < size; i++) {
    arr[i] = i*2;
    printf("&arr[%d]: %p ", i, (void*)&arr[i]);
    printf("arr[%d]: %d\n", i, arr[i]);
  }

  free(arr);
}
```

```
$ gcc 1darray.c -o 1darray
$ ./1darray
&arr: 0x7ffee169ba80
arr:  0x1bb42a0
&arr[0]: 0x1bb42a0 arr[0]: 0
&arr[1]: 0x1bb42a4 arr[1]: 2
&arr[2]: 0x1bb42a8 arr[2]: 4
&arr[3]: 0x1bb42ac arr[3]: 6
&arr[4]: 0x1bb42b0 arr[4]: 8
&arr[5]: 0x1bb42b4 arr[5]: 10
&arr[6]: 0x1bb42b8 arr[6]: 12
&arr[7]: 0x1bb42bc arr[7]: 14
&arr[8]: 0x1bb42c0 arr[8]: 16
&arr[9]: 0x1bb42c4 arr[9]: 18
```

Multi-dimensional arrays

- Machines (and C) provide a *one-dimensional memory (or index) space*.
- When we want multi-dimensional arrays (and we do!) we need to specify a *mapping* between our desired multi-dimensional space and the machine's single-dimensional space.

This is an *index function*.

Index functions

An index function maps a d-dimensional index to a single-dimensional index.

The type of index functions

$$I : \mathbb{N}^d \to \mathbb{N}$$

Index functions are not necessarily literal C functions, but a *conceptual* description of how the array is laid out in memory.

Index functions

An index function maps a d-dimensional index to a single-dimensional index.

The type of index functions

$$I : \mathbb{N}^d \to \mathbb{N}$$

Index functions are not necessarily literal C functions, but a *conceptual* description of how the array is laid out in memory.

Inverse index functions

$$I^{-1} : \mathbb{N} \to \mathbb{N}^d$$

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

How do we lay out this matrix in memory?

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

How do we lay out this matrix in memory?

Row-major order: where elements of each *row* are contiguous in memory:

11	12	13	14	21	22	23	24	31	32	33	34

with index function

$$(i,j) \mapsto i \times 4 + j$$

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

How do we lay out this matrix in memory?

Row-major order: where elements of each *row* are contiguous in memory:

11	12	13	14	21	22	23	24	31	32	33	34

with index function

$$(i, j) \mapsto i \times 4 + j$$

Column-major order: where elements of each *column* are contiguous in memory:

11	21	31	12	22	32	13	23	33	14	24	34

with index function

$$(i, j) \mapsto j \times 3 + i$$

Two-dimensional index functions for $n \times m$ arrays

Row-major indexing

$$(i,j) \mapsto i \times m + j$$

Column-major indexing

$$(i,j) \mapsto j \times n + i$$

Intuition:

- Row-major indexing first *skips* i rows each comprising m elements, then jumps j elements into the row we reach.
- This is why n (the number of rows) is not used for row-major indexing.

Column-major has same intuition, but we skip size-n columns instead.

Two-dimensional arrays in C

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

```c
int *A =
  malloc(3*4*sizeof(int));
A[0] = 11; // first row
A[1] = 12; // first row
A[2] = 13; // first row
A[3] = 14; // first row
A[4] = 21; // second row
...
A[11] = 34;
```

```c
int *A =
  malloc(3*4*sizeof(int));
A[0] = 11; // first col
A[1] = 21; // first col
A[2] = 31; // first col
A[3] = 12; // second col
A[4] = 22; // second col
...
A[11] = 34;
```

Index functions as C functions

```c
int idx2_rowmajor(int n, int m, int i, int j) {
  return i * m + j;
}

int idx2_colmajor(int n, int m, int i, int j) {
  return j * n + i;
}
```

Useful if you get confused when writing index calculations by hand (I often do!)

Careful!

Consider indexing the 3×4 array from before with the expression

```
A[idx2_rowmajor(n, m, 2, 5)].
```

- Trying to access index $(2, 5)$—conceptually out of bounds.
- Index function translates to the flat index $2 \times 3 + 5 = 11$.
- This is *in-bounds* for the 12-element flat array we use to represent our matrix!
- Our program will not crash, but this is probably a bug.

Careful!

Consider indexing the 3×4 array from before with the expression

```
A[idx2_rowmajor(n, m, 2, 5)].
```

- Trying to access index $(2, 5)$—conceptually out of bounds.
- Index function translates to the flat index $2 \times 3 + 5 = 11$.
- This is *in-bounds* for the 12-element flat array we use to represent our matrix!
- Our program will not crash, but this is probably a bug.

```
int idx2_rowmajor(int n, int m, int i, int j) {
  assert(i >= 0 && i < n);
  assert(j >= 0 && j < m);
  return i * m + j;
}
```

Higher dimensions

For a d-dimensional row-major array of shape $n_0 \times \cdots \times n_{d-1}$, the index function where p is a d-dimensional index point is

$$p \mapsto \sum_{0 \leq i < d} p_i \times \prod_{i < j < d} n_j$$

where p_i gets the ith coordinate of p, and the product of an empty series is 1.

Intuition: p_i tells us how many "subarrays" of size $n_{i+1} ... \times ... n_{d-1}$ we need to skip.

Example: four-dimensional indexing

Suppose we have an row-major array of shape

$$n_0 \times n_1 \times n_2 \times n_3$$

and we wish to compute the flat index of element position

$$(p_0, p_1, p_2, p_3)$$

We then have to sum these terms where the *strides* s_i depend on the array size:

$$\begin{array}{rl} & p_0 \times s_0 \\ + & p_1 \times s_1 \\ + & p_2 \times s_2 \\ + & p_3 \times s_3 \end{array}$$

Example: four-dimensional indexing

Suppose we have an row-major array of shape

$$n_0 \times n_1 \times n_2 \times n_3$$

and we wish to compute the flat index of element position

$$(p_0, p_1, p_2, p_3)$$

We then have to sum these terms where the *strides* s_i depend on the array size:

$$\begin{array}{rl} & p_0 \times n_1 \times n_2 \times n_3 \\ + & p_1 \times n_2 \times n_3 \\ + & p_2 \times n_3 \\ + & p_3 \times 1 \end{array}$$

The stride s_i is the product $\prod_{i<j<4} n_j$ of the array size after dropping the first $i+1$ dimensions.

Size passing

Since we represent arrays as the address of their first element, we must manually pass along the size when we call a function with an array.

```c
double sumvec(int n, const double *vector) {
  double sum = 0;
  for (int i = 0; i < n; i++) {
    sum += vector[i];
  }
  return sum;
}
```

As usual: C will not protect us if we pass the wrong size. Be careful.

Slicing

- When using row-major order, the elements of each row are adjacent in memory.
- This allows us to perform efficient *slicing*, by taking the address of the first element in a row.

```
void sumrows(int n, int m,
             const double *matrix, double *vector) {
  for (int i = 0; i < n; i++) {
    vector[i] = sumvec(m, &matrix[i*m]);
  }
}
```

Conclusions

- C's built-in arrays are suitable only for small arrays, typically of static size.
- Dynamic allocation can create single-dimensional dynamic arrays on the heap.
- We can represent multi-dimensional arrays as single-dimensional arrays, by specifying an index function.
- Careful when indexing these home-made arrays—the C language is not much help.