

Array Representation

Philippe Bonnet, bonnet@di.ku.dk
HPPS 2025 – 3b

(with slides from T.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 lecture (and course) are
 - **Regular:** all “rows” of a multi-dimensional array have the same size.
 - **Homogeneous:** all elements have the same type.

More terminology

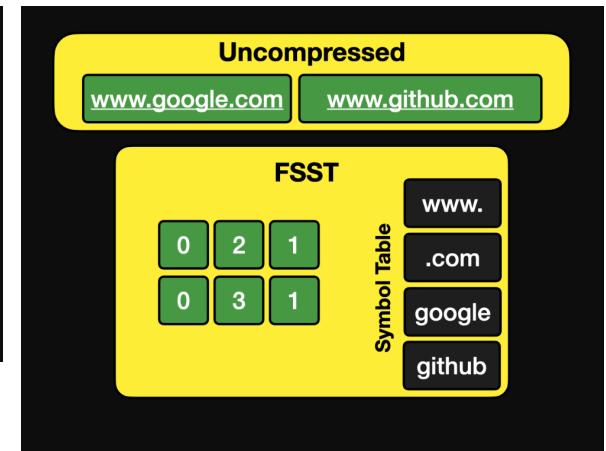
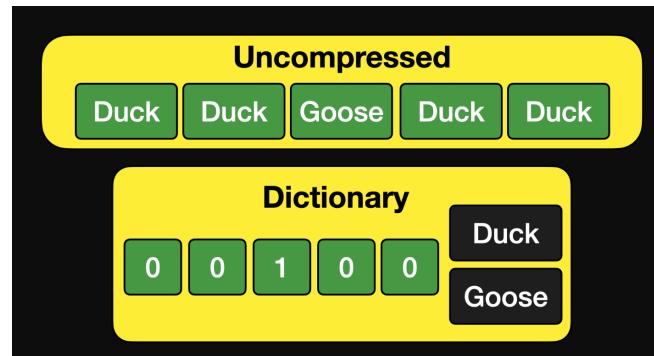
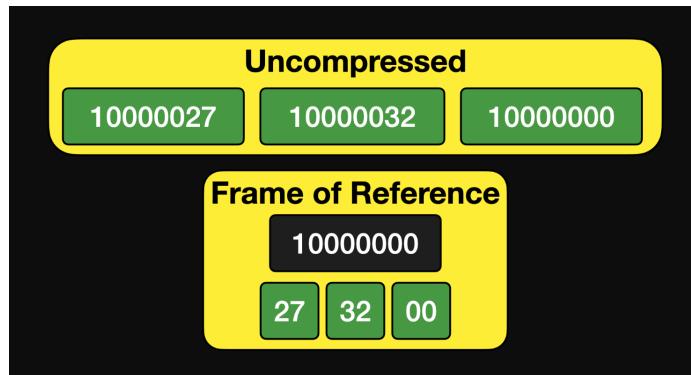
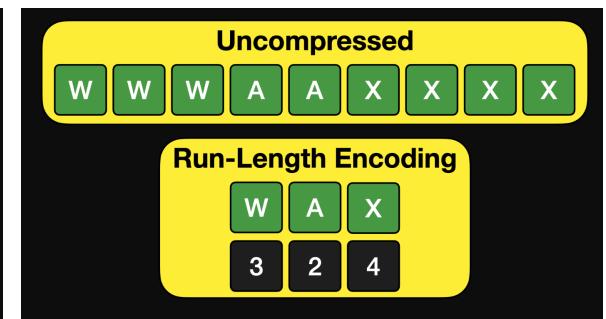
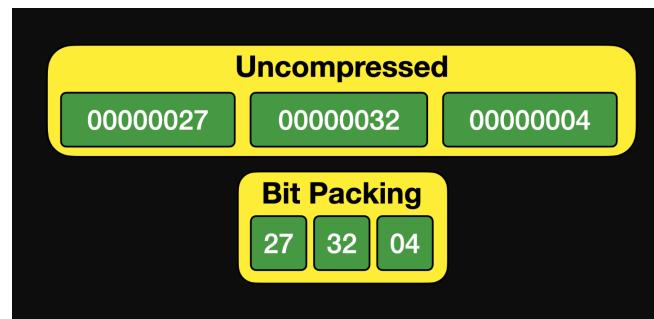
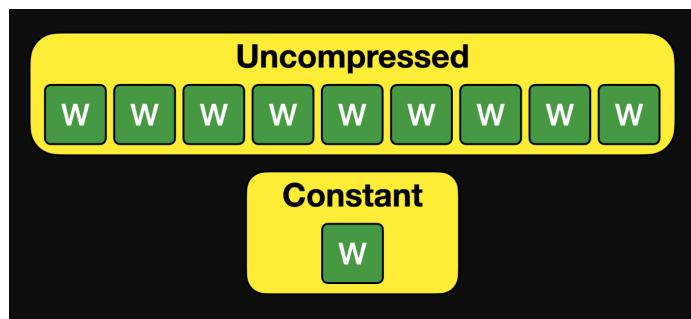
- A **sparse array** is an array in which most of the elements are zero. A **dense array** is not sparse.
- The **rank** of an array is the number of indices required to access a specific element.
 - A vector is represented with an array of rank 1. A matrix is represented with an array of rank 2. A 4-dimensional tensor is represented with an array of rank 4.
- The **dimension** is the size of the array along a given axis.
 - A 10x8 matrix has dimension 10 along the 1st dimension and 8 along the 2nd dimension. Its rank is 2.

Compressed representations (vectors)

Vectors are well-suited to domain specific compression (lossless)

- **Constant encoding:** all vector values are the same
- **Bit packing (bitpacking):** representing small integers using fewer bits
 - Good for small integer values
- **Run length encoding (rle):** sequences of redundant data stored as the data value and the number of repetitions
 - Good if there are many adjacent duplicate values
- **Frame of reference (pfor):** deviation from a base value (with handling of outlier values); possibly encoding the difference between deviations of subsequent values
 - Good if data is ordered
- **Dictionary encoding (dictionary):** offsets into a dictionary of unique values
 - Good if limited number of frequent values
- **FSST (fsst):** Fast Random Access String Compression

Vector compression (DuckDB)



Sparse array compression (CSR)

Dense representation

$$\begin{bmatrix} 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 8 & 0 \end{bmatrix}$$

Sparse representation

1. Vector of non-zero values (values array)

```
[5, 9, 2, 6, 8]
```

2. Column index (for each value in the values array)

```
[2, 1, 6, 0, 5]
```

3. Row pointer (where each row. Starts in the values array)

```
[0, 1, 3, 3, 5]
```

Static and dynamic arrays

Static representation

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

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

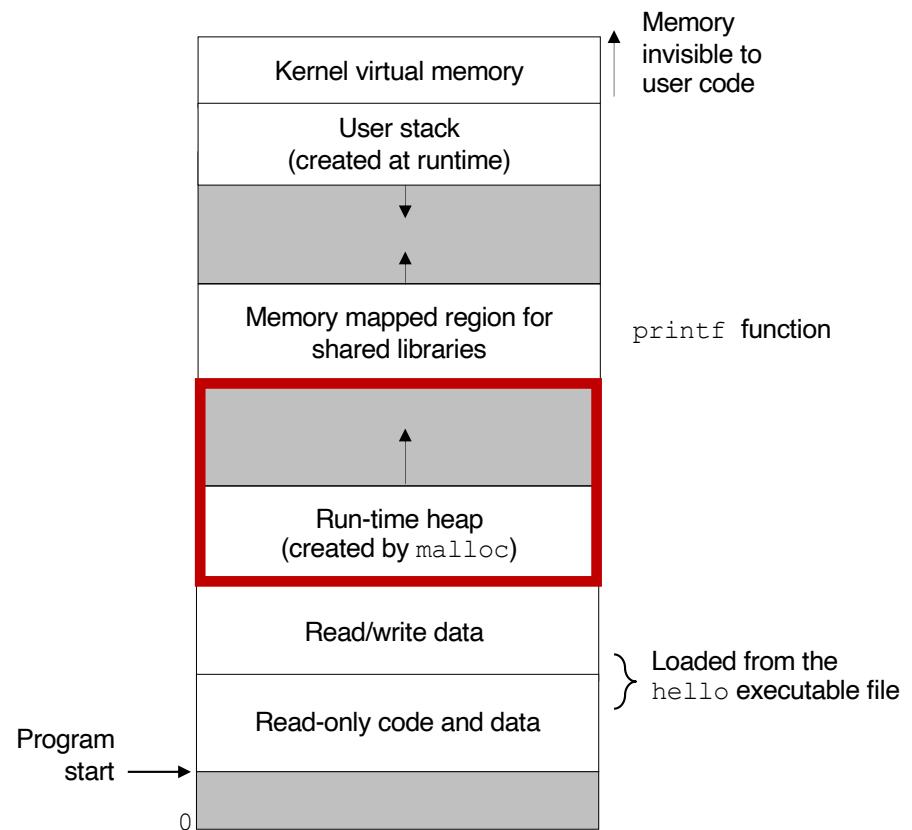
And then we can index it with for example A[1] [2]

Dynamic representation

Dynamic memory allocation

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire memory at run time.
 - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
 - **Explicit allocator:** application allocates and frees space
 - E.g., `malloc` and `free` in C
 - **Implicit allocator:** application allocates, but does not free space
 - E.g. garbage collection in Java, ML, and Lisp
- In C, we deal with explicit memory allocation.

The malloc Package

```
#include <stdlib.h>
void *malloc(size_t size)
    Successful:
        Returns a pointer to a memory block of at least size bytes
        (typically) aligned to 8-byte boundary
        If size == 0, returns NULL
    Unsuccessful: returns NULL (0) and sets errno
void free(void *p)
    Returns the block pointed at by p to pool of available memory
    p must come from a previous call to malloc or realloc
Other functions:
    calloc: Version of malloc that initializes allocated block to zero.
    realloc: Changes the size of a previously allocated block.
    sbrk: Used internally by allocators to grow or shrink the heap
```

malloc Example

```
void foo(int n, int m) {
    int i, *p;

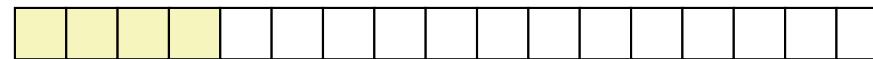
    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;

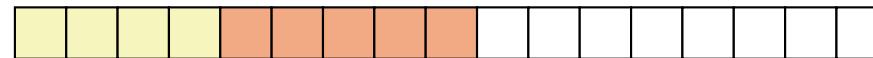
    /* Return p to the heap */
    free(p);
}
```

Allocation Example

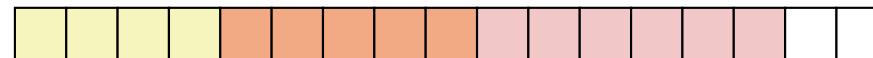
```
p1 = malloc(4)
```



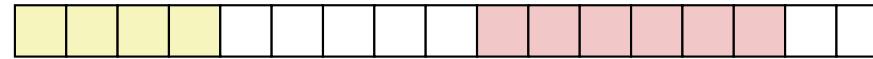
```
p2 = malloc(5)
```



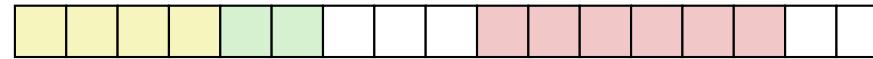
```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



Static and dynamic arrays

Dynamic representation

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

- Suppose `malloc()` returns the address 1000.
- When we do `arr[i]`, C computes the address $1000 + i \times \text{sizeof(int)}$ 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`”.)

Questions:

- **How much memory do we allocate?** An x -element array needs $x \times \text{sizeof(t)}$ 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...

1-dimensional array

```
#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*.

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

The type of index functions

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

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

Row-major vs Column-major order

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

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

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

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 i th coordinate of p , and the product of an empty series is 1.

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

Row-major vs Column-major order

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.

Arrays as parameters

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.

```
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.