Type Systems

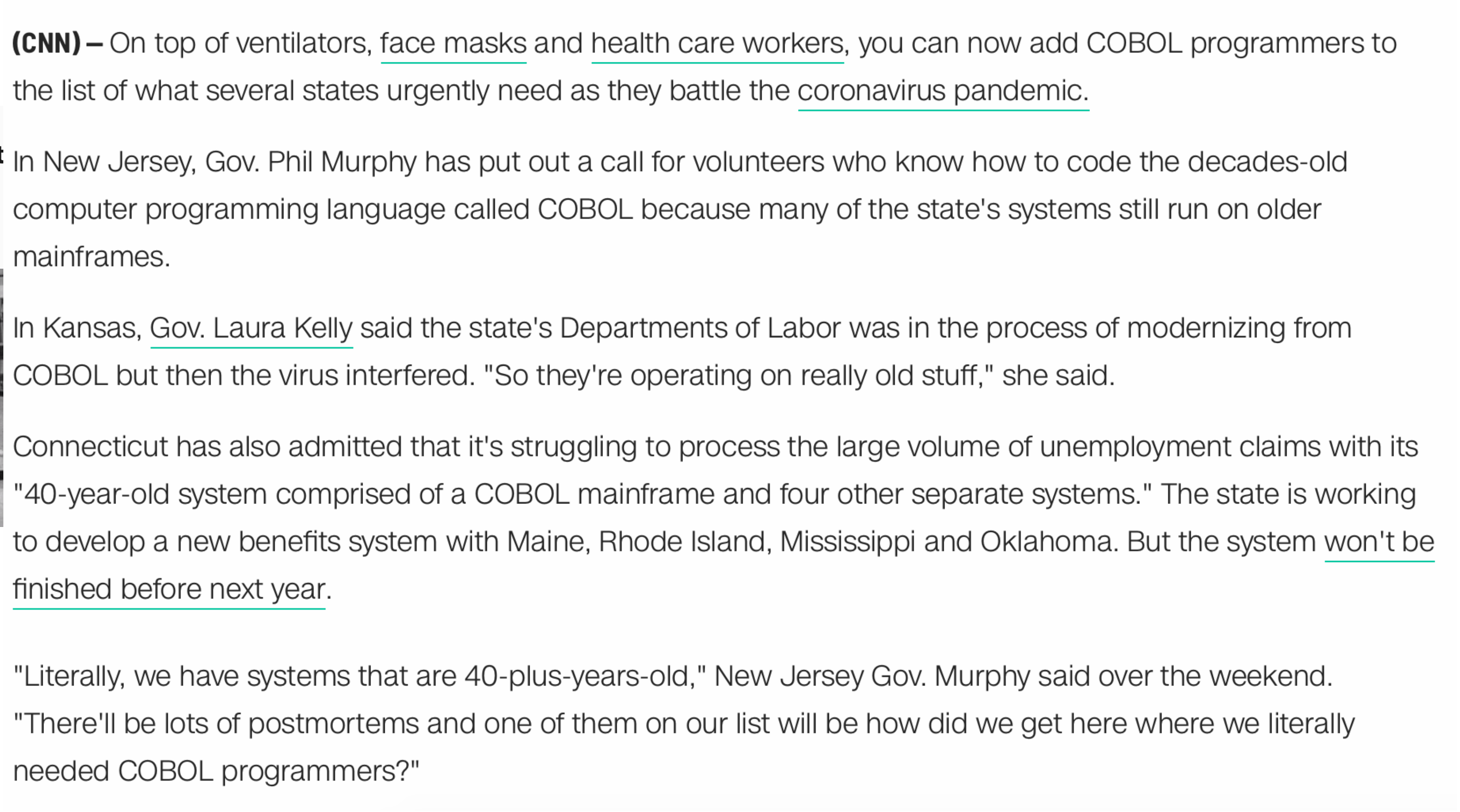
(Chapter 7 and 8 in book)

Reminder: PL are important to Society

Wanted urgently: People who know a half century-old computer language so states can process unemployment claims

By Alicia Lee, CNN
Updated 4:00 PM ET, Wed April 8, 2020

Taken from [here]()



(CNN) – On top of ventilators, face masks and health care workers, you can now add COBOL programmers to the list of what several states urgently need as they battle the coronavirus pandemic.

In New Jersey, Gov. Phil Murphy has put out a call for volunteers who know how to code the decades-old computer programming language called COBOL because many of the state's systems still run on older mainframes.

In Kansas, Gov. Laura Kelly said the state's Departments of Labor was in the process of modernizing from COBOL but then the virus interfered. "So they're operating on really old stuff," she said.

Connecticut has also admitted that it's struggling to process the large volume of unemployment claims with its "40-year-old system comprised of a COBOL mainframe and four other separate systems." The state is working to develop a new benefits system with Maine, Rhode Island, Mississippi and Oklahoma. But the system won't be finished before next year.

"Literally, we have systems that are 40-plus-years-old," New Jersey Gov. Murphy said over the weekend. "There'll be lots of postmortems and one of them on our list will be how did we get here where we literally needed COBOL programmers?"

Overview

- Mechanisms to define types and associate them with language constructs
- Set of rules for:
  - type equivalence: type(var1) = type(var2)?
  - type compatibility: in what context can *var* of type *type* be used?
  - type inference: what is the type of expression “a + 2 * b”?
- Type checking:
  - Does the program obey the language compatibility rules?
  - Type clash: rule violation
- Strongly typed languages *limit* and *enforce* application of operations based on type
- Weakly typed languages: variables not bound to specific types (e.g. C, PHP), several/all conversions are **implicit**
- Statically typed languages are usually strongly typed and can be enforced at compile-time

What are Types?

Depends on the view …

- Denotationally (as in what it denotes or represent): set of values, value in set? object is of a type if its value is in the set
- Structurally: type is built-in (e.g. int, char, etc) or composite (arrays, structures, classes, enumerations)
- Abstraction-based: interface consisting of operations (think of operations as functions being called) → what behavior is valid?

Classification of Types

- Booleans
- Characters and strings
- Numerics
- Enumerations
- Arrays
- Structures

Classification of Types

- Numerics: Integer and floating point numbers (real numbers such as 0.09482)
- Discrete types: integer, Boolean, characters
- Some languages (e.g. C and Fortran) provide several built-in numeric types varying in precision (the size of the type, and the range of values they can represent). See also GNU Multi-Precision Library or GMP and the Boost C++ Library.
- Numeric types in general not portable across languages
- Languages can provide signed (e.g. -128 to 127) and unsigned (0 to 255)
- Unsigned integers: cardinals
- Some languages provide complex types: imaginary, tuples
- Other languages (e.g. Scheme and Common Lisp) provide rational types, symbols, lists, or other things (See Coq types),etc.

Classification of Types

- Enumerations (Introduced in Pascal):
  - Example: `type weekday = (sun, mon, tue, wed, thu, fri, sat);`
  - Values are ordered, e.g. sun < thu
  - Mechanisms for finding the next greater value (*succ* for successor) and the previous lower value (*pred* for predecessor)
    - Example (Pascal): `tomorrow := succ(today)`
  - Can be used in loops:
    - Example (Pascal): `for today := mon to fri do begin` …
    - Also: Pascal only allows enumeration types in loops (integer, char), no real
  - Enumerations define new types in Pascal, not compatible with integers

Subrange

- Also introduced in Pascal
- *Contiguous subset* of values of some discrete *base* type:
  - 0 to 10 vs -10 to 5
  - Can “rename” them
  - Example (Pascal):

```
type  score = 0..10;
      workday = mon..fri;
if (day in mon..fri) then …
```

Composite Types

- Scalars (non-composite): any type that will represent one element:

  Example (C): `int c = 50;` // whatever value c has, it's only 1 value

- Contrast with non-scalar types: several values referable by a single variable (as in an array or list)
- Composite types: values have many parts
- Record (introduced in COBOL): consist of fields, each of a "simpler" type; its type is the cartesian product of the types of the fields

  Example (C):

  ```
  struct height { int feet; int inches; };
  struct person { char * name; int age; float weight; struct height };
  ```

- Variant record (union): similar to record, but *only one field can be used at any time*; its type is the disjoint union of its fields

  Example (C):

  ```
  union number { int n_int; long long int n_llint; float n_flt; };
  ```

Composite Types

- Arrays: formally, functions from an index type to a component type;
- Sets (also introduced in Pascal): the set type is the powerset of its base type; Pascal restricted sets to have at most 256 different values
- Pointers (l-values): reference to an object of the pointer's base type
- Lists: sequence of elements, but no notion of mapping or index (at least conceptually) → no direct access
- Files: represent data on devices; conceptually, a function mapping an index type (like an integer) to a component type; include notion of "current position"
  - Example (C): FILE * f; // a file of char
- Pascal allows to define files of user-defined record types:
  - Read and write operations on multiples of Record

Type Checking

- Most common in statically typed languages, but also widely used in dynamic scripting language (e.g. Python)
- Type required when defining variables, constants, **functions**
- Involves checking for:
    - **type compatibility**: conversion, coercion, non-converting type casts
    - **type inference**: decide the type of expression from its subexpressions
    - **type equivalence**: stricter version of type compatibility

Type Equivalence

- Answers the question: How to decide when two types are the same?
- Two main strategies:
  - Structural equivalence: ”same components, in the same order”
  - Name equivalence: type names matter, each definition is a new type
    - Example (in some artificial language similar to C):

```
typedef int intb;
typedef int inta;  // inta and intb would be quite different types
```

Structural Equivalence

- In Structural Equivalence:
  - Format does not matter: spacing ignored
  - Order of fields <u>usually matters</u>
- Language design question: Should the order of fields change the type?
  - Why?
  - Ideas?

```
type R1 = record
   a, b : integer
end;
```

$=$

```
type R1 = record
  a : integer;
  b : integer
end;
```

$\neq$

```
type R1 = record
  b : integer;
  a : integer
end;
```

Structural Equivalence

- Same array size
- Different indexing schemes
- Pascal allowed to define arbitrary bounds for arrays
- Should they be considered different types?

```
type str1 = array [1..10] of char;
```

= ?

```
type str2 = array [0..9] of char;
```

- Why?
- Ideas?

Question: Should the compiler warn the programmer when assigning arrays from one variable of type str1 to a variable of type str2?

Name Equivalence

- Consider the following:

```
type new_type = old_type; // Algol syntax
```

```
typedef old_type new_type; // C syntax
```

- `new_type` is an *alias* for `old_type`

compatibility among parent/child types

- Design question: should both names be considered the same type or different types?
- Two types of name equivalence:
  - *Strict name equivalence* (aliased types are distinct types)
  - *Loose name equivalence* (aliased types are the same type)
- Can achieve both with *derived types* and *subtypes*

Type Conversion and Casts

- *Type conversion* (aka *cast*): tells the compiler to treat a variable or expression as some other type
- Examples:

```
a := floor(3.75 * b);
r = (float) n;
n = (int) r;
```

Scenarios:

1. Types could be structurally equivalent, but language uses name equivalence
2. Types have different values, but share a common range of values (and the programmer knows what he/she is doing)
3. Types have different low-level implementation (usually boils down to some architecture specific function for converting types) → matters more for built-in types

Type Compatibility

- Most languages do not require equivalence of types in every context
- Resort to loser rules: type compatibility
- Examples:
  - Assignment: type(lhs) must be compatible with type(rhs)
  - Operands: types of operands must be compatible with some common type that supports the operator
  - Subroutine calls: types of arguments should be compatible with types of formal parameters

Type Compatibility

- *Type coercion*: automatic type conversion performed by the language/compiler
- Example: C has a relatively weak type system, performs a lot of coercion under the hood
- Not limited to built-in types, languages could allow coercion of arrays and records (structs)
    - Example (in Fortran):
        - Permits coercion of arrays if they have the same shape: same number of dimensions, same size along each dimension, same shape on base type; the keyword here is *shape*
        - Records: same number of fields, fields of same shape (order matters), names do not matter

Type Inference

- Question: what determines the overall type of an expression?
- A few examples:
  - Result of arithmetic expression is the types of the operands, possibly by coercing one of them into the more general one
  - Result of a comparison is a Boolean
  - Result of an assignment is the type of the left-hand side (lhs)
- More complicated example, subranges:
  - Assume two ranges are defined, range1 as -10..10 and range2 as 0..40
  - Assume a is of type range1 and b is of type range2
  - What is the type of a+b?
  - Depends, but usually is the type of the base type, i.e. typeof(-10)

Composite Types

- Records / Structures (as in C/C++): `struct` keyword
- Variant Records / Unions
- Arrays
- Record of arrays
- Arrays of records
- Pointers
- Objects

As machine memory addressing is linear, all composite types are laid out in some form ➔ flattened

Records and Structures

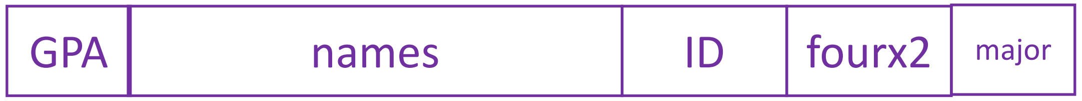
- Basic principle: To put together data in some way that “makes sense”, i.e. to structure it
- For example, to logically gather the information of a student in In C/C++:

```
struct student {
  char * names[10];
  char ID[10];
  char fourx2[10];
  int major;
};
```

- Originally introduced in COBOL (COmmon Business-Oriented Language) circa 1960, then followed by Algol 68; Fortran 90 calls it types

Records and Structures

```
struct student {
  float GPA;
  char * names[10];
  char ID[10];
  char fourx2[10];
  int major;
};
struct student s1;
struct student s2;
```



- Student s1 is declared somewhere in the program
- Assume the memory address used by s1 is M
- <u>In general, the compiler is free to do many things, among them changing the layout of the structure</u>
- If no changes are made to the original layout, the address offsets of s1's fields would be the following:
  - GPA: 0
  - names: 4
  - ID: 4 + 10*4 = 44
  - fourx2: 4 + 10*4 + 10*1 = 54
  - major: 4 + 10*4 + 10*1 + 10*1 = 64
- Total size of an instance of struct student would be 64 + 4 bytes
- The base address of s2 would then be M + 68.

Variants and Unions

- Informally, bundling of several data types, each identified by a field (as in the structure case)
- Memory allocated only for the largest field
- Field name determines how the memory is used / accessed
- Union can only be used in ONE WAY AT A TIME; otherwise bits and bytes are just interpreted “as is”
- Think about: How would you implement records/structs and variants/unions in your project compiler?

```
union student {
  double gpa; // 8 bytes
  char letter; // 1 byte
  float pct; // 4 bytes
};
```

Arrays

- Homogeneous data collection, i.e. a finite number of instances of some base data type
- Stored contiguously in memory
- Can be multi-dimensional:
  - `int A[N]; // 1-dimensional`
  - `int B[M][N]; // 2-dimensional`
  - `int C[M][N][P]; // 3-dimensional`
- Each individual element of the array identified by some multi-dimensional integer tuple, e.g. <i,j,k> = <0,2,4>
- Language and compiler map the integer tuple to some memory location:
  - $Z^n \rightarrow N$

Arrays

- Declaration obviously varies per language
- Language can define things as:
  - ❑ What data types can one use to access elements in the array?
  - ❑ Minimum address of an array along each dimension:
    - ❑ Zero-based arrays in C/C++
    - ❑ 1-based arrays as in Pascal
    - ❑ Arbitrary bounds (also in Pascal): part of the array type declaration
  - ❑ Special operators to access arrays:
    - ❑ Index operator, usually [] or ()
    - ❑ Slice operator, usually ":"

❑ Storage mechanisms for dense arrays (mostly non-zeros) vs sparse arrays (very high fraction of zeros, 90% or more of zeros)

❑ Lexicographic comparison: A, B arrays, A < B

```
[3,4,10] < [2,5,6] = FALSE;
[3,4,10] < [4,0,1] = TRUE
```

❑ Intrinsic functions to determine size of: whole array, each dimension, etc

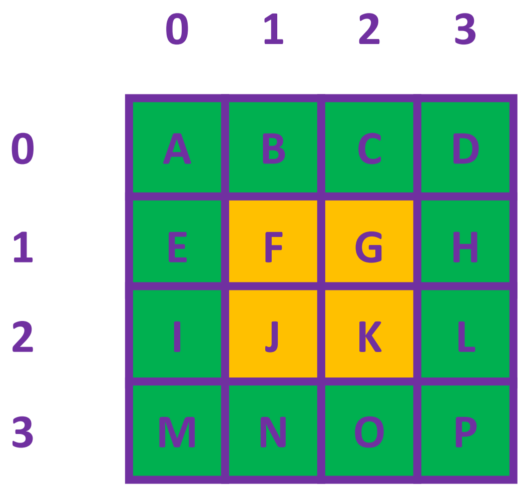
Arrays

- Fortran:
  - `real, dimension (10,10) :: mat`
- Modula-3:
  - `VAR mat: ARRAY [1..10],[1..10] OF REAL;`
  - `VAR mat: ARRAY [1..10] OF ARRAY [1..10] OF REAL;`
- C:
  - `double mat[10][10];`

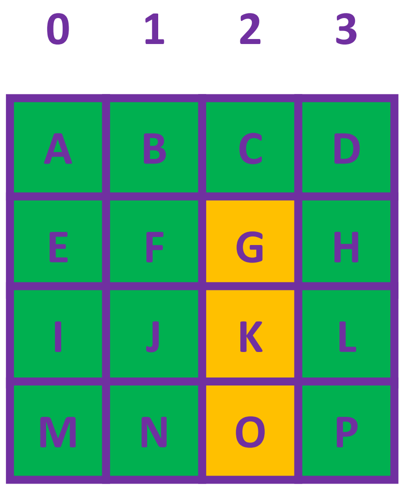
Arrays

Array slices (or Sections)

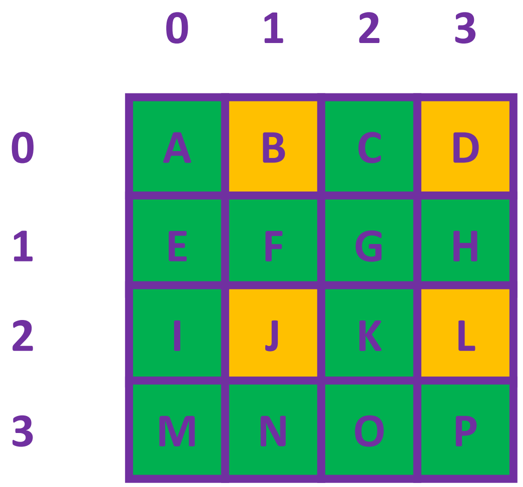
- Probably introduced in Fortran (circa 1957)
- Popular access mechanism in Matlab, Python, etc
- Allows to access subset or regions of multi-dimensional arrays
- Examples:
    - matrix(3:6, 4:7): from 3 to 6 and from 4 to 7
    - matrix (6:, 5) : from 6 onwards and only 5
    - matrix(:4, 2:8:2): up to 4, and starting from 2, up to 8, every 2
    - matrix(:, (/2, 5, 9/)): everything and only 2, 5 and 9



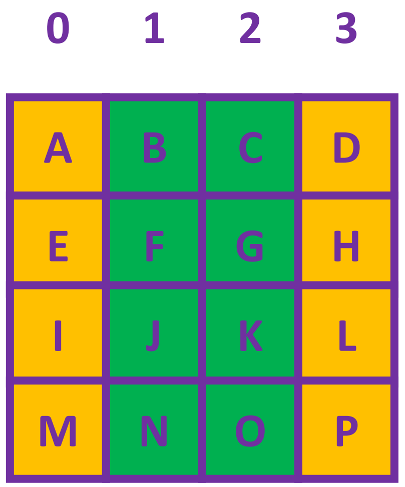
matrix(1:2, 1:2)



matrix(1:3, 2:2)



matrix(0:3:2, 1:3:2)



matrix(:, (/0,3\))

Arrays: Memory Layout

A	B	C	D
E	F	G	I
J	K	L	M

2D array of 3 x 4

Row Major Order (e.g. C):

A	B	C	D	E	F	G	I	J	K	L	M

Column Major Order (e.g. Fortran):

A	E	J	B	F	K	C	G	L	D	I	M

Arrays Memory Layout

In general, given an array $A[d_1,d_2,…,d_n]$:

- Row Major Order: each dimension d_k consists of $[d_{k+1},…,d_n]$ arrays
- Column Major Order: each dimension d_k consists of $[d_1,…,d_{k-1}]$ arrays

In memory:

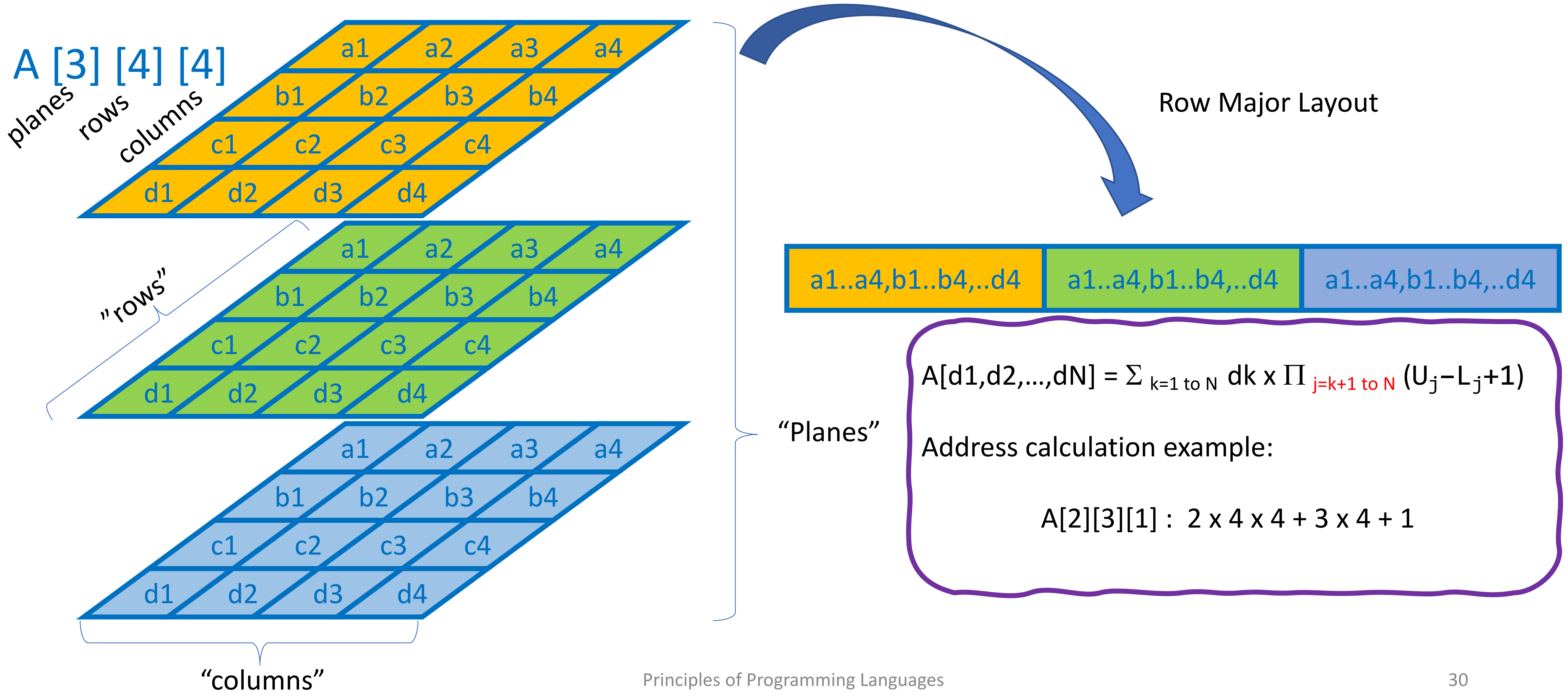
- Row Major Order stores dimensions “from left to right”
- Column Major Order stores dimensions “for right to left”

Address calculation, assume an array A[L1..U1,L2..U2,…LN..UN]:

- Row Major Order: $A[d1,d2,…,dN] = \Sigma_{k=1 \text{ to } N}\ dk \times \Pi_{j=k+1 \text{ to } N} (U_j - L_j + 1)$
- Column Major Order: $A[d1,d2,…,dN] = \Sigma_{k=N \text{ to } 1}\ dk \times \Pi_{j=1 \text{ to } k-1} (U_j - L_j + 1)$

Think about: How would you implement static (stack) multi-dimensional arrays in your compiler?

Row Major Order – Memory Layout

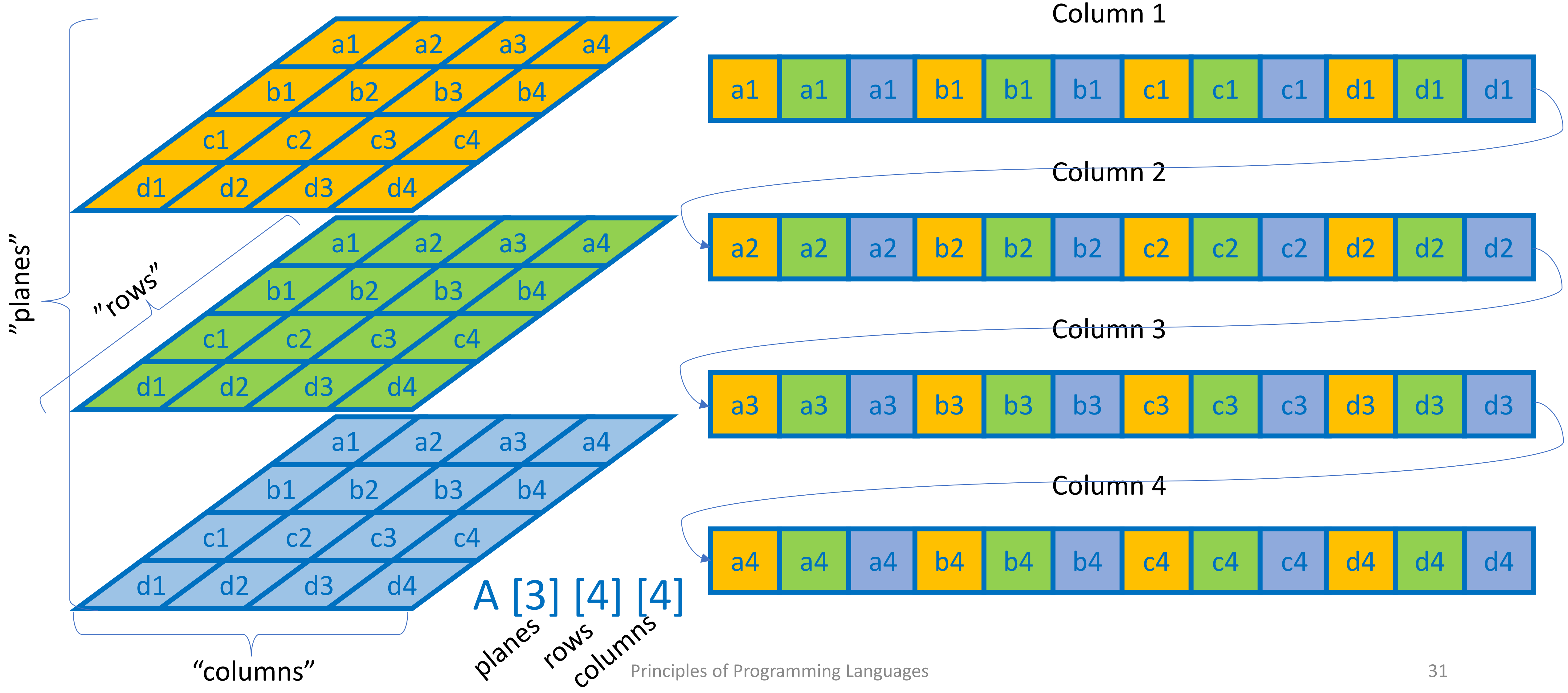


$A[d1,d2,\ldots,dN] = \Sigma_{k=1 \text{ to } N}\ dk \times \Pi_{j=k+1 \text{ to } N} (U_j - L_j + 1)$

Address calculation example:

A[2][3][1] : 2 x 4 x 4 + 3 x 4 + 1

Column Major Order - Memory Layout



"planes"
"rows"
"columns"
a1 a2 a3 a4
b1 b2 b3 b4
c1 c2 c3 c4
d1 d2 d3 d4
A [3] [4] [4]
planes
rows
columns
Column 1
a1 a1 a1 b1 b1 b1 c1 c1 c1 d1 d1 d1
Column 2
a2 a2 a2 b2 b2 b2 c2 c2 c2 d2 d2 d2
Column 3
a3 a3 a3 b3 b3 b3 c3 c3 c3 d3 d3 d3
Column 4
a4 a4 a4 b4 b4 b4 c4 c4 c4 d4 d4 d4

Pointers

- Introduced in PL/I ("Programming Language One")
- Language abstraction for "raw address" handling and dynamic memory allocation
- Requires abstractions for
  - Assigning chunks of memory during program execution
  - Reclaiming chunks of memory
  - Keeping track of what was assigned where (bookkeeping and accounting done by compiler and runtime system)
  - Operators accessing, storing and retrieving data to/from pointers

Pointers

- Behavior of pointers normally defined by a couple of things:
  - Imperative or functional language, e.g. C or Lisp?
  - Employment of reference model or value model
- Functional languages allocate memory as needed (everything in the heap), and use a reference model for names
- Variables in imperatives languages may use reference or value model:
  - C uses a value model: A = B puts value B into (address) of A
  - If B should refer to A, both should be pointers
- Java uses a mix: value model for built-in types, and reference model for user-defined types (objects)
- C# introduces the `unsafe` qualifier to permit usage of raw pointers

Pointer Operators

- Dereferencing operator (prefix * in C or postfix ^ in Pascal/Modula) permits to access the value to which a pointer refers to
- In C, pointers and arrays are closely related:
  - Subscript operators [] are syntactic sugar for pointer arithmetic:
    - a[3] is equivalent to *(a+3)
    - a[3] is equivalent to 3[a]
  - Pointer arithmetic is one of C's strengths:
    - `while (*p++); // same as while (p[i++] != 0); p being char *`
    - `a[i][j] = (*(a+i))[j] = *(a[i]+j) = *(*(a+i)+j)`
  - sizeof operator: returns the size in bytes of an object or type; when given a pointer, it returns the size of the pointer itself:
    - double *a; // sizeof(a) = 4 bytes while sizeof(*a) = 8 bytes
    - double (*b)[10]; // pointer to array of 10 doubles, sizeof(*b) = 80

Garbage Collection

- Mechanism for reclaiming memory
- Provided by managed languages such as Java, C#, Scala, Go
- C uses a wonderful abstraction for garbage collection: the programmer
- Main classes of garbage collection:
    - Reference counting:
    - Smart pointers
    - Tracing collection: Mark-and-sweep (See here if interested)
- Most algorithms differ in terms of memory space (overhead) required for bookkeeping and the time overhead required to perform the static and runtime analyses, as well as the execution of the collection

Garbage Collection

Reference counting:

- Keep counters for each chunk of dynamically assigned memory
  - Counter started at 1 with “new”
  - Doing: A = B, decrements the counter of A
  - Subroutines epilogue decrement the counters of all references
  - Reclamation performed when counter reach zero, i.e. when nothing is using some memory previously assigned
- Weakness : when is an object useful?
  - #references = 0: perfectly useless
  - #references > 0: could still be useless, for instance, a circular list
- See tombstones

Garbage Collection: Reference Counting

```
Example:
ptr_type ptr1 = allocate (); //  ref_count(object1) = 1
ptr_type ptr2 = ptr1; //  ref_count(object1) = 2
ptr_type ptr3 = allocate (); // ref_count(object2) = 1
ptr2 = ptr3; // ref_count(object1) = 1, ref_count(object2) = 2
ptr1 = NULL; // ref_count(object1) = 0 → deallocate object1
```

Garbage Collection

Smart pointers:

- Usually provided as libraries
- They provide: reference counting, bounds checking, debugging instrumentation, among other things
- See **`unique_ptr, shared_ptr, weak_ptr`** in C++:
  - unique_ptr: single owner of an object, pointer assignment transfers ownership
  - shared_pointer: implements reference counting; bad for circular data structures
  - weak_ptr: can be used in tandem with shared_pointers but do not increment the reference count