Applications of Multi-dimensional Arrays (1A)

Young Won Lim
7/19/21

Please send corrections (or suggestions) to youngwlim@hotmail.com.
This document was produced by using LibreOffice.

Access expressions
and dual type constrains

c **[i][j][k]** 3-d access

Sub-array types in a 3-d array

int c [L][M][N];

c	[i]	[j][k]
c[i]	[j]	[k]
c[i][j]	[k]	

abstract data	virtual array pointer
3-d array names c	2-d array pointer c
int [L][M][N]	int (*)[M][N]
2-d array names c[i]	1-d array pointer c[i]
int [M][N]	int (*) [N]
1-d array names c[i][j]	0-d array pointer c[i][j]
int [N]	int (*)

dual type

Associativity and Equivalence Relations

left-to-right associativity		left-to-right associativity
((c [i])[j])[k]	≡	*(*(*(c +i) +j) +k)

X[n]	≡	*(X+n)

given c[i][j]	c[i][j][k]	≡	*(c[i][j]+k)	for all k
given c[i]	c[i][j]	≡	*(c[i]+j)	for all j
given c	c[i]	≡	*(c+i)	for all i

Requirements for the expression **c[i][j][k]**

c[i][j][k]

for a given c[i][j],	for all k	c[i][j][k] = *(c[i][j]+k)
for a given c[i],	for all j	c[i][j] = *(c[i]+j)
for a given c,	for all i	c[i] = *(c+i)

3 contiguity requirements

for a given c[i][j],	contiguous c[i][j][k]'s
for a given c[i],	contiguous c[i][j]'s
for a given c,	contiguous c[i]'s

for a given subarray pointer — contiguous subarrays

Equivalent requirements for the expression **c[i][j][k]**

for all k	c[i][j][k] = *(c[i][j]+k)	
for all j	c[i][j] = *(c[i]+j)	
for all i	c[i] = *(c+i)	

⟺

&c[i][j][k] = c[i][j]+k	for all k
&c[i][j] = c[i]+j	for all j
&c[i] = c+i	for all i

⇕

c[i][j][0] = *(c[i][j])
c[i][0] = *(c[i])
c[0] = *(c)

with contiguous subarrays

⟺

&c[i][j][0] = c[i][j]
&c[i][0] = c[i]
&c[0] = c

with contiguous subarrays

Sub-array address calculation in a 3-d array

&c[i][j][k] = c[i][j]+k	for all k	= c[i][j]	+ k *sizeof(c[i][j][0])
&c[i][j] = c[i] +j	for all j	= c[i]	+ j *sizeof(c[i][0])
&c[i] = c +i	for all i	= c	+ i *sizeof(c[0])

&c[i][j][k]	for all k	= &c[i][j][0]	+ k * 4
&c[i][j]	for all j	= &c[i][0][0]	+ j * 4*4
&c[i]	for all i	= &c[0][0][0]	+ i * 3*4*4

```
int     c [2][3][4];
```

Two approaches for the **3-d** access pattern **c[i][j][k]**

General requirements

c [i][j][k]

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

Pointer array approach

```
int** c[2];
int*  b[2*3];
int   c[2*3*4];
```

```
c[i][j][k]   :: int
c[i][j]      :: int *
c[i]         :: int **
c            :: int ***
```

c[i] ⇐ &b[i*3]
b[j] ⇐ &a[j*4]

with contiguous a, b, c

Explicit
Arrays of Pointers with
Multiple Indirection

N-dim Array approach

```
int c[2][3][4];
```

```
c[i][j][k] :: int
c[i][j]    :: int (*)
c[i]       :: int (*)[4]
c          :: int (*)[3][4]
```

c[i][j] ⇐ &c[i][j][0]
c[i] ⇐ &c[i][0][0]
c ⇐ &c[0][0][0]

with contiguous c[i], c[i][j], c[i][j][k]

Implicit
Nested
Virtual Array Pointers

3-d access pattern **c[i][j][k]** – N-dim array approach

General requirements

c [i][j][k]

↕

&c[i][j][k]	= c[i][j]+k	for all k
&c[i][j]	= c[i]+j	for all j
&c[i]	= c+i	for all i

⟺

N-dim Array approach

```
int c[2][3][4];
```

```
c[i][j][k] :: int
c[i][j]    :: int (*)
c[i]       :: int (*)[4]
c          :: int (*)[3][4]
```

```
c[i][j]  ⇐  &c[i][j][0]
c[i]     ⇐  &c[i][0][0]
c        ⇐  &c[0][0][0]
```

with contiguous c[i], c[i][j], c[i][j][k]

Implicit
Nested
Virtual Array Pointers

Virtual assignments

virtual assignments

c	⇐	&c[0][0][0]
c[i]	⇐	&c[i][0][0]
c[i][j]	⇐	&c[i][j][0]

row major ordering
contiguous linear layout

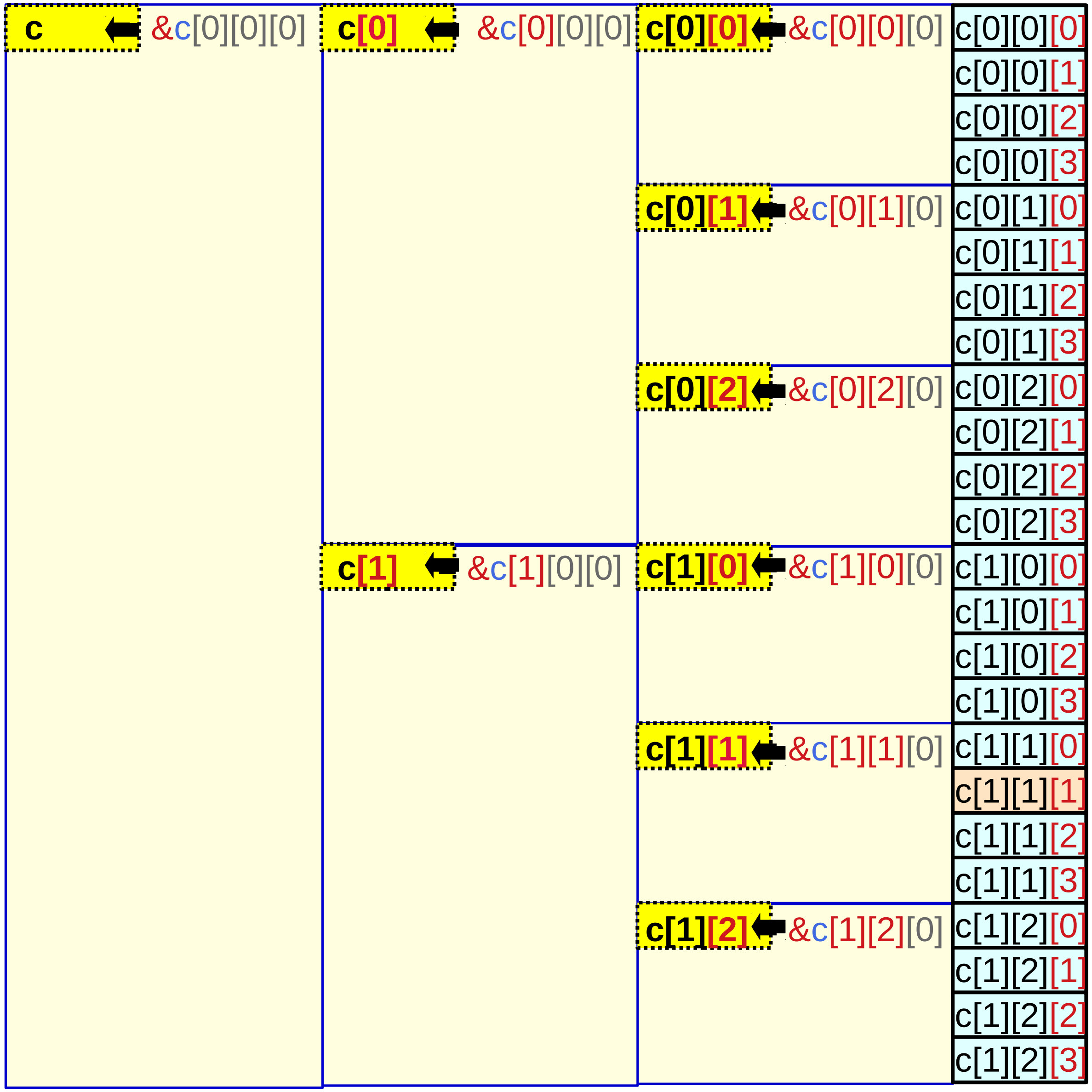
		type casts	address values
c	⇐	(int (*)[3][4])	&c[0][0][0]
c[i]	⇐	(int (*)[4])	&c[i][0][0]
c[i][j]	⇐	(int (*))	&c[i][j][0]

if c, c[i], c[i][j] were real pointer variables,
type casts would be needed

Virtual assignments of virtual array pointers

virtual assignments

c	⟸	&c[0][0][0]
c[i]	⟸	&c[i][0][0]
c[i][j]	⟸	&c[i][j][0]



Virtual assignments and contiguity requirements

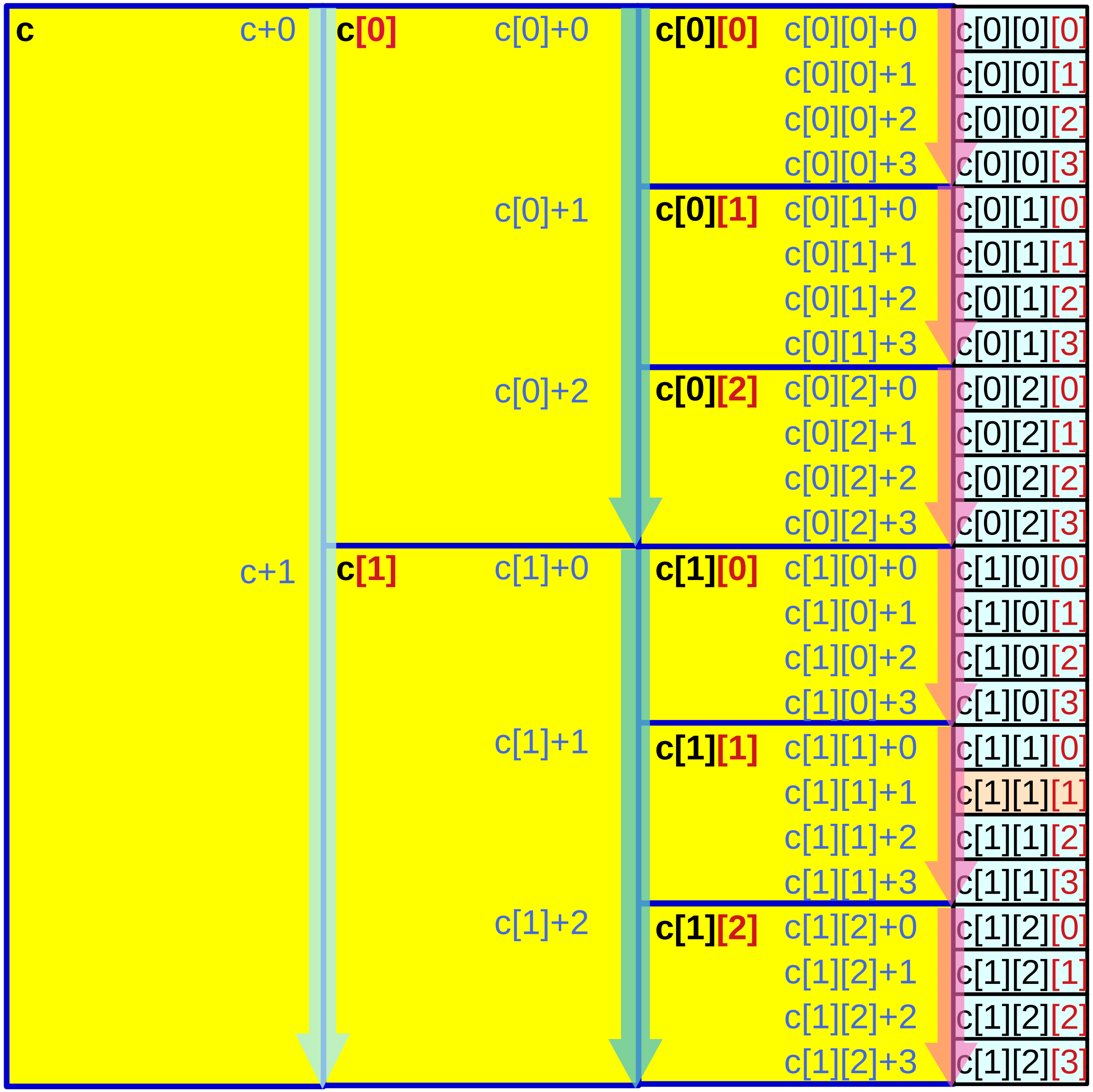
Three contiguity requirements

&c[i][j][k]	= c[i][j]+k	for all k
&c[i][j]	= c[i]+j	for all j
&c[i]	= c+i	for all i

for a given c[i][j], contiguous c[i][j][k]'s

for a given c[i], contiguous c[i][j]'s

for a given c, contiguous c[i]'s



3-d access pattern **c[i][j][k]** – N-dim array approach

3 contiguity constraints

&c[i][j][k]	= c[i][j]+k	for all k
&c[i][j]	= c[i]+j	for all j
&c[i]	= c+i	for all i

virtual assignments

c[i][j]	⟵	&c[i][j][0]
c[i]	⟵	&c[i][0][0]
c	⟵	&c[0][0][0]

with contiguous c[i], c[i][j], c[i][j][k]

3 contiguity constraints

&c[i][j][k]	= c[i][j]+k	for all k
&c[i][j]	= c[i]+j	for all j
&c[i]	= c+i	for all i

Dual type constraints

c[i][j]	= &c[i][j]
c[i]	= &c[i]
c	= &c

virtual assignments

c[i][j]	= &c[i][j][0]
c[i]	= &c[i][0][0]
c	= &c[0][0][0]

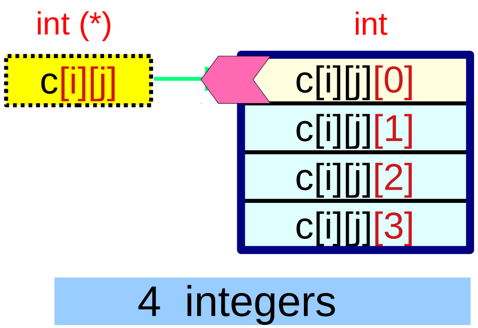
Virtual array pointer

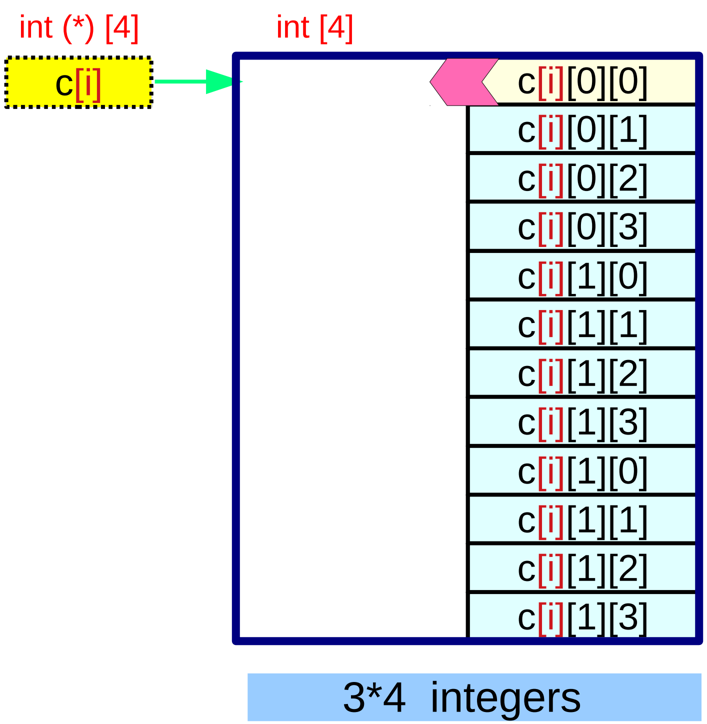
c[i][j]+k : k integer away
c[i]+j : 4*j integer away
c+i : 3*4*i integer away

Abstract data

c[i][j] has 4 integers
c[i] has 3*4 integers
c has 2*3*4 integers

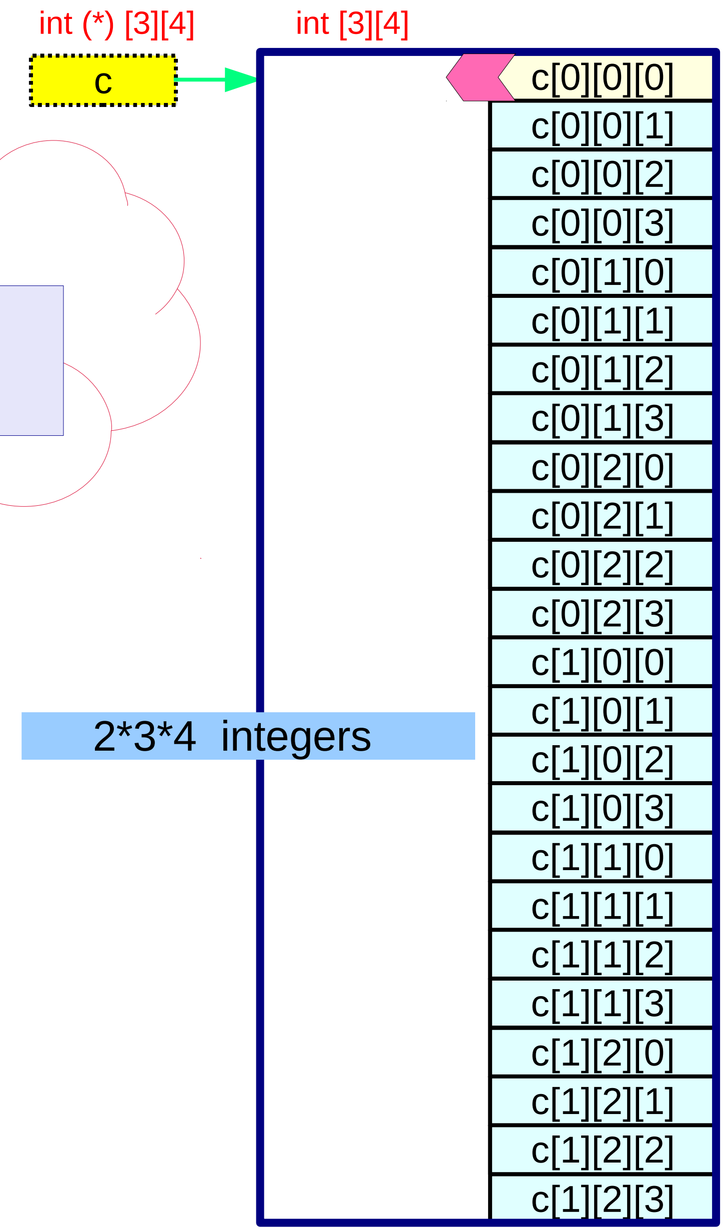
Assigning **c[i][j]**, **c[i]**, **c**



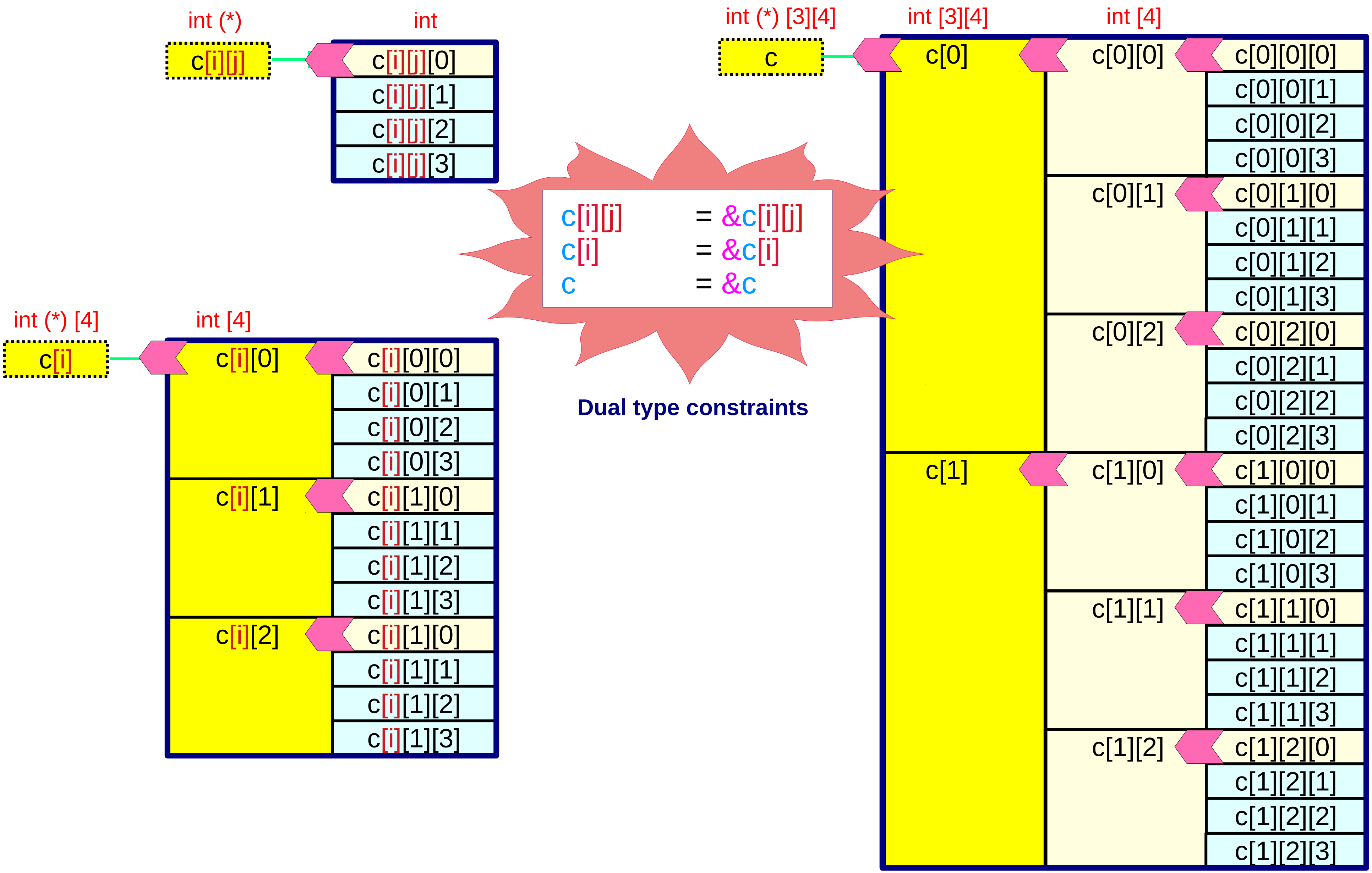


virtual assignments

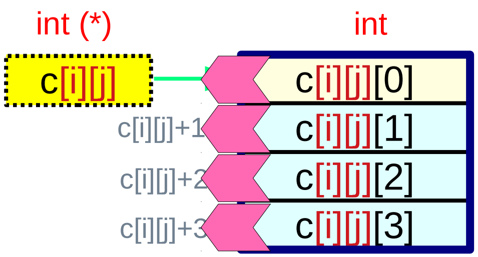
c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]



The addresses **c[i][j]**, **c[i]**, **c** with dual type constraints



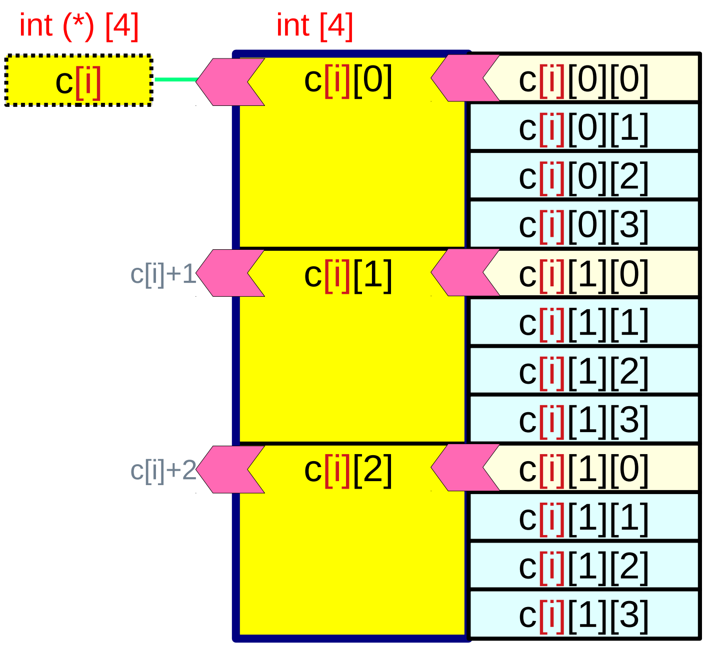
The addresses **c[i][j]**+k, **c[i]**+j, **c**+i



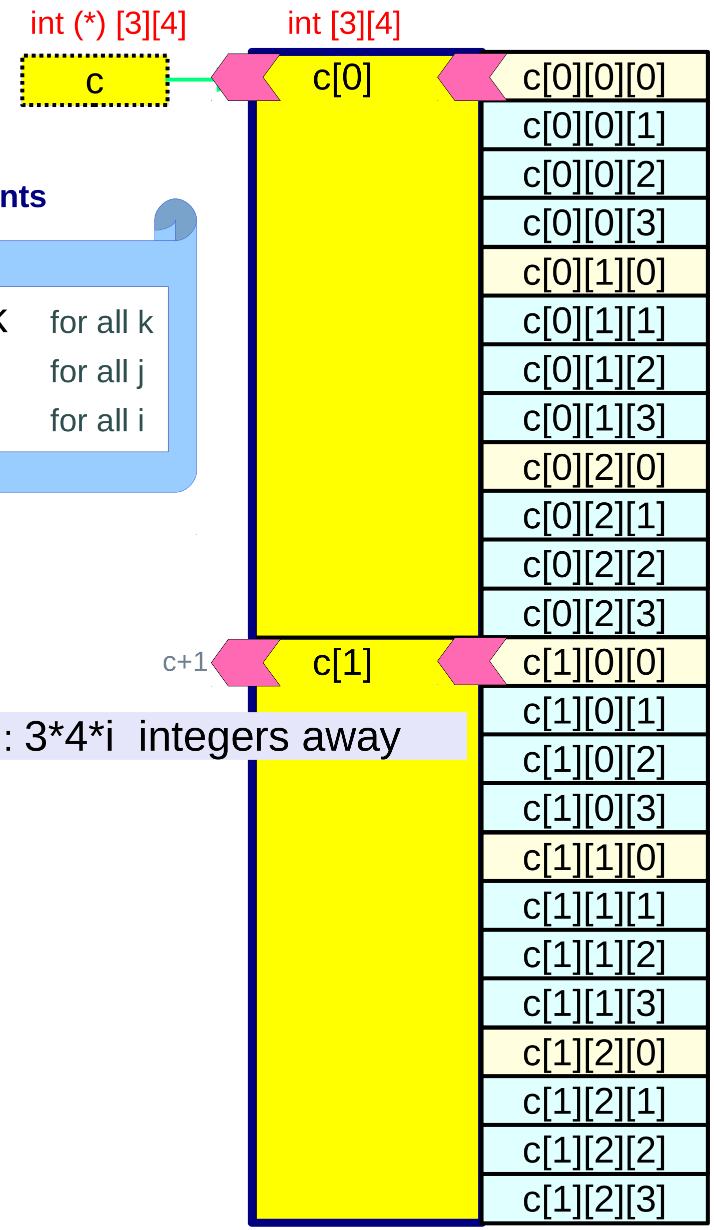
c[i][j]**+k** : k integer away

3 contiguity constraints

&c[i][j][k] = c[i][j]+k		for all k
&c[i][j] = c[i]+j		for all j
&c[i] = c+i		for all i



c[i]**+j** : 4*j integers away



c**+i** : 3*4*i integers away

Assignment → 3 contiguity requirements

int c [L][M][N];

c [i][j][k]

multi-dimensional arrays

assignments

c[i][j]	=	&c[i][j][0]
c[i]	=	&c[i][0][0]
c	=	&c[0][0][0]

→

constraints

&c[i][j][k]	=	c[i][j]+k	for all k
&c[i][j]	=	c[i]+j	for all j
&c[i]	=	c+i	for all i

assignments

c[i][j]	=	&c[i][j][0]
c[i]	=	&c[i][0][0]
c	=	&c[0][0][0]

←

constraints

&c[i][j][k]	=	c[i][j]+k	for all k
&c[i][j]	=	c[i]+j	for all j
&c[i]	=	c+i	for all i

Virtual array pointers and strides

int c [2][3][4];

&c[i][j][0] = c[i][j]
&c[i][0] = c[i]
&c[0] = c

with contiguous subarrays

&c[i][j][k]	= c[i][j]+k	for all k
&c[i][j]	= c[i]+j	for all j
&c[i]	= c+i	for all i

virtual assignments

c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

Virtual assignments

int (*)	c[i][j]	=	(int (*))	&c[i][j][0]
int (*) [4]	c[i]	=	(int (*) [4])	&c[i][0][0]
int (*) [3][4]	c	=	(int (*) [3][4])	&c[0][0][0]

Pointer Types

Sizes of abstract data types

int [4]	c[i][j]	size =4*4
int [3][4]	c[i]	size =3*4*4
int [2][3][4]	c	size =2*3*4*4

Abstract Data Types

Strides of array elements

c[i][j][0]	stride =4*4
c[i][0][0]	stride =3*4*4
c[0][0][0]	stride =2*3*4*4

contiguous

c[i][j]	contains	4 integers	i=[0:1], j=[0:2], k=[0:3]
c[i]	contains	3*4 integers	i=[0:1], j=[0:2], k=[0:3]
c	contains	2*3*4 integers	i=[0:1], j=[0:2], k=[0:3]

Virtual array pointer increment and strides

int c [2][3][4];

&c[i][j][0] = c[i][j]
&c[i][0] = c[i]
&c[0] = c

with contiguous subarrays

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

virtual assignments

c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

Pointer Types			
c[i][j]	has an address of	c[i][j][0]	as its value
c[i]	has an address of	c[i][0][0]	as its value
c	has an address of	c[0][0][0]	as its value

Pointer Types			
c[i][j]+1	has an address of	c[i][j][1]	1 integer away
c[i]+1	has an address of	c[i][1][0]	4*1 integers away
c+1	has an address of	c[1][0][0]	3*4*1 integers away

Pointer Types			
c[i][j]+k	has an address of	c[i][j][k]	k integers away
c[i]+j	has an address of	c[i][j][0]	4*j integers away
c+i	has an address of	c[i][0][0]	3*4*i integers away

Assignment ← 3 contiguity requirements

multi-dimensional arrays

c [i][j][k]

assignments

c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

→

3 contiguity requirements

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

assignments

c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

←

3 contiguity requirements

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

Array pointer relationships and dual type constraints

```
int     c [2][3][4];
```

&c[i][j][0] = c[i][j]
&c[i][0] = c[i]
&c[0] = c

with contiguous subarrays

&c[i][j][k] = c[i][j]+k	for all k
&c[i][j] = c[i]+j	for all j
&c[i] = c+i	for all i

virtual assignments

c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

Array pointer relationships

c[i][j] = &c[i][j][0]

c[i] = &c[i][0]

c = &c[0]

Dual type constraints

&c[i][0] = c[i][0]

&c[0] = c[0]

Virtual array pointer values

&c[i][j][0] = c[i][j]
&c[i][0] = c[i]
&c[0] = c

with contiguous subarrays

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

Array pointer relationships

c[i][j]= &c[i][j][0]

c[i] = &c[i][0]

c = &c[0]

Dual type constraints

c[i][j]= &c[i][j][0]

c[i] = &c[i][0]

c = &c[0]

c[0][0]= &c[0][0][0]

c[0] = &c[0][0]

c = &c[0]

c[i][0]= &c[i][0][0]

c[i] = &c[i][0]

c = &c[0]

c[i][j]= &c[i][j][0]

c[i] = &c[i][0]

c = &c[0]

Subarray starting addresses

c →	c[0] →	c[0][0] →	c[0][0][0]
		c[0][1] →	c[0][1][0]
		c[0][2] →	c[0][2][0]
	c[1] →	c[1][0] →	c[1][0][0]
		c[1][1] →	c[1][1][0]
		c[1][2] →	c[1][2][0]

c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

c[i][j]= &c[i][j][0]

c[i][0]= &c[i][0][0]
c[i] = &c[i][0]

c[0][0]= &c[0][0][0]
c[0] = &c[0][0]
c = &c[0]

c[0][0] = &c[0][0][0]
c[0][1] = &c[0][1][0]
c[0][2] = &c[0][2][0]
c[1][0] = &c[1][0][0]
c[1][1] = &c[1][1][0]
c[1][2] = &c[1][2][0]

c[0] = &c[0][0] = &c[0][0][0]
c[1] = &c[1][0] = &c[1][0][0]

c = &c[0] = &c[0][0] = &c[0][0][0]

Contiguity constraints in a multi-dimensional array

Array pointers and dual types

int (*) [N] — int (*) [M][N] — int (*) [L][M][N]

Array Pointer

1-dim array	2-dim array	3-dim array	
int [N]	int [M][N]	int [L][M][N]	Array Name
int (*)	int (*) [N]	int (*)[M][N]	the corresponding *virtual* array pointer

n-dimensional Array	dual type
(**n**-1)-dimensional Virtual Array Pointer	

Array pointer approach – contiguity constraints

abstract data virtual pointer

contiguous c[i], for a given c

c[0]
c[1]

abstract data virtual pointer

contiguous c[i][j], for a given c[i]

c[0][0] c[0][1] c[0][2]

c[1][0] c[1][1] c[1][2]

primitive data virtual pointer

contiguous c[i][j][k], for a given c[i][j]

c[0][0][0] c[0][0][1] c[0][0][2] c[0][0][3]

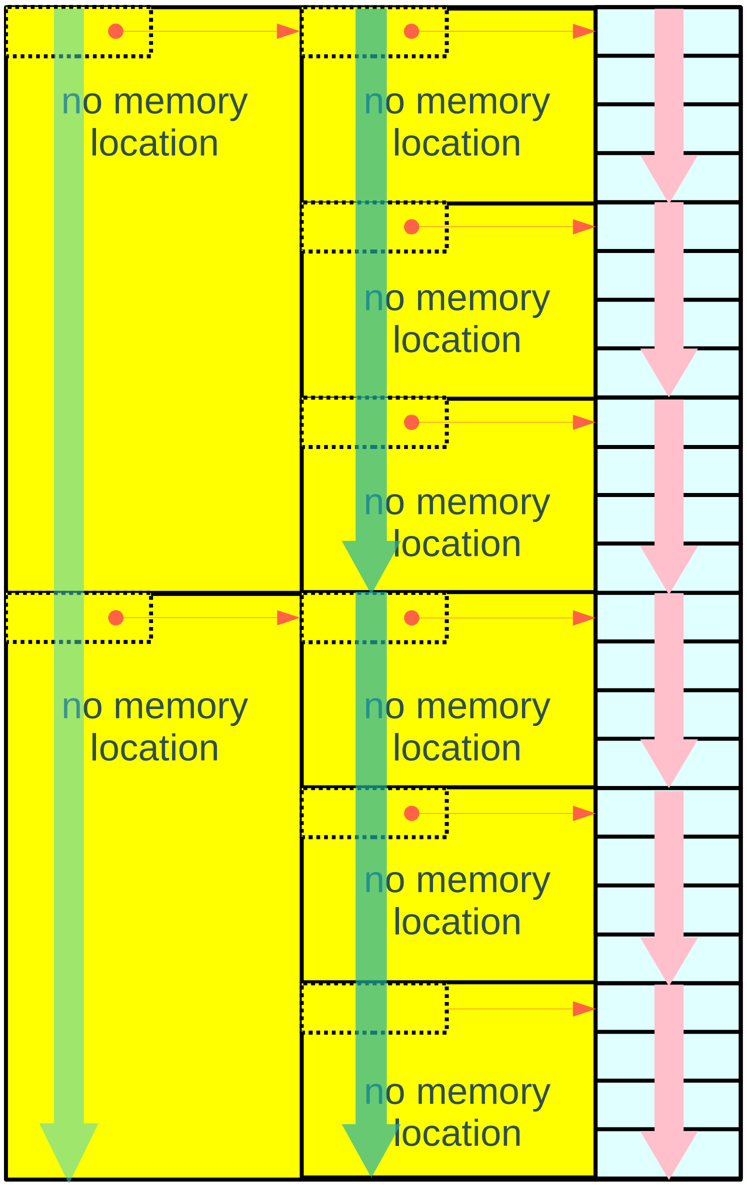
c[0][1][0] c[0][1][1] c[0][1][2] c[0][1][3]

c[0][2][0] c[0][2][1] c[0][2][2] c[0][2][3]

c[1][0][0] c[1][0][1] c[1][0][2] c[1][0][3]

c[1][1][0] c[1][1][1] c[1][1][2] c[1][1][3]

c[1][2][0] c[1][2][1] c[1][2][2] c[1][2][3]



Equivalence and contiguity (1)

consecutive address　　　consecutive data

$*(X+n) \equiv X[n]$

contiguous index : n

pointer type　　**abstract data type**

for a given int (*)　　continuous N int's

int **X[4];** for a given X, contiguous X[i] : **primitive types**

for a given int * (*)　　continuous N int (*)'s

int * **X[4];** for a given X, contiguous X[i] : **pointer types**

for a given atype (*)　　continuous N atype's

atype **X[4];** for a given X, contiguous X[i] : **abstract data types**

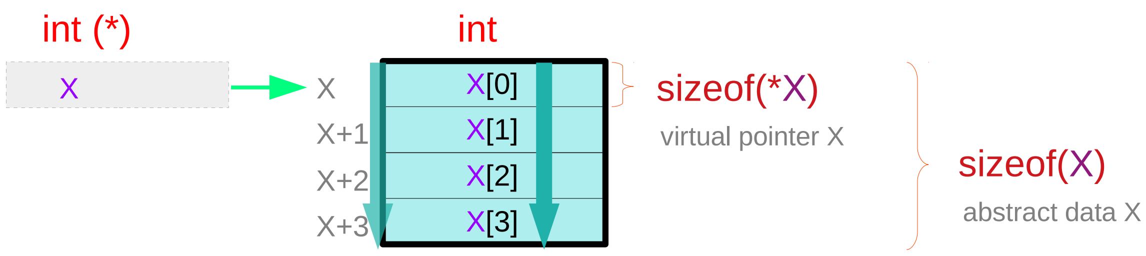
Equivalence and contiguity (2)

consecutive address — consecutive data

$*(X+n) \equiv X[n]$

contiguous index : n

int X[4]; for a given X, contiguous X[i] : **primitive types**



int * X[4]; for a given X, contiguous X[i] : **pointer types**

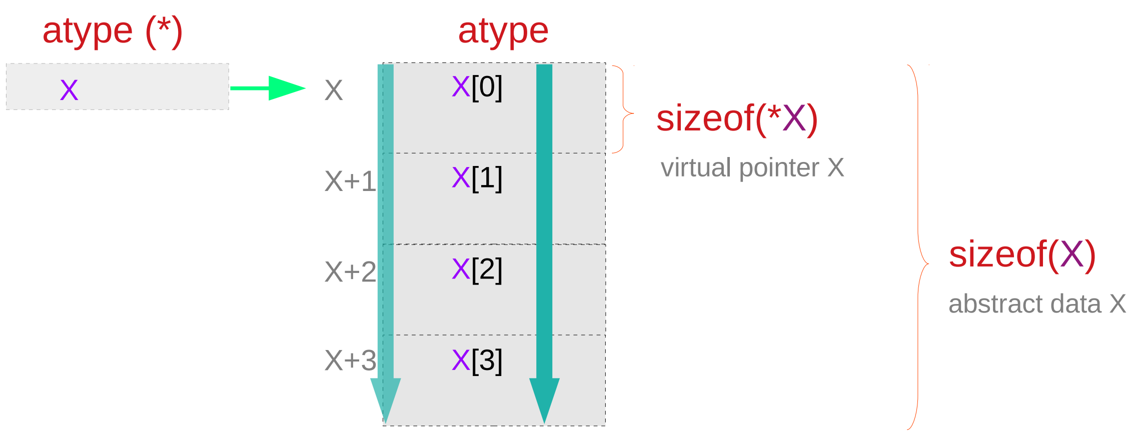
Equivalence and contiguity (3)

consecutive address	consecutive data

$*(X+n) \equiv X[n]$

contiguous index : n

atype X[4]; for a given X, contiguous X[i] : **abstract data types**



atype (*)	atype
int (*)	int
int (*) [N]	int [N]
int (*) [M][N]	int [M][N]
int (*) [L][M][N]	int [L][M][N]

Recursive applications of equivalences

By definition, contiguous memory locations are assumed

consecutive address		consecutive data
*(X+n)	≡	X[n]

contiguous index : n

*(p[m]+n)	⟺	p[m][n]	**Type 1**
(*(p+m))[n];	⟺	p[m][n];	**Type 2**

for a given int (*) — continuous N int's

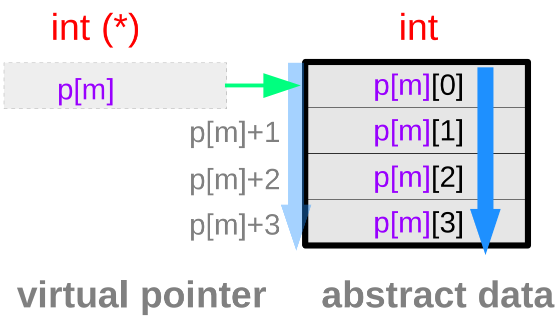
X = p[m] — contiguous index : **n**

for a given int (*) [N] — continuous M int [N]'s

X = p — contiguous index : **m**

Contiguity constraints in **2-d** arrays

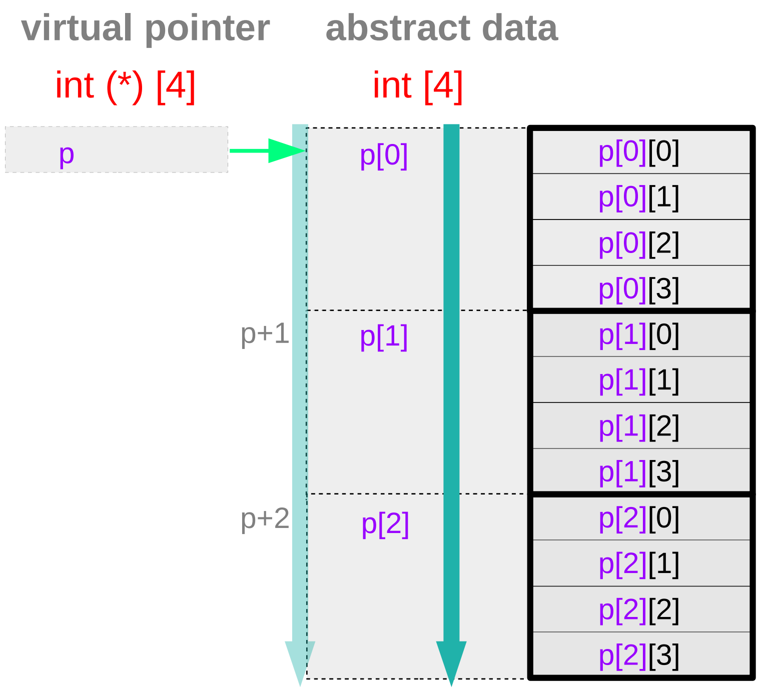
*(p[m]+n) ⟺ p[m][n] **Type 1**



for a given p[m], thus for a given p and m,
p[m][n]'s must be contiguous for all n.
p[m][0], p[m][1], … , p[m][N-1]

contiguous index : n

(*(p+m))[n]; ⟺ p[m][n]; **Type 2**



for a given p,
p[m]'s must be contiguous for all m.
p[0], p[1], … , p[M-1]
each p[m] contains N elements

contiguous index : m

Type 1 contiguity constraints (1)

consecutive address consecutive data

*(X+n) ≡ X[n]

contiguous index : n

*(p[m]+n) ⟺ p[m][n] **Type 1**

for a given p[m] contiguous index : **n**

pointer type **abstract data type**

for a given int (*) continuous N int's

int **p[M][N];** for a given p[m], contiguous p[m][n]: **primitive types**

for a given int * (*) continuous N int *'s

int * **p[M][N];** for a given p[m], contiguous p[m][n] : **pointer types**

for a given atype (*) continuous N atype's

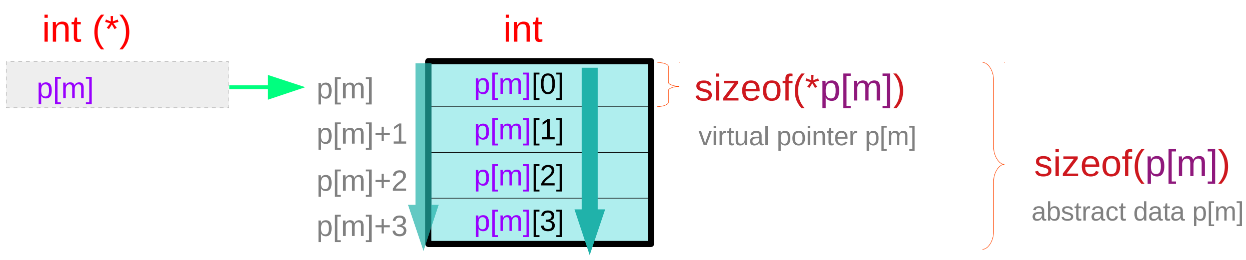
atype **p[M][N];** for a given p[m], contiguous p[m][n] : **abstract data types**

Type 1 contiguity constraints (2)

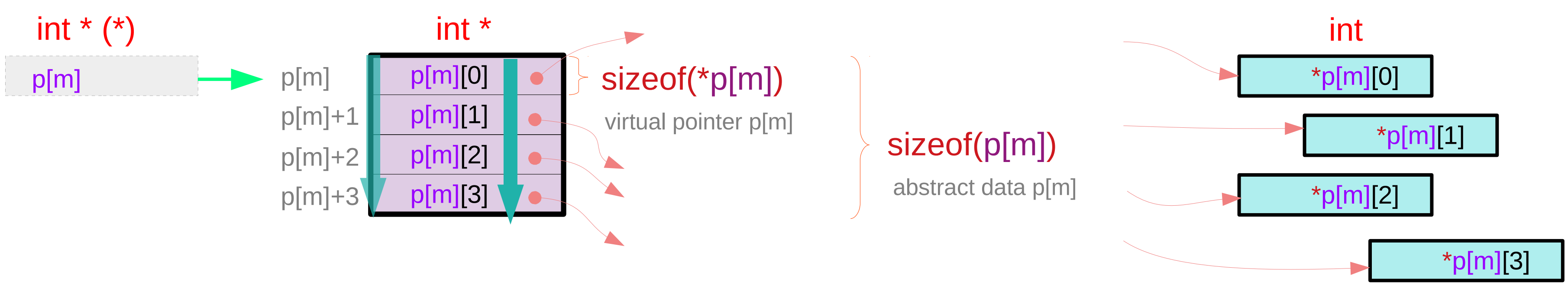
*(p[m]+n) ⟷ p[m][n] **Type 1**

for a given p[m]   contiguous index : **n**

int p[M][4]; for a given p[m], contiguous p[m][n] : **primitive types**   m = 0, 1, …, M-1



int * p[M][4]; for a given p[m], contiguous p[m][n] : **pointer types**   m = 0, 1, …, M-1



Type 1 contiguity constraints (3)

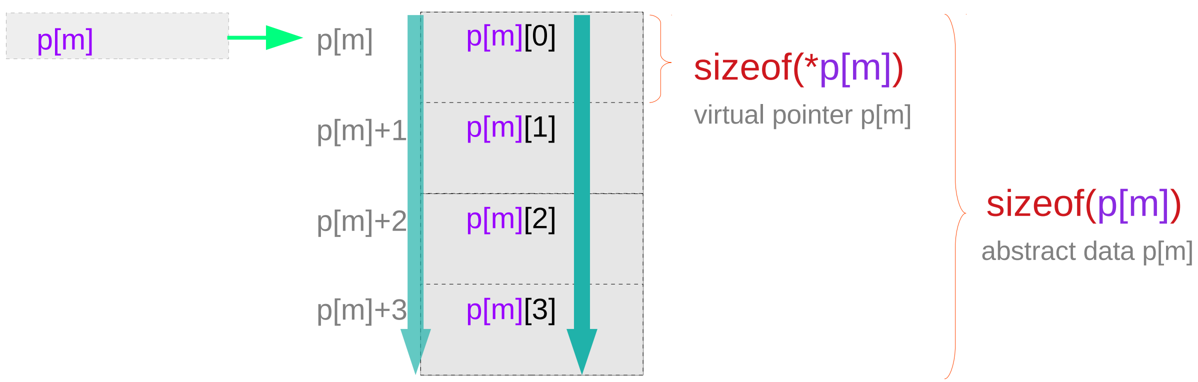
*(p[m]+n) ⟺ p[m][n] **Type 1**

for a given int (*) — continuous N int's

for a given p[m] — contiguous index : **n**

atype p[M][4]; for a given p[m], contiguous p[m][n] : **abstract data types** m = 0, 1, …, M-1

atype (*) atype



can be recursively applied

atype (*)	atype
int (*)	int
int (*) [N]	int [N]
int (*) [M][N]	int [M][N]
int (*) [L][M][N]	int [L][M][N]

Type 2 contiguity constraints (1)

consecutive address — consecutive data

*(X+n) ≡ X[n]

contiguous index : n

(*(p+m))[n]; ⟺ p[m][n]; **Type 2**

for a given p — contiguous index : **m**

pointer type — **abstract data type**

for a given int (*) [N] — continuous M int [N]'s

int **p[M][N];** for a given p, contiguous p[m] : **primitive types**

for a given int * (*) [N] — continuous M int * [N]'s

int * **p[M][N];** for a given p, contiguous p[m] : **pointer types**

for a given atype (*) [N] — continuous M atype [N]'s

atype **p[M][N];** for a given p, contiguous p[m] : **abstract data types**

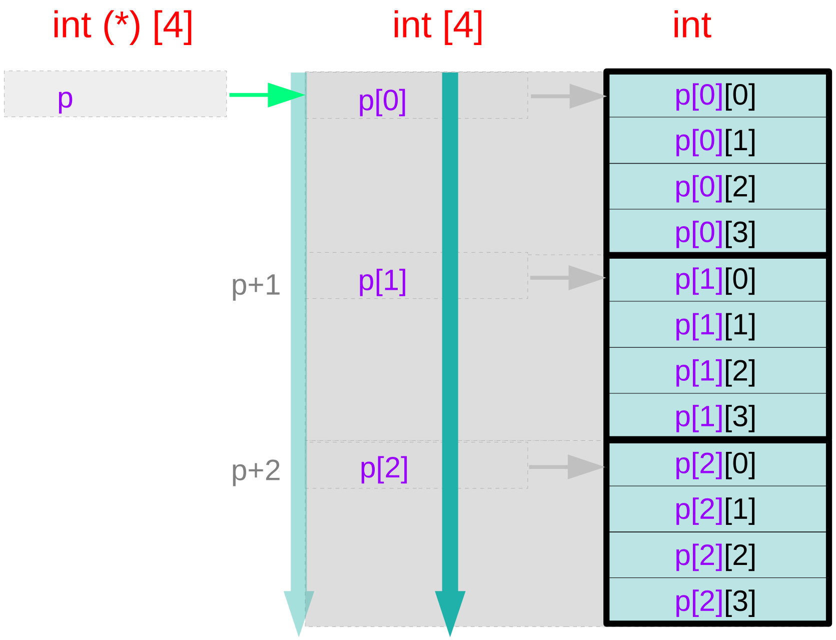
Type 2 contiguity constraints (2)

(*(p+m))[n]; ⟺ p[m][n]; **Type 2**

for a given p

contiguous index : **m**

int p[M][4]; for a given p, contiguous p[m] : **primitive types** m = 0, 1, …, M-1

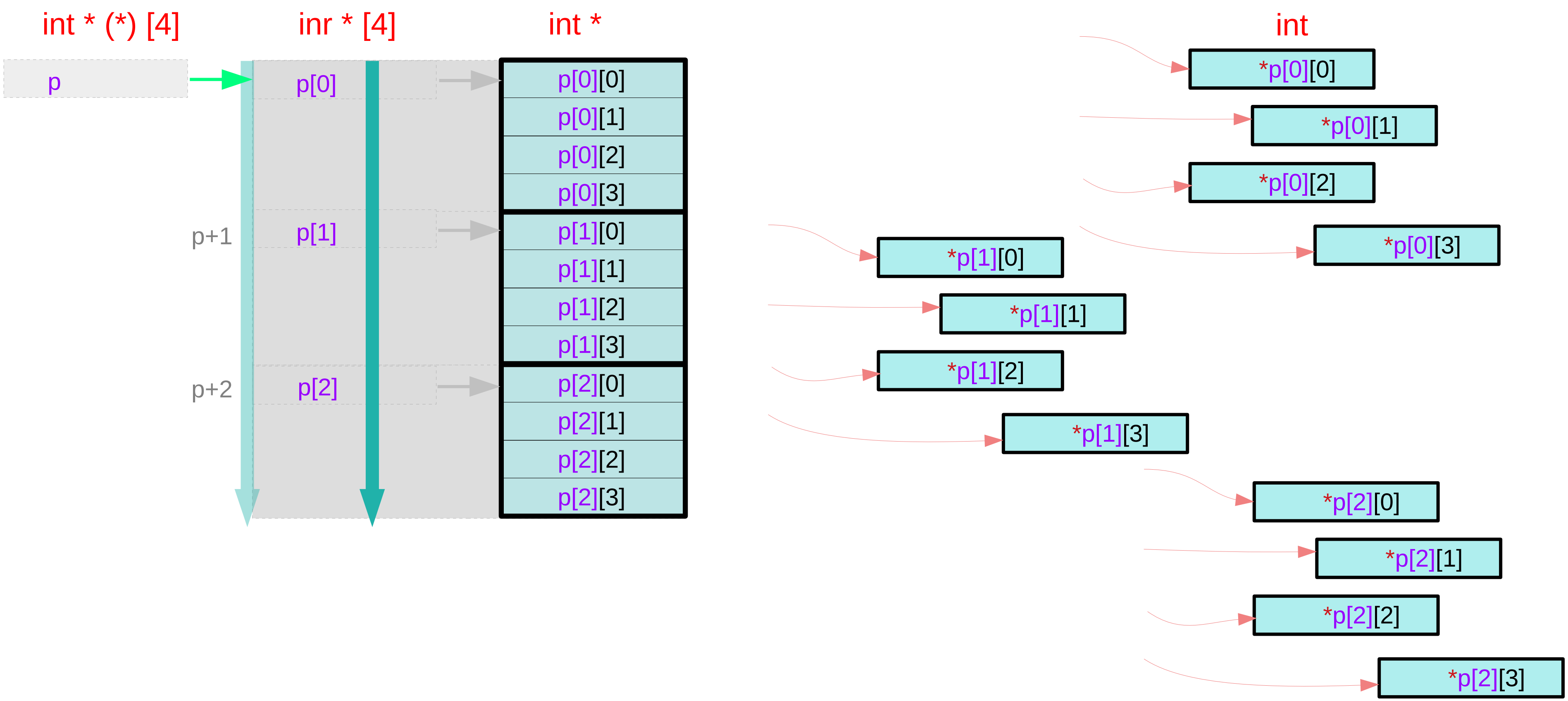


Type 2 contiguity constraints (3)

(*(p+m))[n]; ⟺ p[m][n]; **Type 2**

for a given p   contiguous index : **m**

int * p[M][4]; for a given p, contiguous p[m] : **pointer types**   m = 0, 1, …, M-1



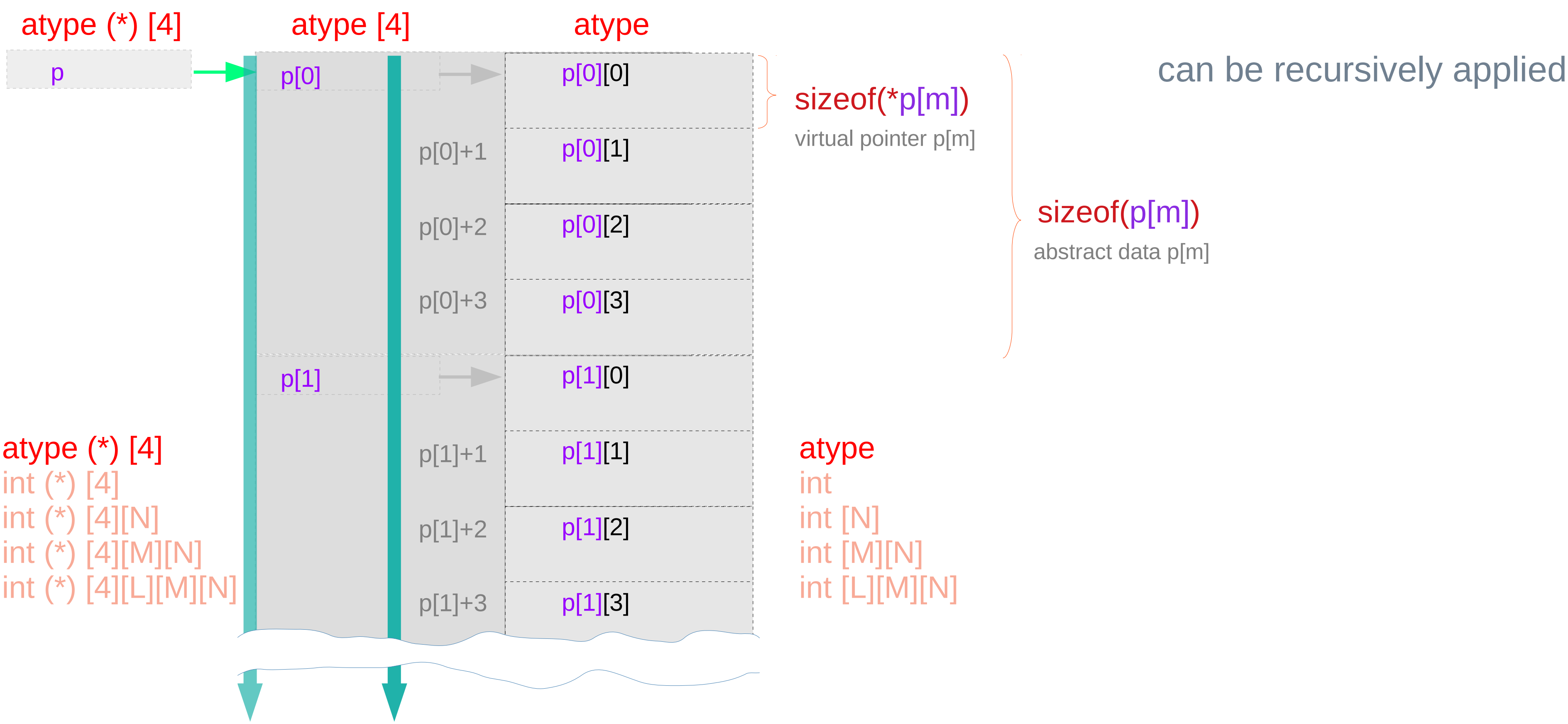
Type 2 contiguity constraints (4)

(*(p+m))[n]; ⟺ p[m][n]; **Type 2**

for a given p

contiguous index : **m**

atype p[M][4]; for a given p, contiguous p[m] : **abstract data types** m = 0, 1, …, M-1



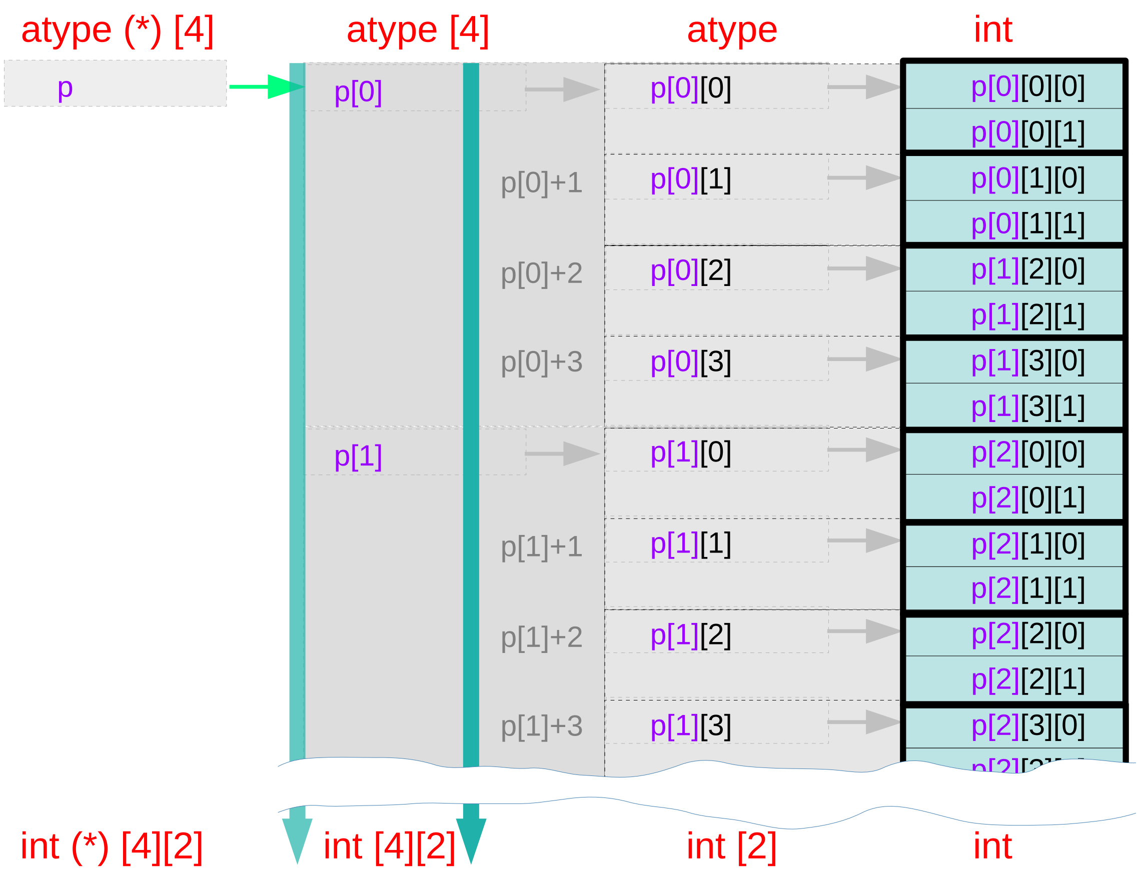
Type 2 contiguity constraints (5)

(*(p+m))[n]; ⟷ p[m][n]; **Type 2**

for a given p

contiguous index : **m**

atype p[M][4]; for a given p, contiguous p[m] : **abstract data types** m = 0, 1, …, M-1



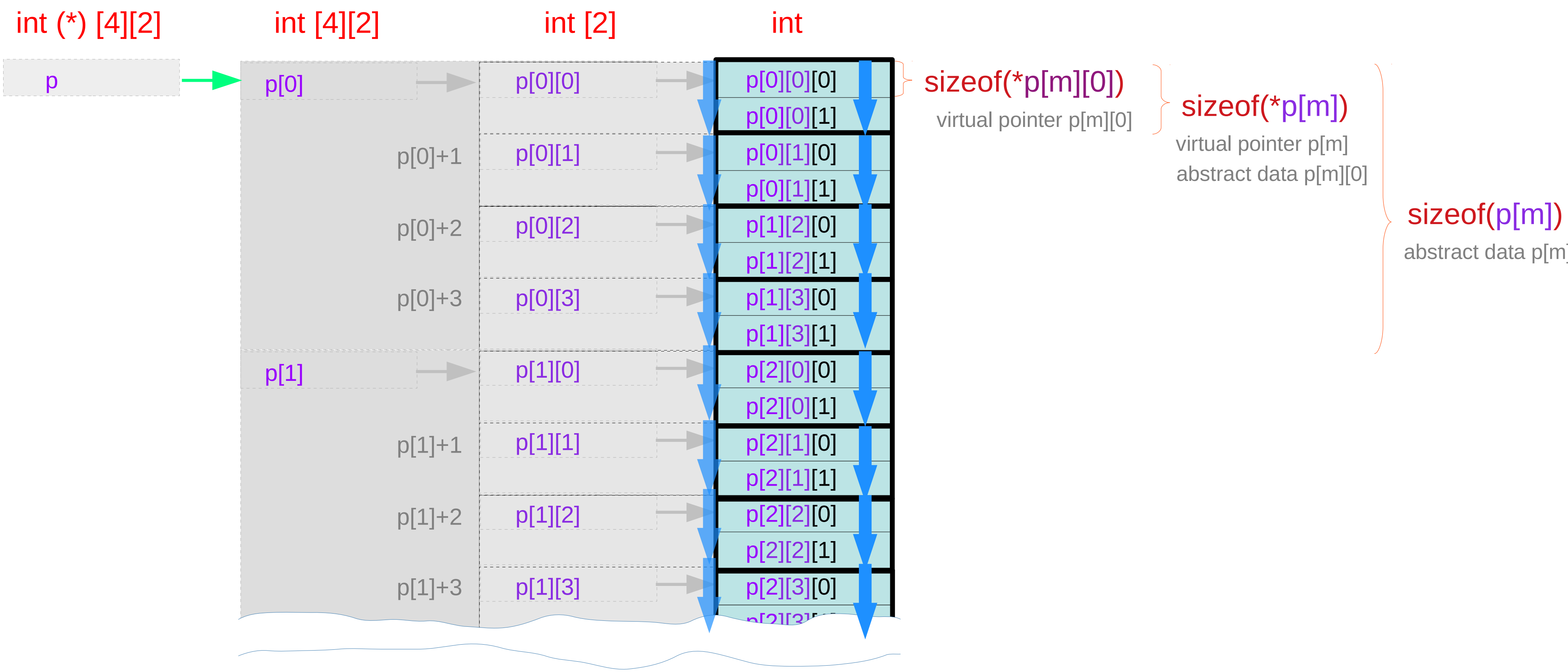
if atype = int [2]

Type 2 contiguity constraints (6)

*(p[m][n]+k) ⟷ p[m][n][k]

for a given p[m][n] contiguous index : **k**

atype p[M][N][2]; for a given p[m][n], contiguous p[m][n][k] : **abstract data types** k = 0, 1

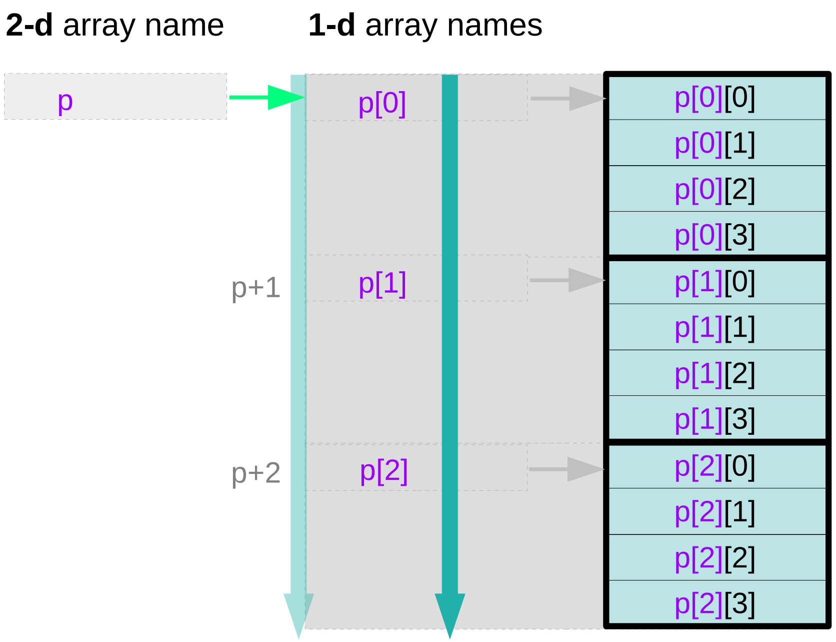


Contiguity constraints for **p** – virtual pointers

(*(p+m))[n]; ⟷ p[m][n];

for a given p

contiguous index : **m**



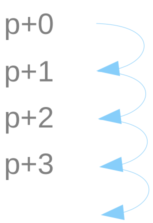
p[0] = &p[0][0]

p = &p[0]

no physical locations

the <u>same</u> addresses

virtual pointer p[0]



sizeof(p[0]) = dual type size = 16 bytes

p	=	&p[0][0]
p+1	=	&p[1][0]
p+2	=	&p[2][0]
p+3	=	&p[3][0]

contiguous p[m] ➡ contiguous p[m][n]

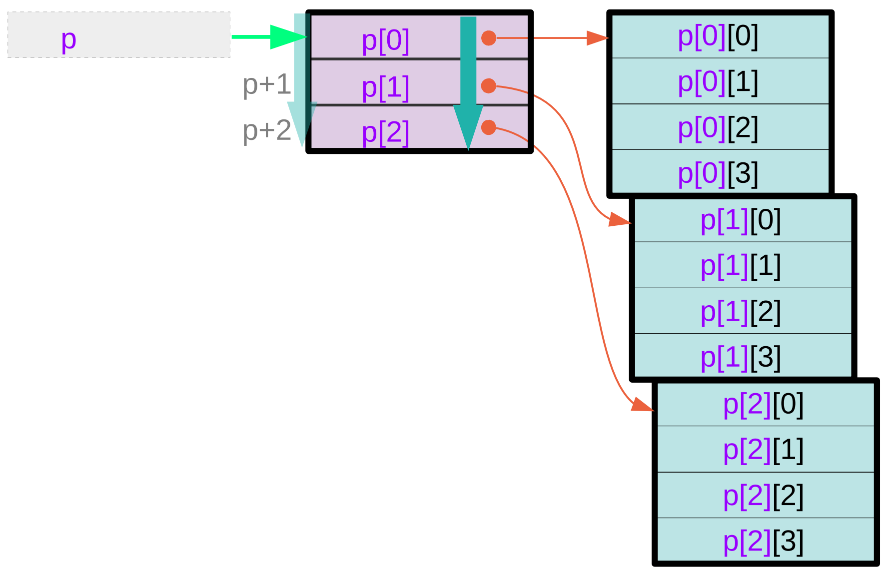
Contiguity constraints for **p** – real pointers

(*(p+m))[n]; ⟺ p[m][n];

for a given p

contiguous index : **m**

1-d array of pointers



p[0] = &p[0][0]

p = &p[0]

the <u>different</u> physical locations

the <u>different</u> addresses

real pointer p[0]

p+0
p+1
p+2
p+3

sizeof(p[0]) = size of a pointer = 4 / 8 bytes

p	≠	&p[0][0]
p+1	≠	&p[1][0]
p+2	≠	&p[2][0]
p+3	≠	&p[3][0]

contiguous p[m] ➡ contiguous p[m][n]

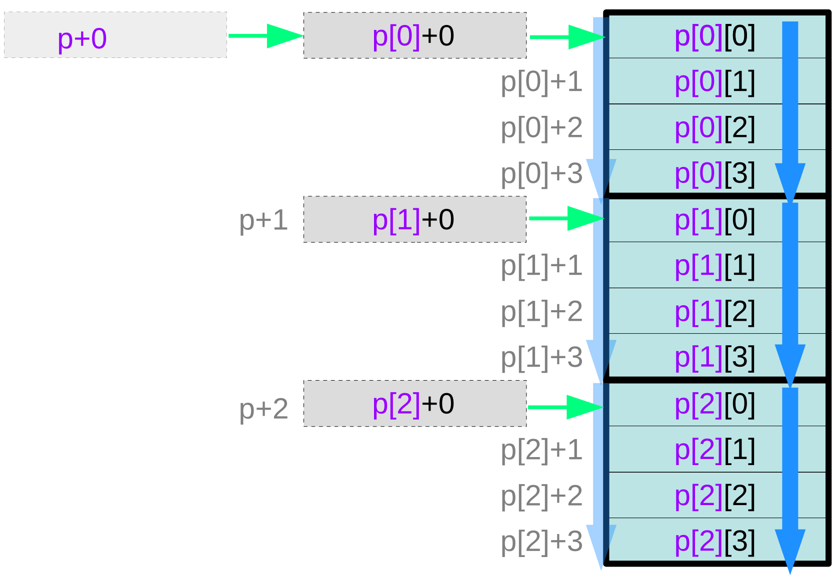
Not necessarily

Contiguity constraints for **p[m]** – virtual pointers

*(p[m]+n) ⟺ p[m][n]

for a given p[m] contiguous index : **n**

2-d array name   **1-d** array names



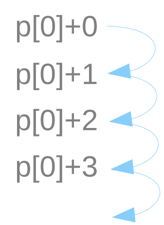
p[0] = &p[0][0]

p = &p[0]

no physical locations

the same addresses

virtual pointer p[0]



sizeof(p[0][0]) = integer size = 4 bytes

the same addresses

contiguous p[m] ➡ contiguous p[m][n]

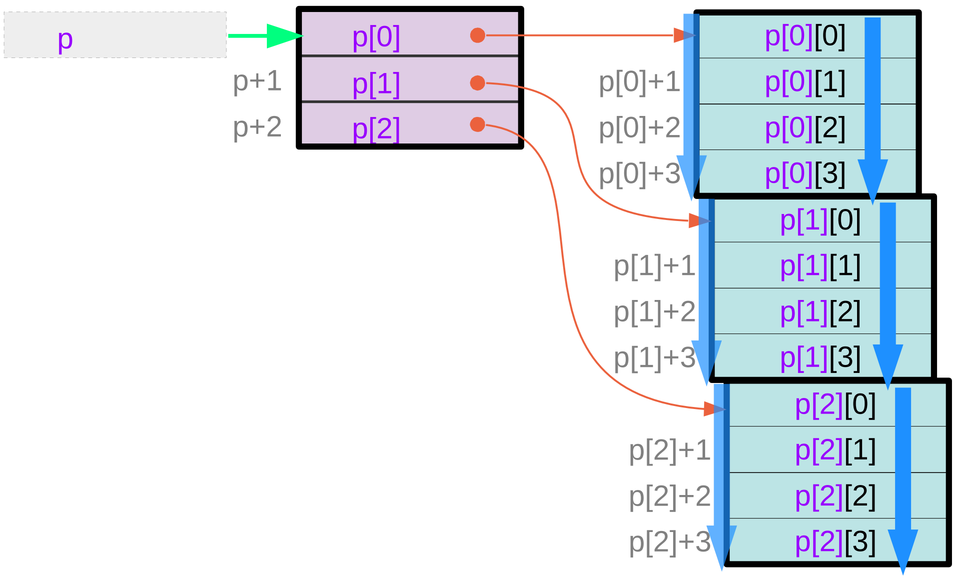
virtual array pointer ⟺ no real memory locations

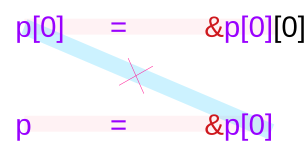
Contiguity constraints for **p[m]** – real pointers

*(p[m]+n) ⟺ p[m][n]

for a given p[m] contiguous index : **n**

1-d array of pointers

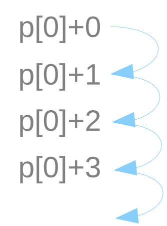




the different physical locations

the different addresses

real pointer p[0]



sizeof(p[0][0]) = integer size = 4 bytes

contiguous p[m] ➡ contiguous p[m][n]

Not necessarily

2-d array accessing expression **a[i][j], b[i][j], c[i][j]**

```
int a[M][N] ;
```

Multi-dimensional Array

a set of pointer assignments are necessary

```
int (*b)[N] ;
int   d[N] ;
```

Array Pointer

```
b = d ;
```

```
int * c[M] ;
int   e[M*N] ;
```

Pointer Array

```
c[0] = e + 0*N ;
c[1] = e + 1*N ;
 •••
```

*(a+m) ⟺ a[m]

a[0], a[1], … , a[M-1]
are contiguous

*(a[m]+n) ⟺ a[m][n]

a[m][0], a[m][1], … , a[m][N-1]
are contiguous

*(b+m) ⟺ b[m]

b[0], b[1], … , b[M-1]
are contiguous

*(b[m]+n) ⟺ b[m][n]

b[m][0], b[m][1], … , b[m][N-1]
are contiguous

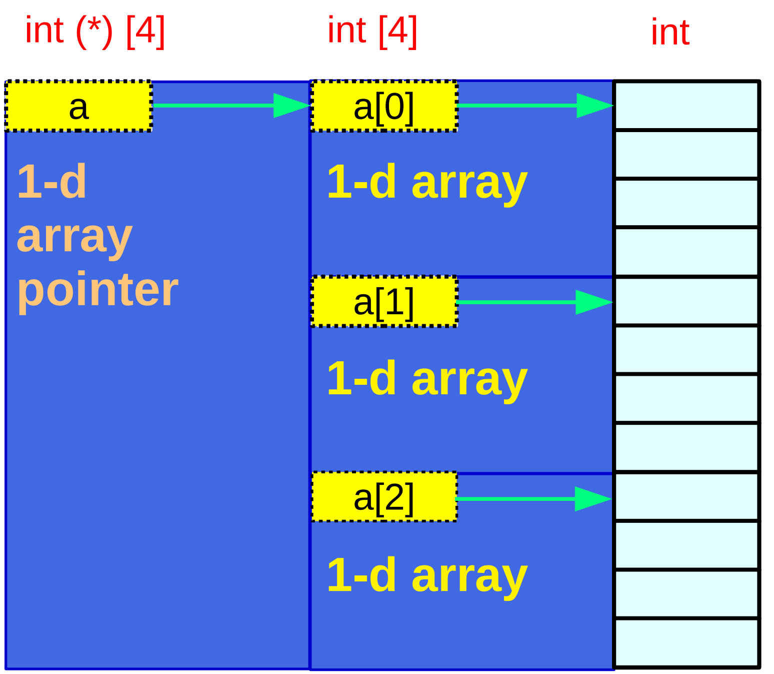
*(c+m) ⟺ c[m]

c[0], c[1], … , c[M-1]
are contiguous

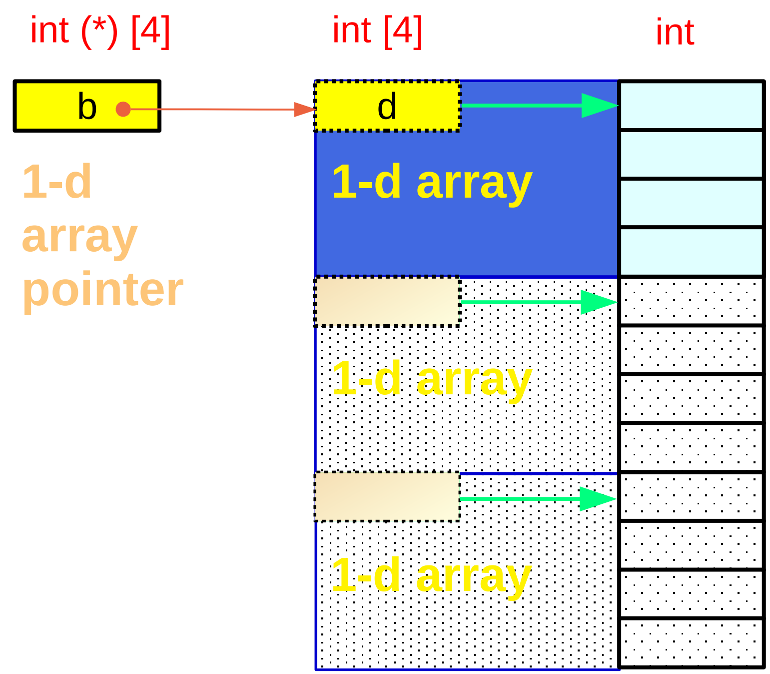
*(c[m]+n) ⟺ c[m][n]

c[m][0], c[m][1], … , c[m][N-1]
are contiguous

Virtual Pointer Arrays vs Array Pointers



Multi-dimensional Array　Virtual Array Pointer



Array Pointer

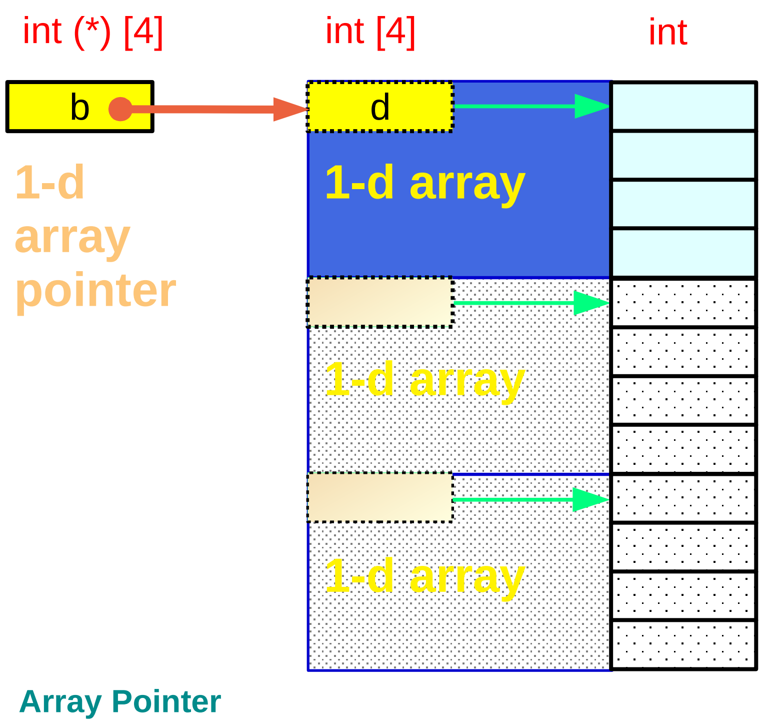
```
int a[M][N] ;
```

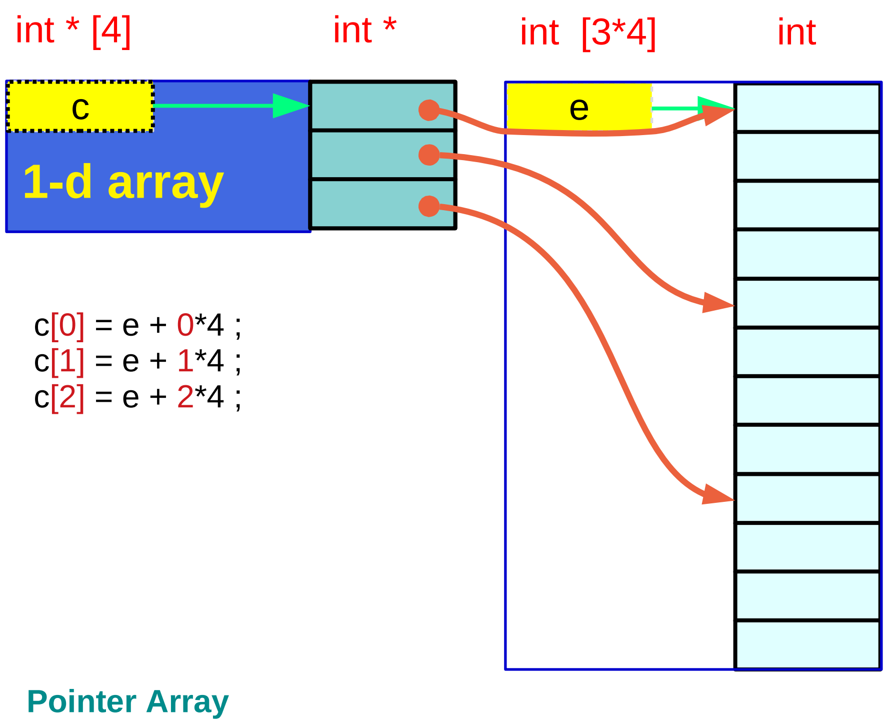
a has a dual type
a is an abstract 2-d array
a is also a virtual pointer to an 1-d array

```
int (*b)[4] ;
```

b is a real pointer to a 1-d array
which has 4 integer elements

Array Pointer vs. Pointer Array (1)






```
int (*b)[4] ;
int    d[4] ;
```

with proper assignments

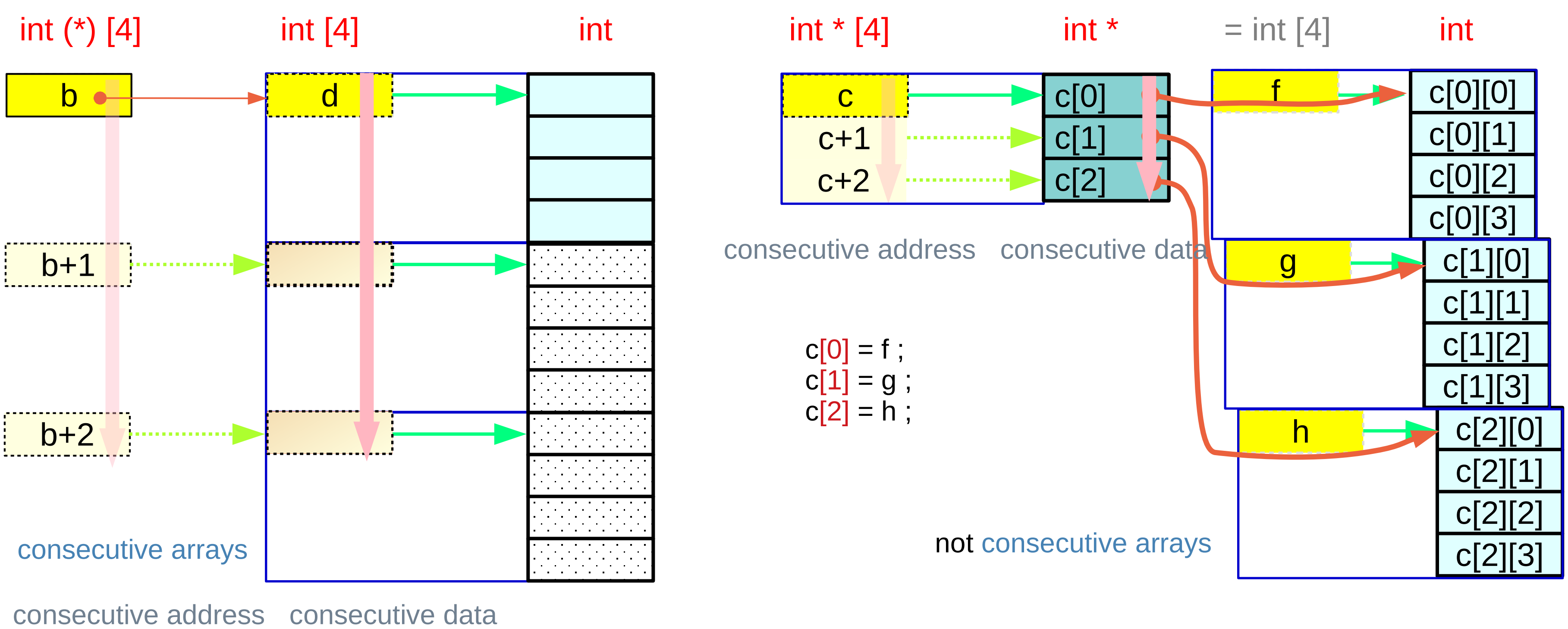
```
int * c[3] ;
int   e[3*4] ;
```

with proper assignments

b is a real pointer to a 1-d array **d**
b has 4 integer elements
b+1 points to the next 1-d array

c is an array of 3 integer pointers
e is an 1-d array and has 3*4 integer elements
c[i]'s divide **e** into 3 parts

Array Pointer vs. Pointer Array (2)




```
int (*b)[4] ;
int   d[4] ;
```

Array Pointer

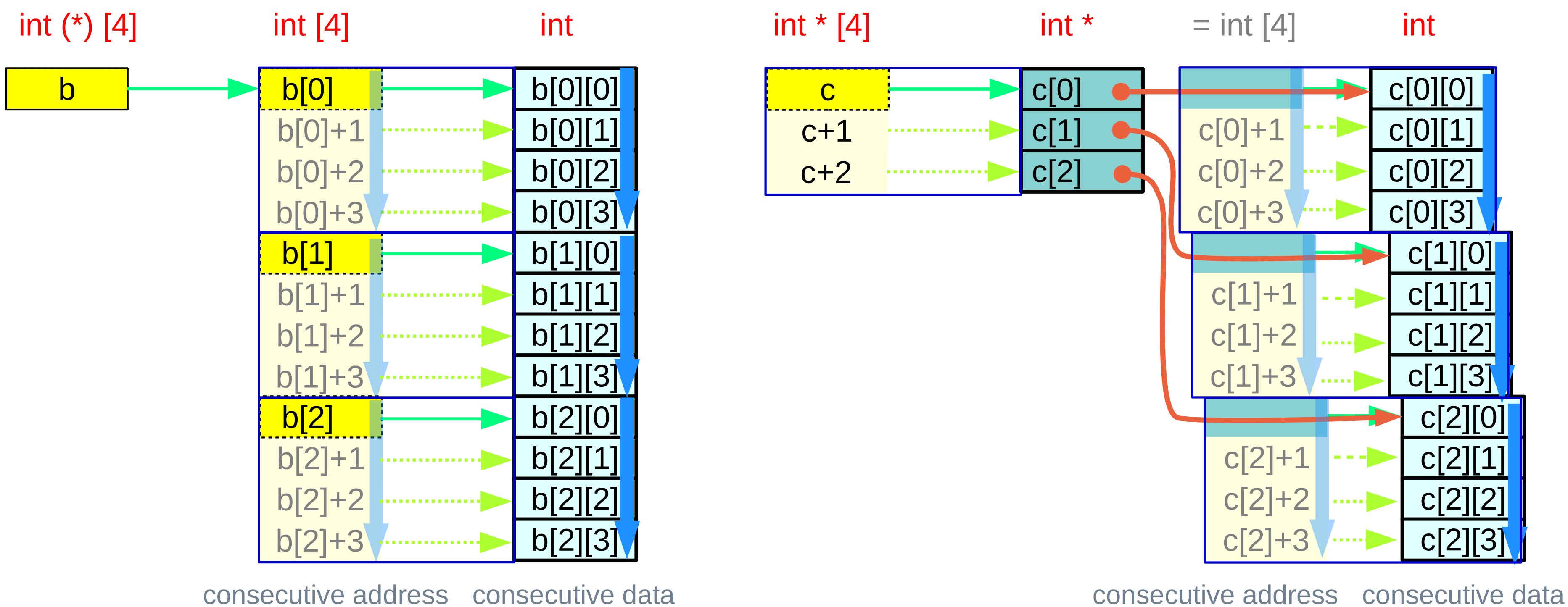
```
int * c[3] ;
int   f[4], g[4], h[4] ;
```

Pointer Array

b has the type of an 1-d array pointer
b[0] = ***b** is the alias of a 1-d array **d**
b[1] is the name of the next 1-d array

c is an array of 3 integer pointers
c[i] has the type of an integer pointer
c[i] points to the first integer of each 1-d array

Array Pointer vs. Pointer Array (3)




```
int (*b)[4] ;
int   d[4] ;
```
Array Pointer

```
int * c[3] ;
int   f[4], g[4], h[4] ;
```
Pointer Array

b[i] is the name of an 1-d array
b[i][j] is the element of such an 1-d array

c[i] can be viewed as the name of an 1-d array
c[i][j] is the element of such an 1-d array

Three contiguity constraints for 3-d arrays

Pointer Array Approach (array of pointers)

c[i][j][k]	⬌	*(c[i][j] +k)
*(c[i][j] +k)	⬌	*(*(c[i] +j) +k)
((c[i] +j) +k)	⬌	*(*(*(c +i) +j) +k)

contiguous **int**	int
contiguous pointers to **int**	int *
contiguous double pointers to **int**	int **

the contiguity constraints are satisfied by allocating arrays of pointers

Array Pointer Approach (pointer to arrays)

c[i][j][k]	⬌	*(c[i][j] +k)
*(c[i][j] +k)	⬌	*(*(c[i] +j) +k)
((c[i] +j) +k)	⬌	*(*(*(c +i) +j) +k)

contiguous **0-d** arrays	int	int
contiguous **1-d** arrays	int [4]	int *
contiguous **2-d** arrays	int [3][4]	int (*) [4]

The contiguity constraints are satisfied by row major ordered linear data layout

Contiguous array pointers c[i][j][k] ≡ *(c[i][j] +k)

c[i][j][k] ⟷ *(c[i][j] +k)

c[i][j]

int [4]	**4** contiguous 0-d arrays
int *	points to the 1st 0-d array
int	0-d array

	[k]
sizeof(**c[i][j]**)	
sizeof(**c[i][j][k]**)	* 4
sizeof(**int**)	* 4

Address Value
c[i][j] + k

&c[i][j][0] + k * sizeof(*c[i][j])
&c[i][j][0] + k * sizeof(c[i][j][0])
&c[i][j][0] + k * 4

*(c[i][j] +k) ⟷ *(*(c[i] +j) +k)

c[i]

int [3][4]	**3** contiguous 1-d arrays
int (*) [4]	points to the 1st 1-d array
int [4]	1-d array

	[j] [k]
sizeof(**c[i]**)	
sizeof(**c[i][j][k]**)	* 3 * 4
sizeof(**int**)	* 3 * 4

Address Value
c[i] + j

&c[i][0][0] + j * sizeof(*c[i])
&c[i][0][0] + j * sizeof(c[i][0])
&c[i][0][0] + j * 4 * 4

((c[i] +j) +k) ⟷ *(*(*(c +i) +j) +k)

c

int [2][3][4]	**2** contiguous 2-d arrays
Int (*) [3][4]	points to the 1st 2-d array
int [3][4]	2-d array

	[i] [j] [k]
sizeof(**c**)	
sizeof(**c[i][j][k]**)	* 2 * 3 * 4
sizeof(**int**)	* 2 * 3 * 4

Address Value
c + i

&c[0][0][0] + i * sizeof(*c)
&c[0][0][0] + i * sizeof(c[0])
&c[0][0][0] + i * 4 * 3 * 4

Contiguous array pointers c[i][j][k] ≡ *(c[i][j] +k)

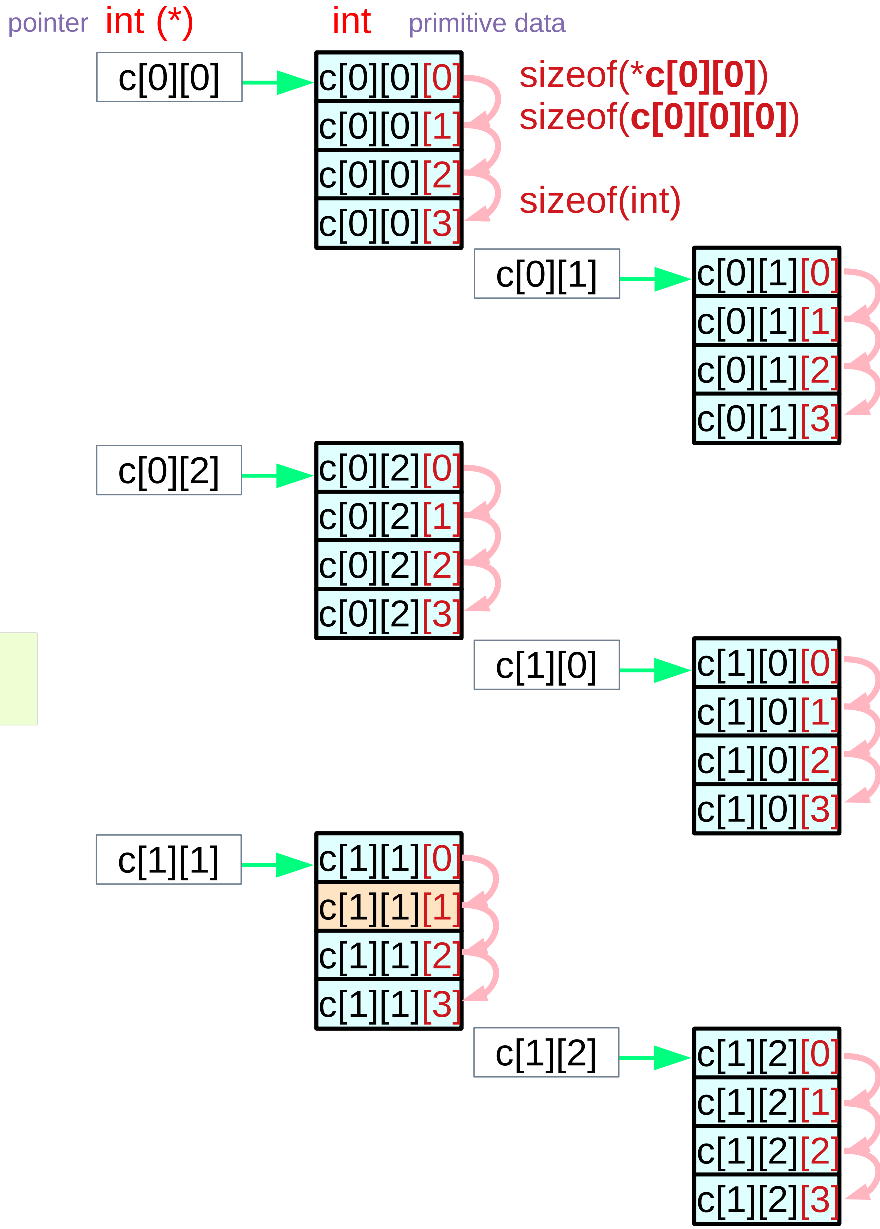
c[0][0][0] =	*(c[0][0] + 0)
c[0][0][1] =	*(c[0][0] + 1)
c[0][0][2] =	*(c[0][0] + 2)
c[0][0][3] =	*(c[0][0] + 3)
c[0][1][0] =	*(c[0][1] + 0)
c[0][1][1] =	*(c[0][1] + 1)
c[0][1][2] =	*(c[0][1] + 2)
c[0][1][3] =	*(c[0][1] + 3)

⋮ ⋮

c[i][j][k] ⟷ *(c[i][j] +k)

contiguous 0-d array

```
int c[2][3][4];
```



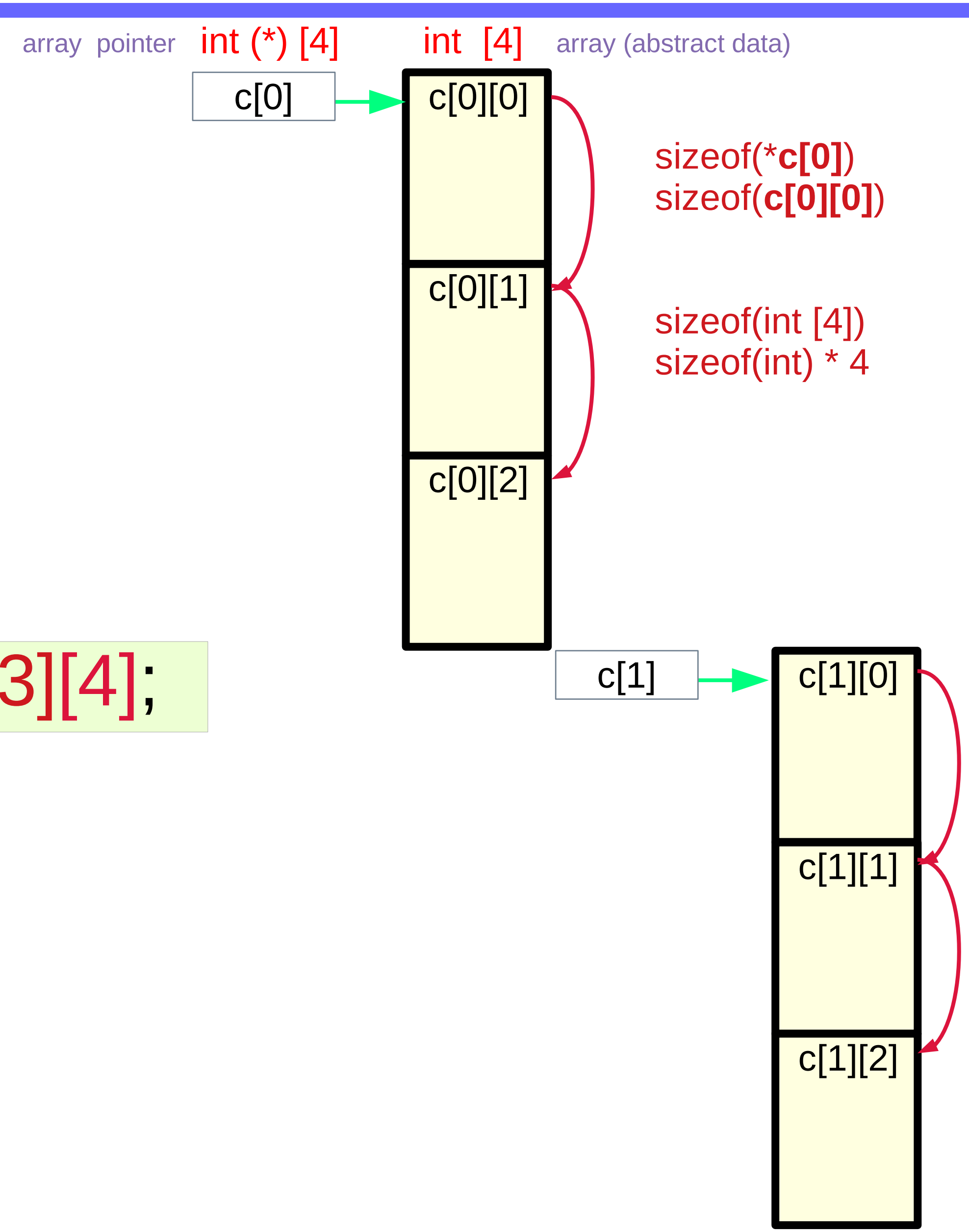
Contiguous array pointers c[i][j] ≡ *(c[i] +j)

c[0][0] =	*(c[0] + 0)
c[0][1] =	*(c[0] + 1)
c[0][2] =	*(c[0] + 2)
c[1][0] =	*(c[1] + 0)
c[1][1] =	*(c[2] + 1)
c[1][2] =	*(c[3] + 2)

*(c[i][j] +k) ⟷ *(*(c[i] +j) +k)

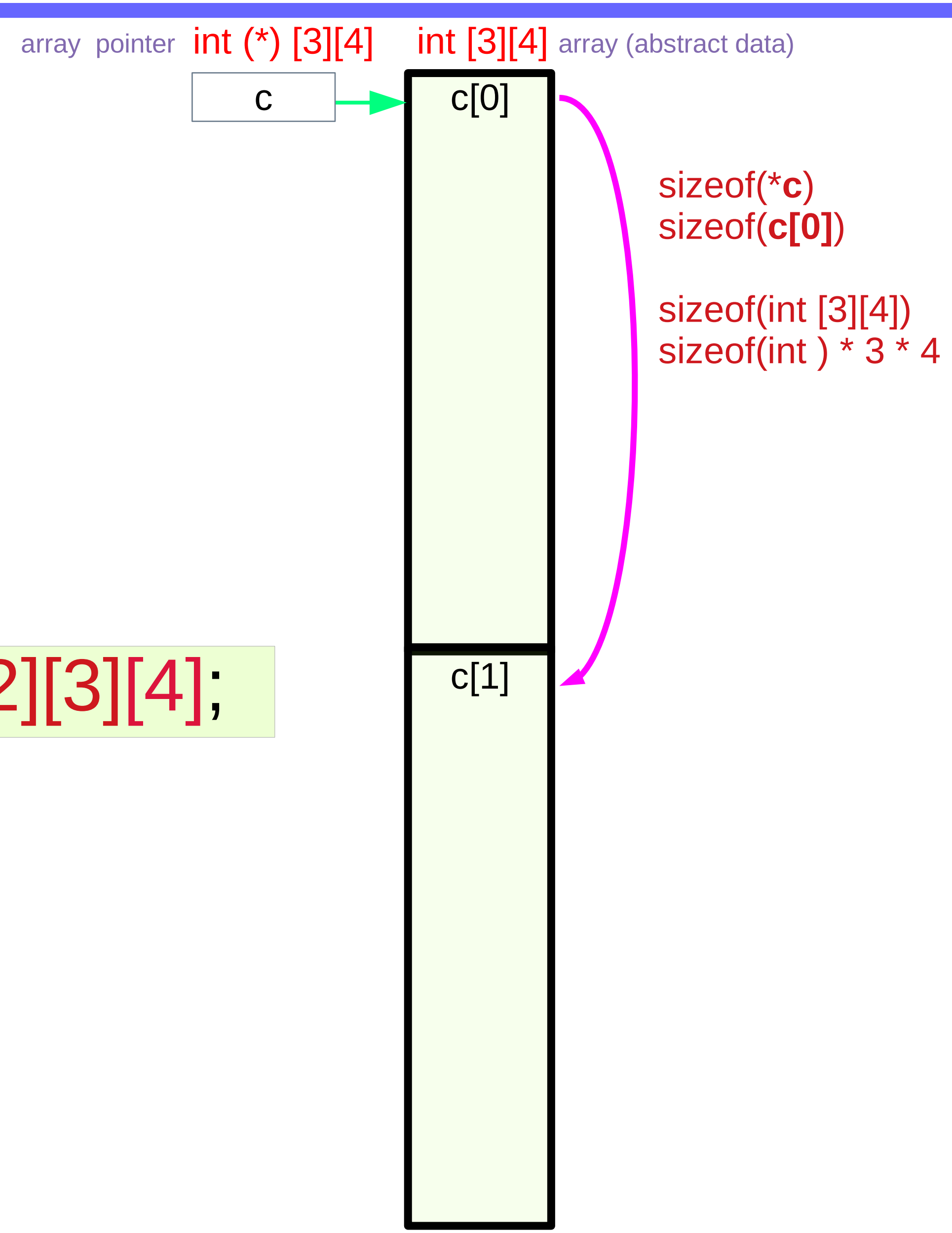
contiguous 1-d arrays

int c[2][3][4];



Contiguous array pointers c[i] ≡ *(c +i)

c[0] =	*(c + 0)
c[1] =	*(c + 1)



((c[i] +j) +k) ⟺ *(*(*(c +i) +j) +k)

contiguous 2-d arrays

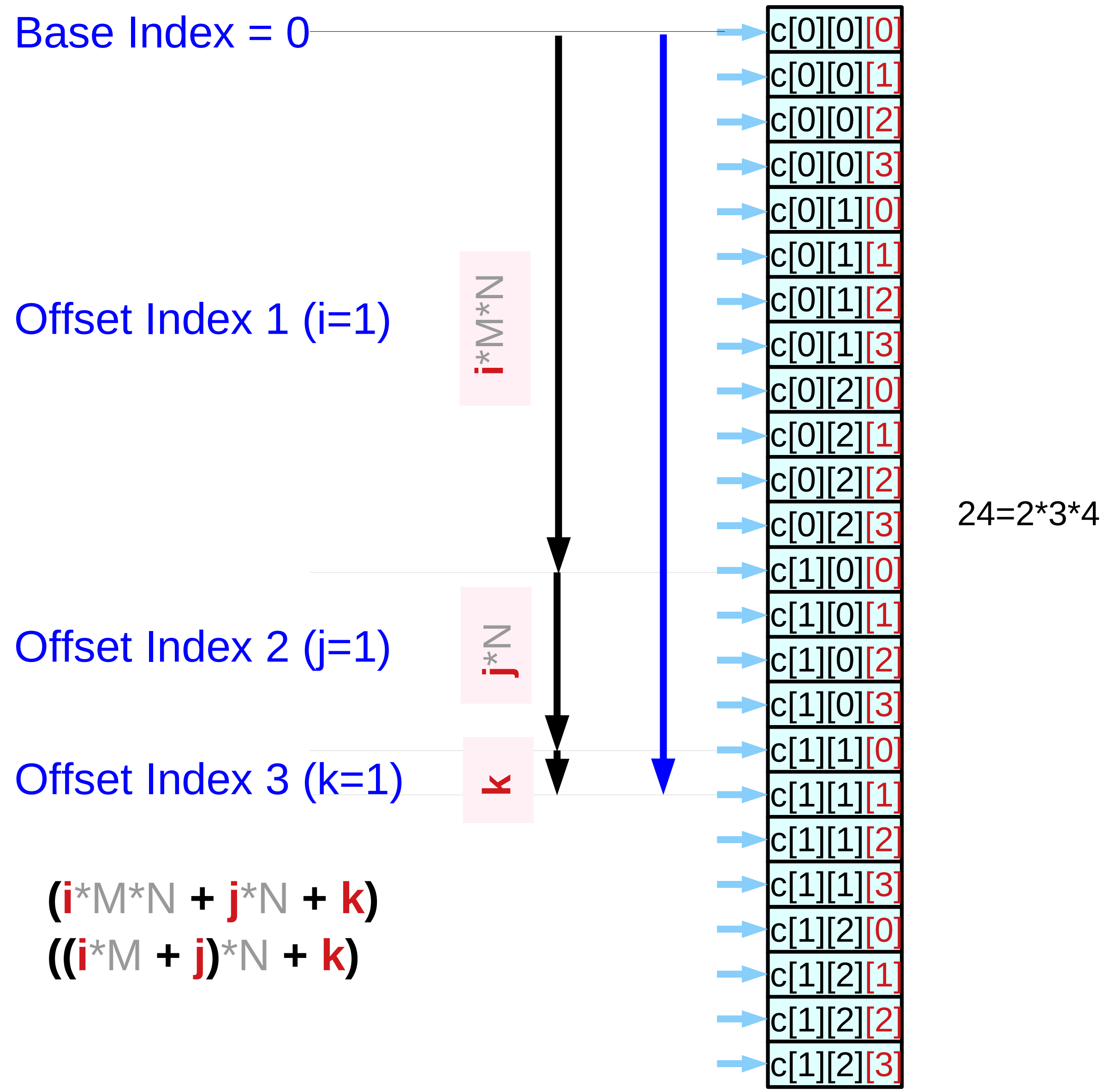
int c[2][3][4];

Contiguous linear layout

```
int    c [L][M][N];

       c [i][j][k];
```

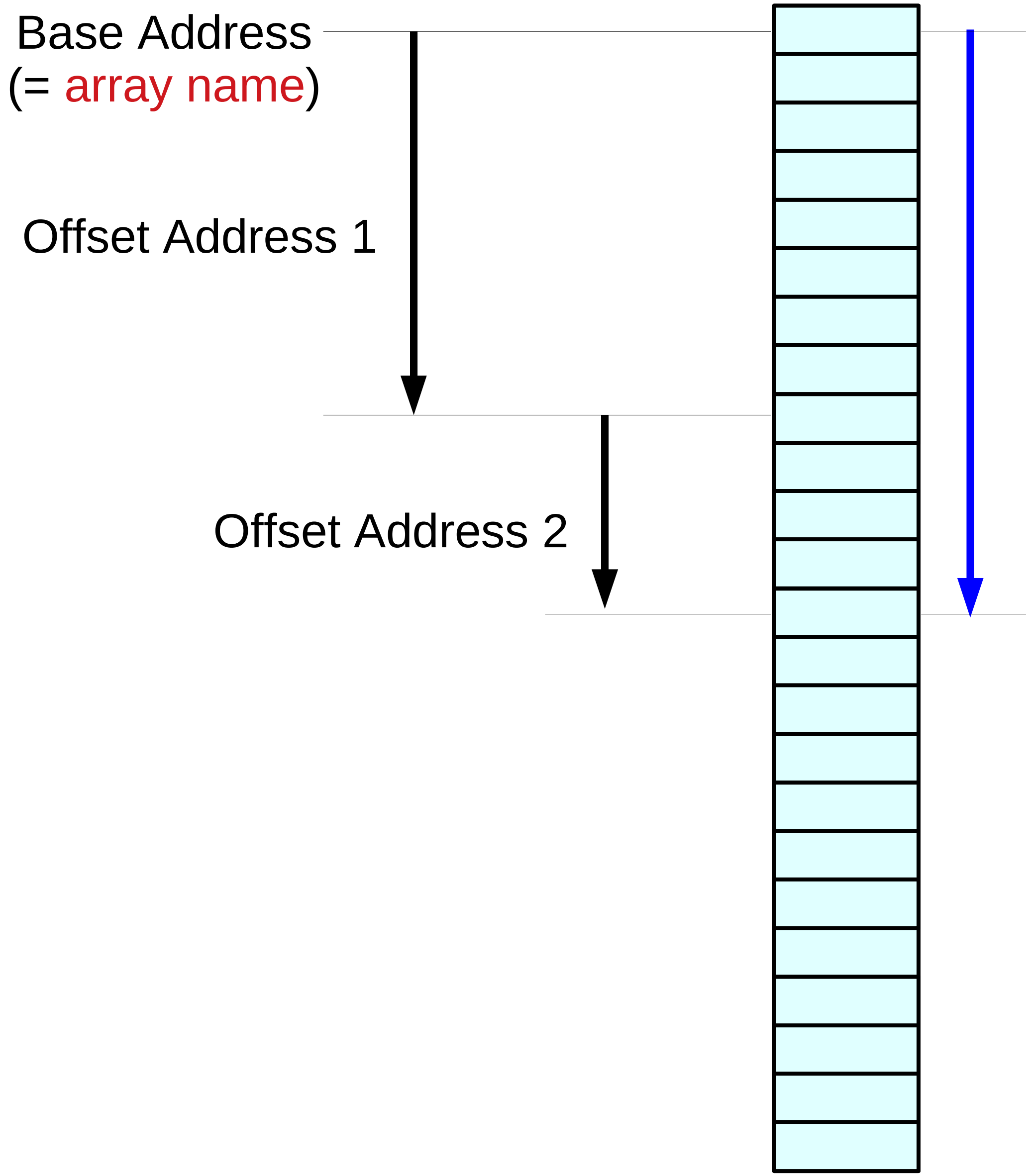
L	M	N
i	**j**	**k**
i*M*N	**j*N**	**k**

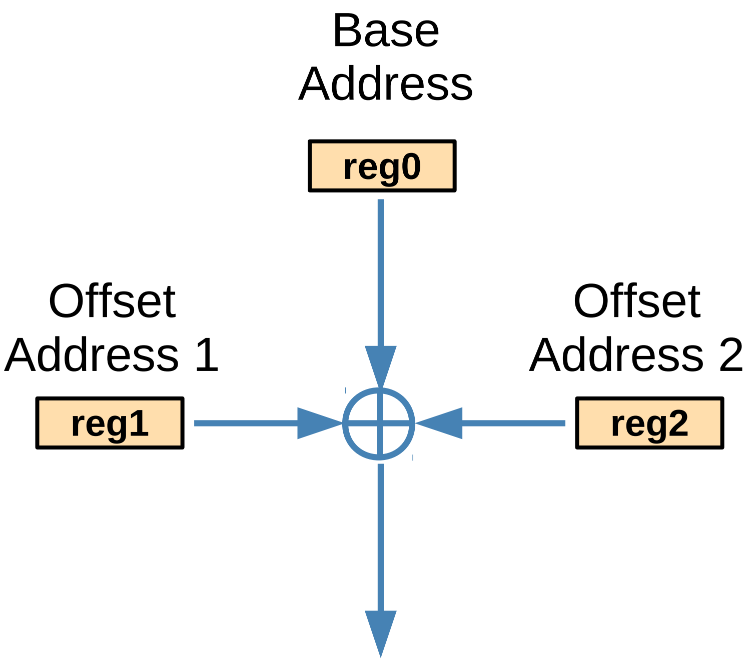


$(i*M*N + j*N + k)$

$((i*M + j)*N + k)$

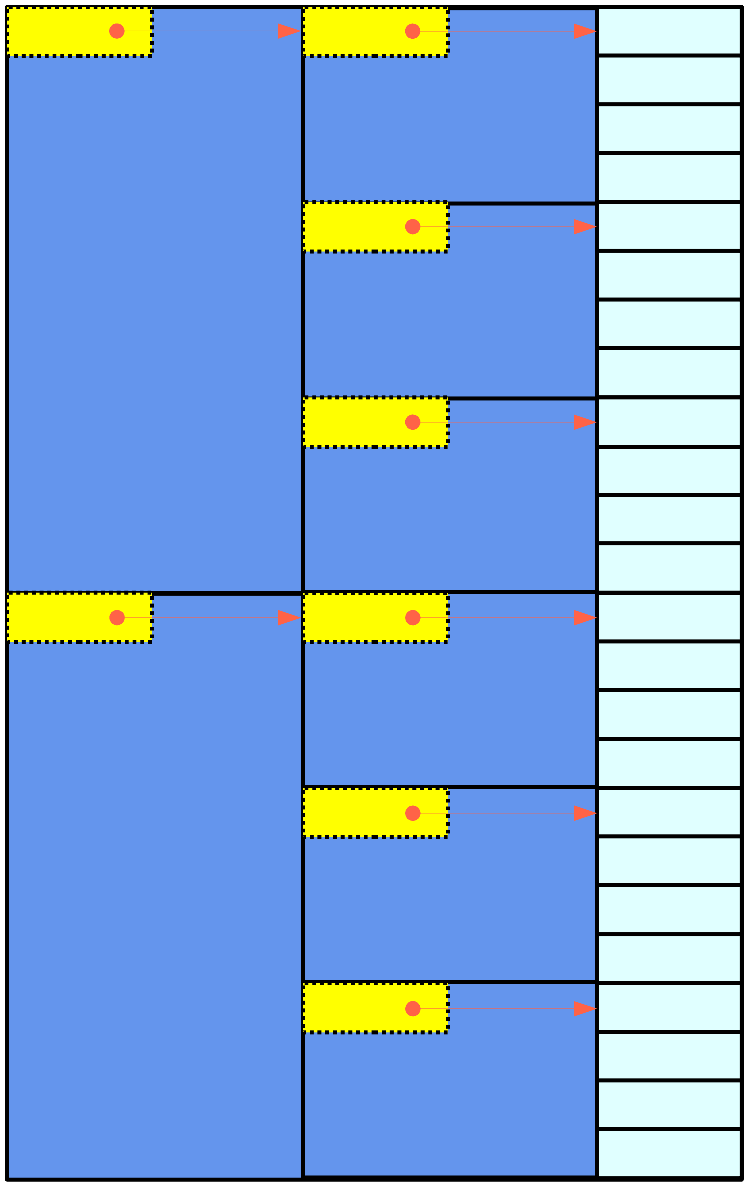
Base and Offset Addressing

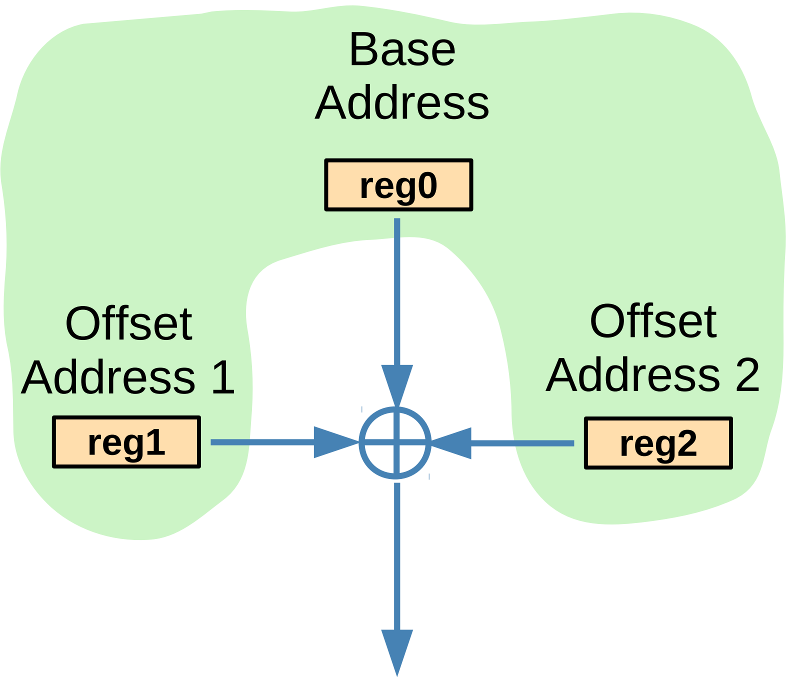




compiler
assembly instruction
registers in the CPU

Array Pointer Approach





register based address **computations**
eliminate the pointer arrays – by a compiler

Array Pointer Approach
(pointer to arrays)

References

[1] Essential C, Nick Parlante
[2] Efficient C Programming, Mark A. Weiss
[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
[4] C Language Express, I. K. Chun