Applications of Pointers (1A)

Young Won Lim
7/17/18

Copyright (c) 2010 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

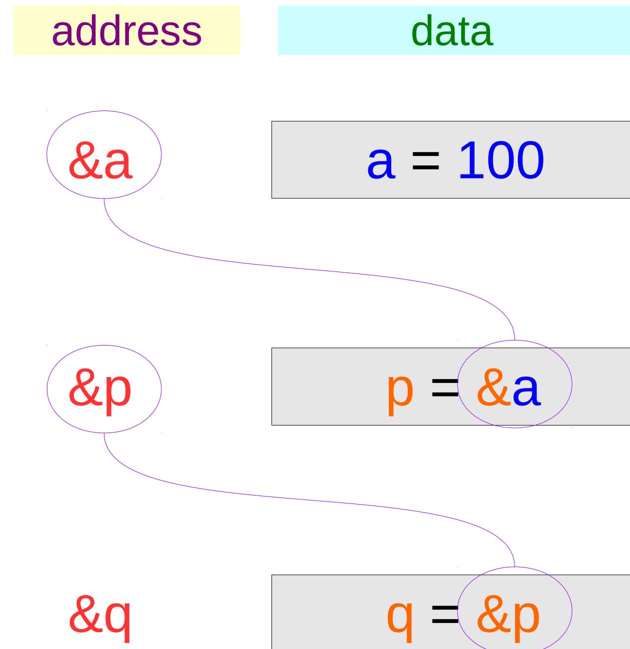
Double Pointers

Variables and their addresses

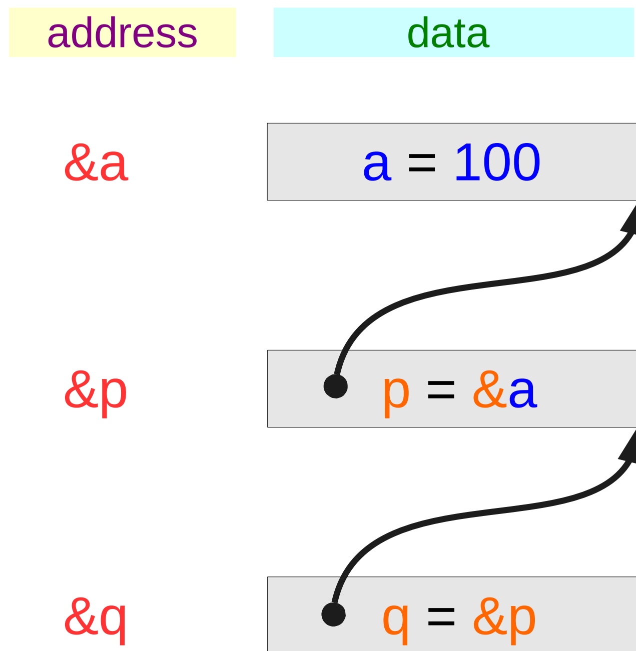
	address	data
int a;	&a	a
int * p;	&p	p
int ** q;	&q	q

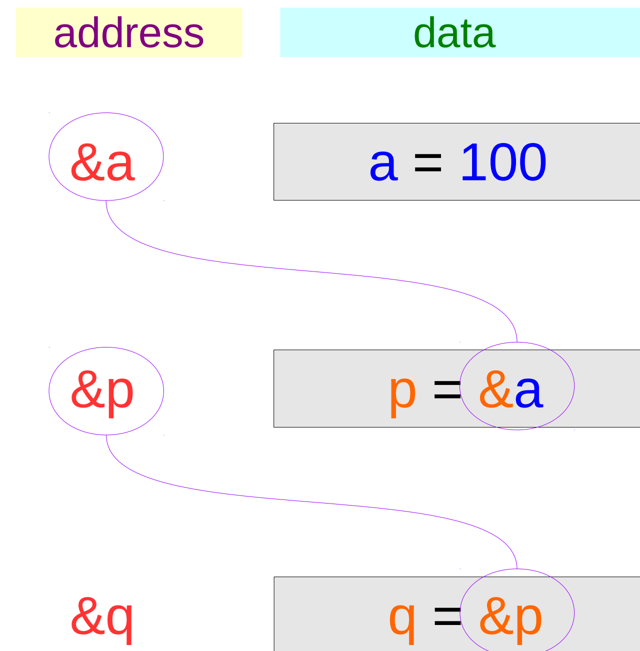
Initialization of Variables

```
int      a = 100;

int   *  p = &a;

int  **  q = &p;
```



Traditional arrow notations

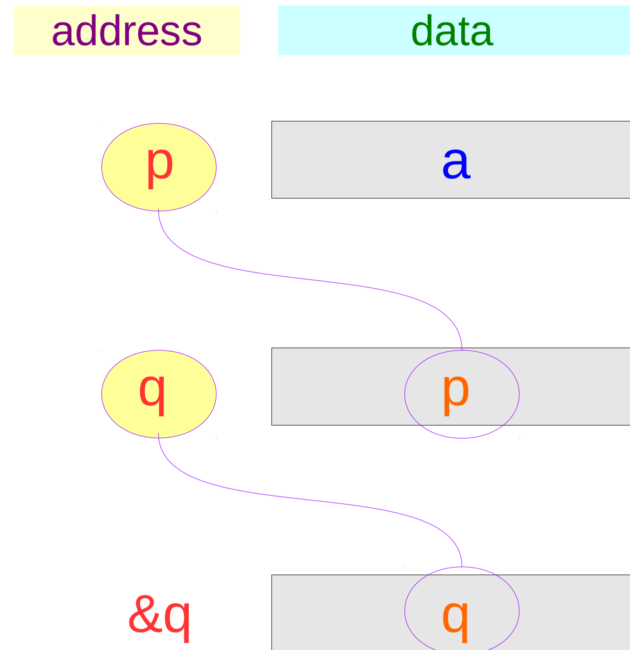




LSB, little endian

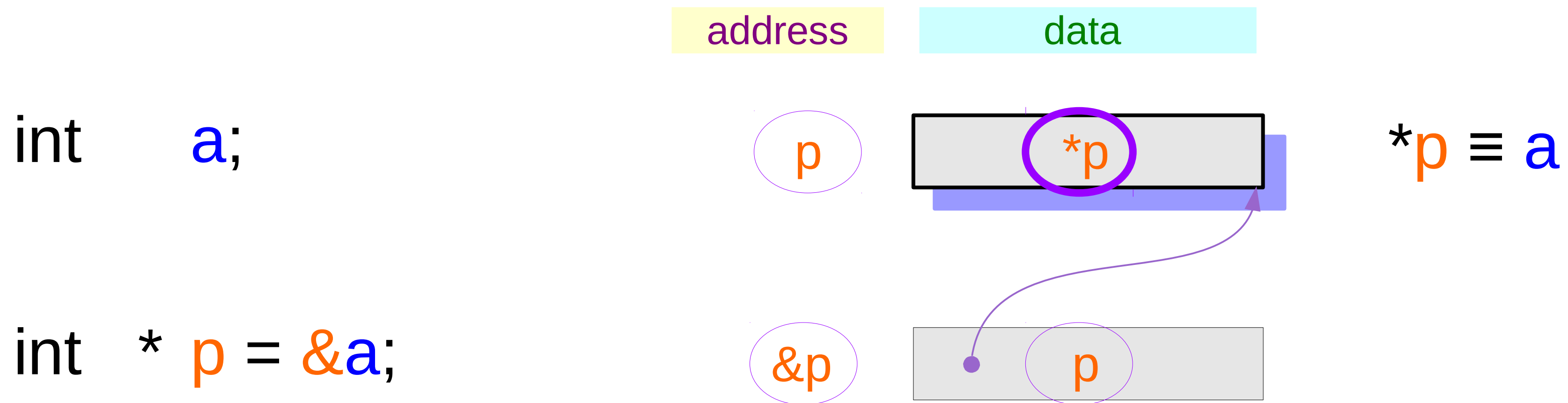
Pointed addresses : p, q

```
int      a;

int   * p = &a;

int  ** q = &p;
```



p = &a
q = &p

A dereferenced variable : *p



An aliased variable : *p

```
int     a;

int   * p = &a;
```

Address assignment

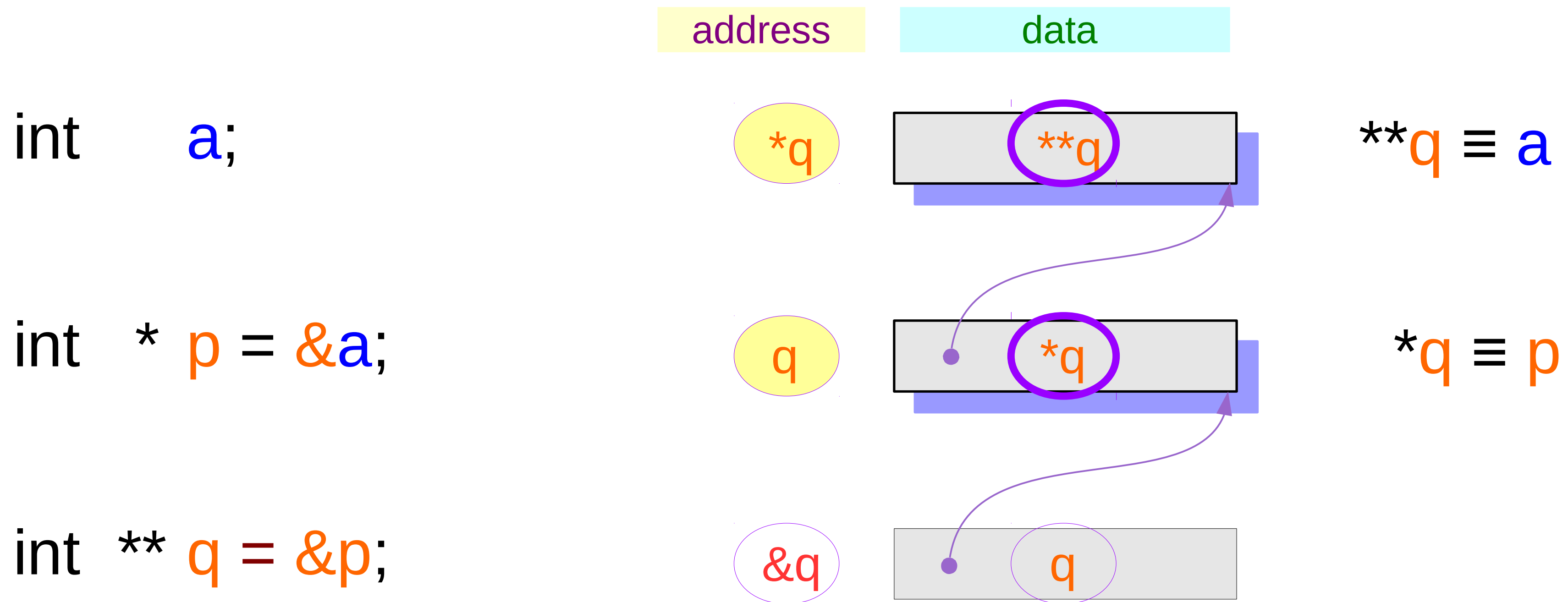
Variable aliasing

p = &a ➡ *p ≡ a

p ≡ &a
*(p) ≡ *(&a)
* p ≡ a

equivalent relations after address assignment

Dereferenced variables : *q, **q



Aliased variables : *q, **q

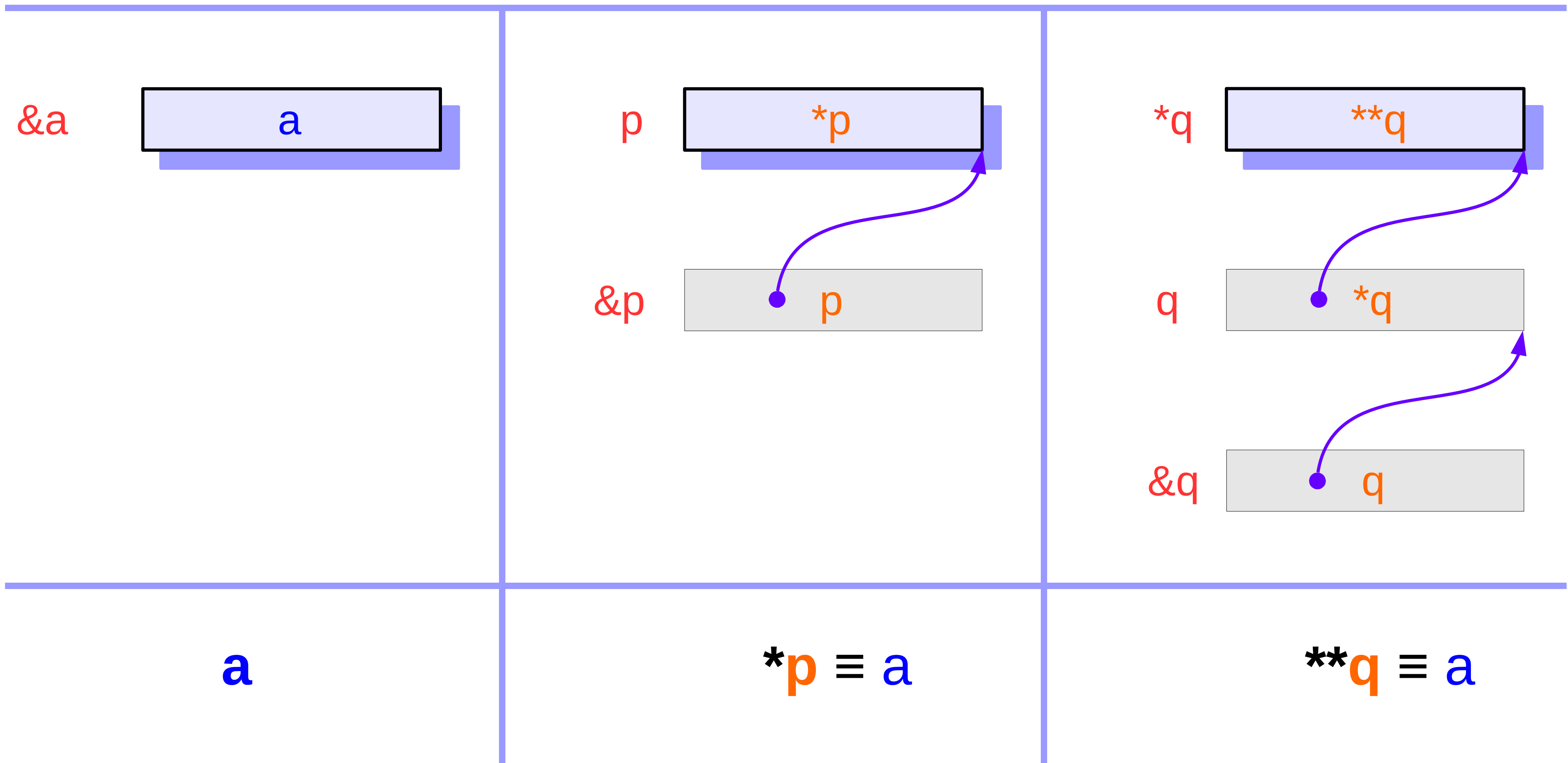
```
int     a;
int   * p = &a;
int  ** q = &p;
```

Address assignment		Variable aliasing
p = &a	➡	*p ≡ a
q = &p	➡	*q ≡ p
	➡	**q ≡ a

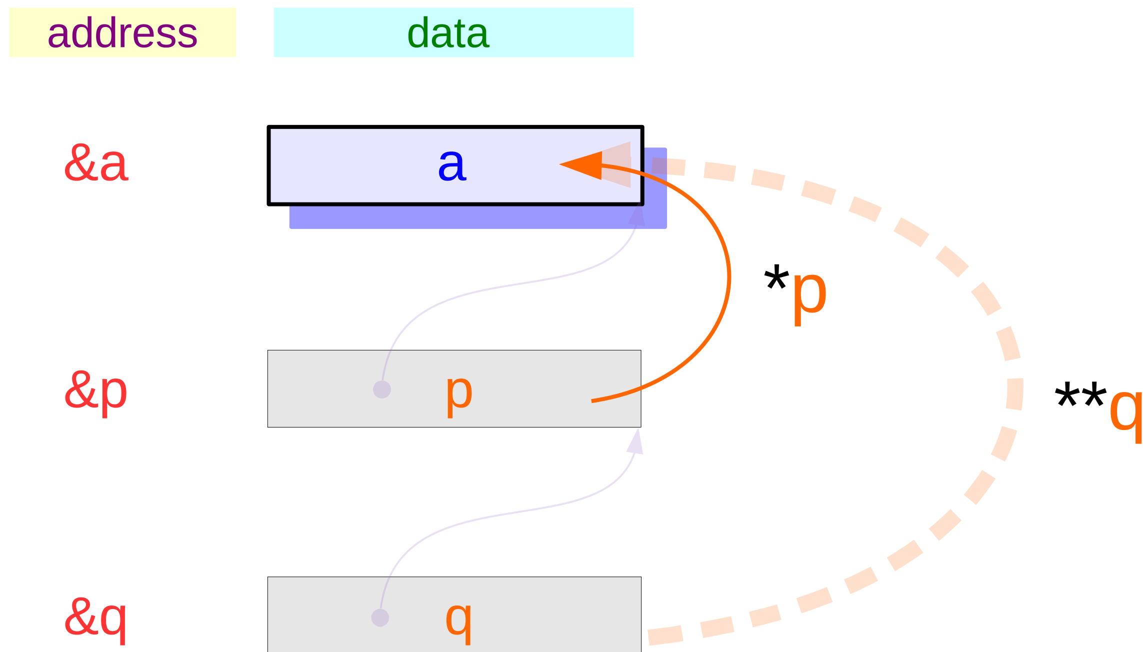
q	≡	&p
*(q)	≡	*(&p)
* q	≡	p
**q	≡	*p
**q	≡	a

equivalent relations after address assignment

Two aliased variables of **a** : ***p**, ****q**



Two more ways to access **a** : *p, **q



1) Read / Write a
2) Read / Write *p
3) Read / Write **q

Variable Definitions

```
int     a;
```

a can hold an *integer*

address	data
&a	a

```
a = 100;
```

a holds 100

address	data
&a	a ← 100

Pointer Variable Definition

int *   p;

p can hold an ***address***

int * p;

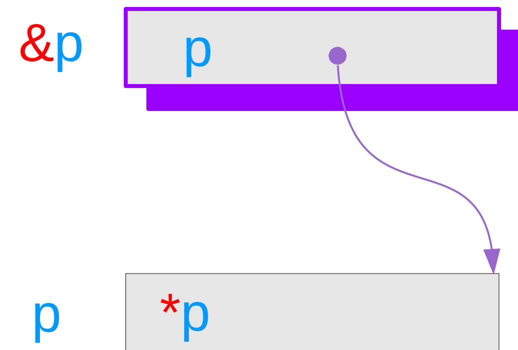
pointer to int

p holds an ***address*** of a int type data

int * p;

int

p** holds a int type ***data



Double Pointer Variable Definition

```
int **      q;
```

q holds an *address*

int ** q;

pointer to pointer to int

q holds an ***address*** of
a pointer to int type data

int * *q;

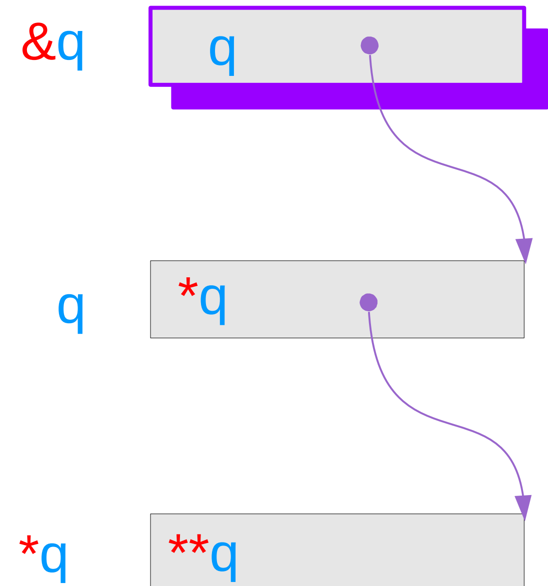
pointer to int

q** holds an ***address of
a int type data

int **q;

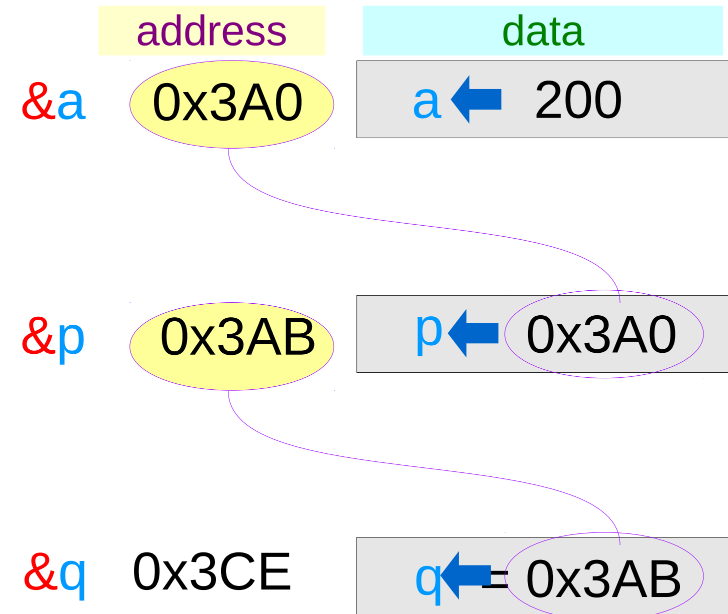
int

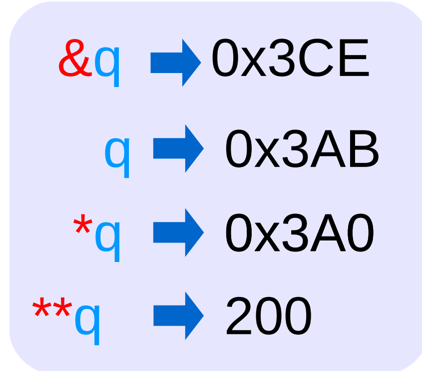
****q** holds a int type ***data***



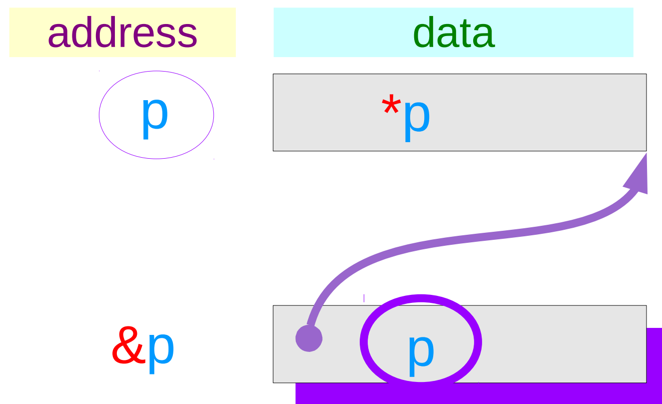
Pointer Variable Examples

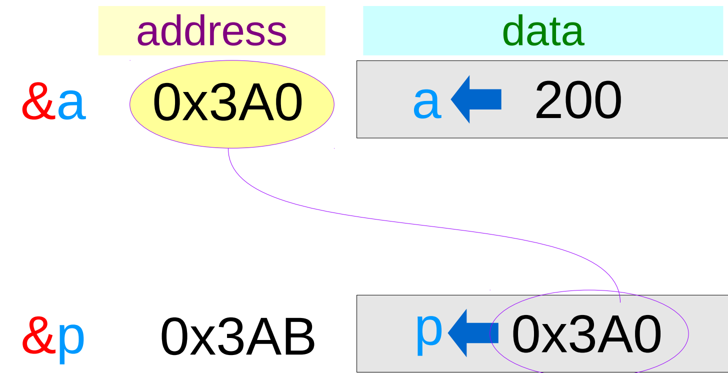
```
int     a = 200;

int *   p = & a;

int **  q = & p;
```





Pointer Variable **p** with an arrow notation

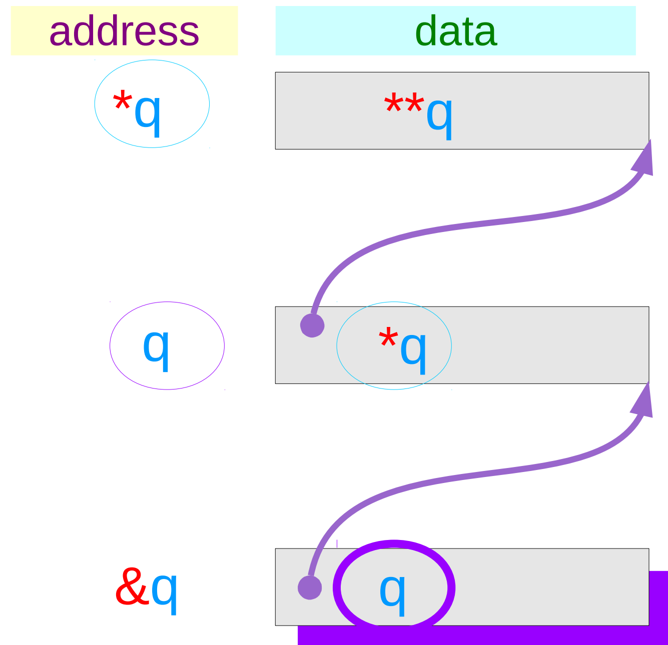




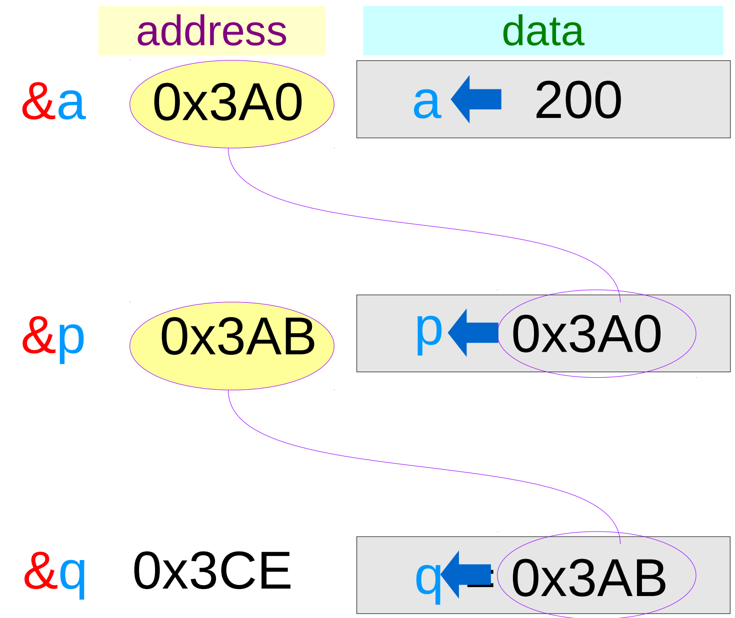
using an arrow notation

&p → 0x3AB
p → 0x3A0
*p → 200

Pointer Variable **q** with an arrow notation



using an arrow notation



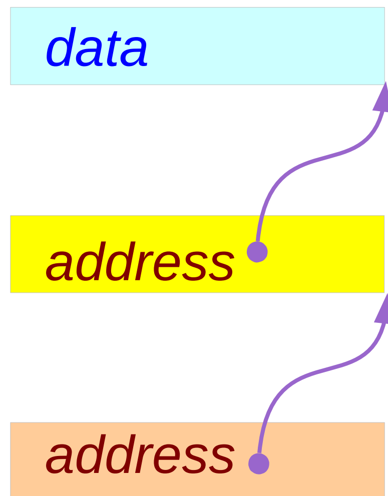
&q ➡ 0x3CE

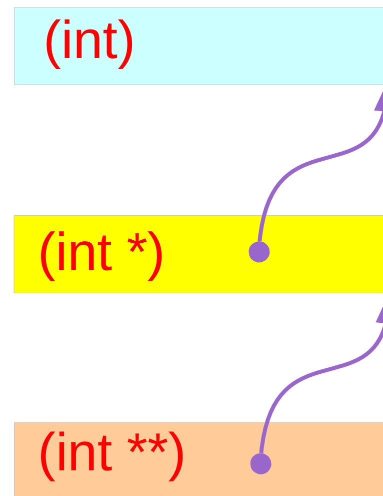
q ➡ 0x3AB

*q ➡ 0x3A0

**q ➡ 200

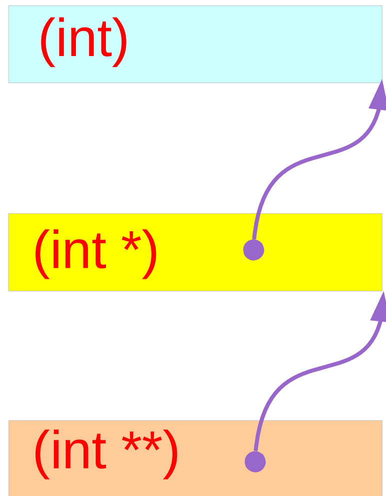
Pointers – a type view

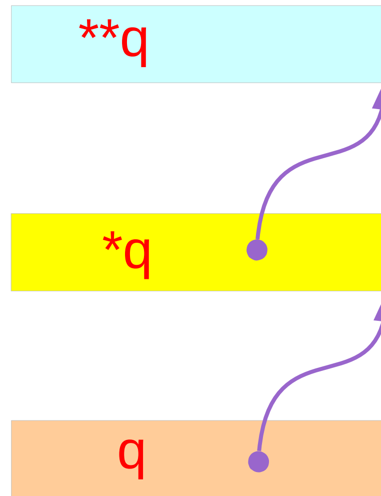


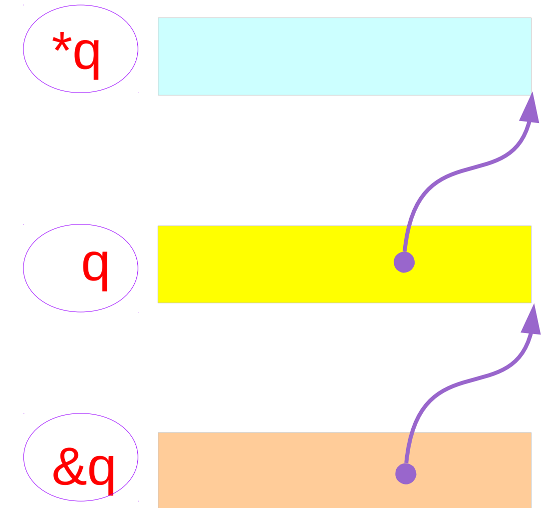


Types

Pointers – other view



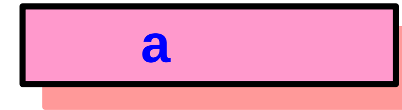


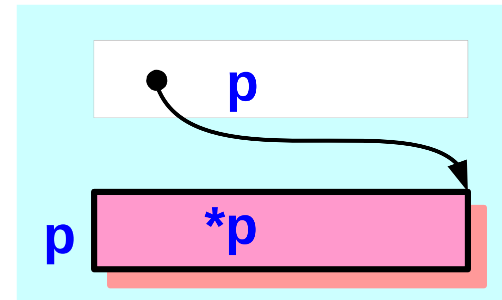


Types **Variables** **Addresses**

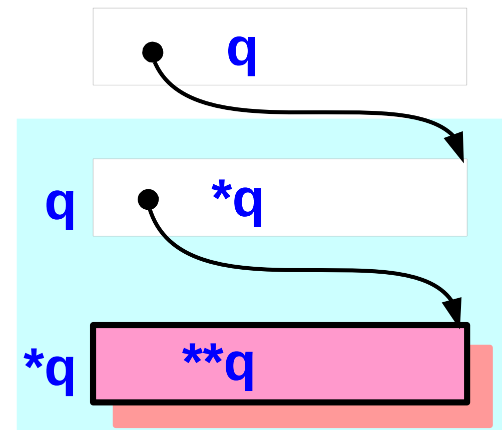
Single and double pointer examples (1)

```
int   a ;
int * p ;
int **q ;
```



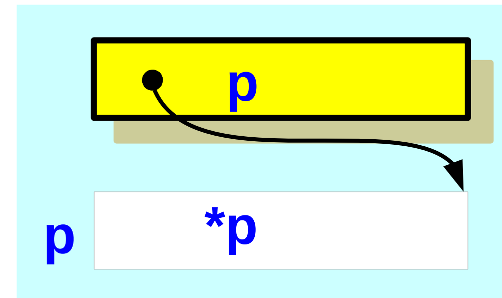


a, ***p**, and ****q**:
int variables

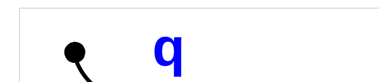


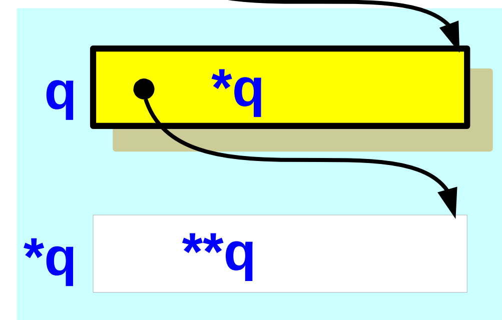
Single and double pointer examples (2)

```
int     a  ;
int *   p  ;
int **  q  ;
```



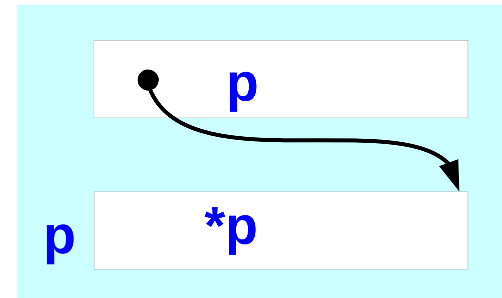
p and ***q :**
int <u>pointer</u> variables
(singlepointers)



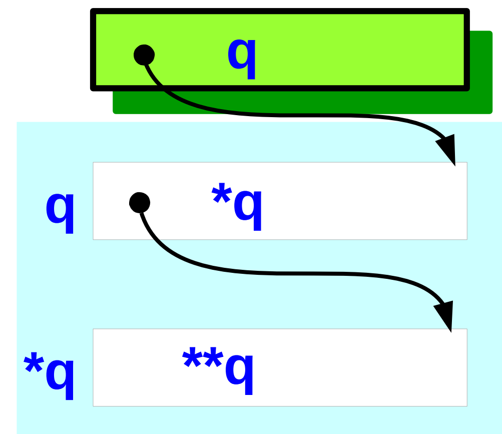


Single and double pointer examples (3)

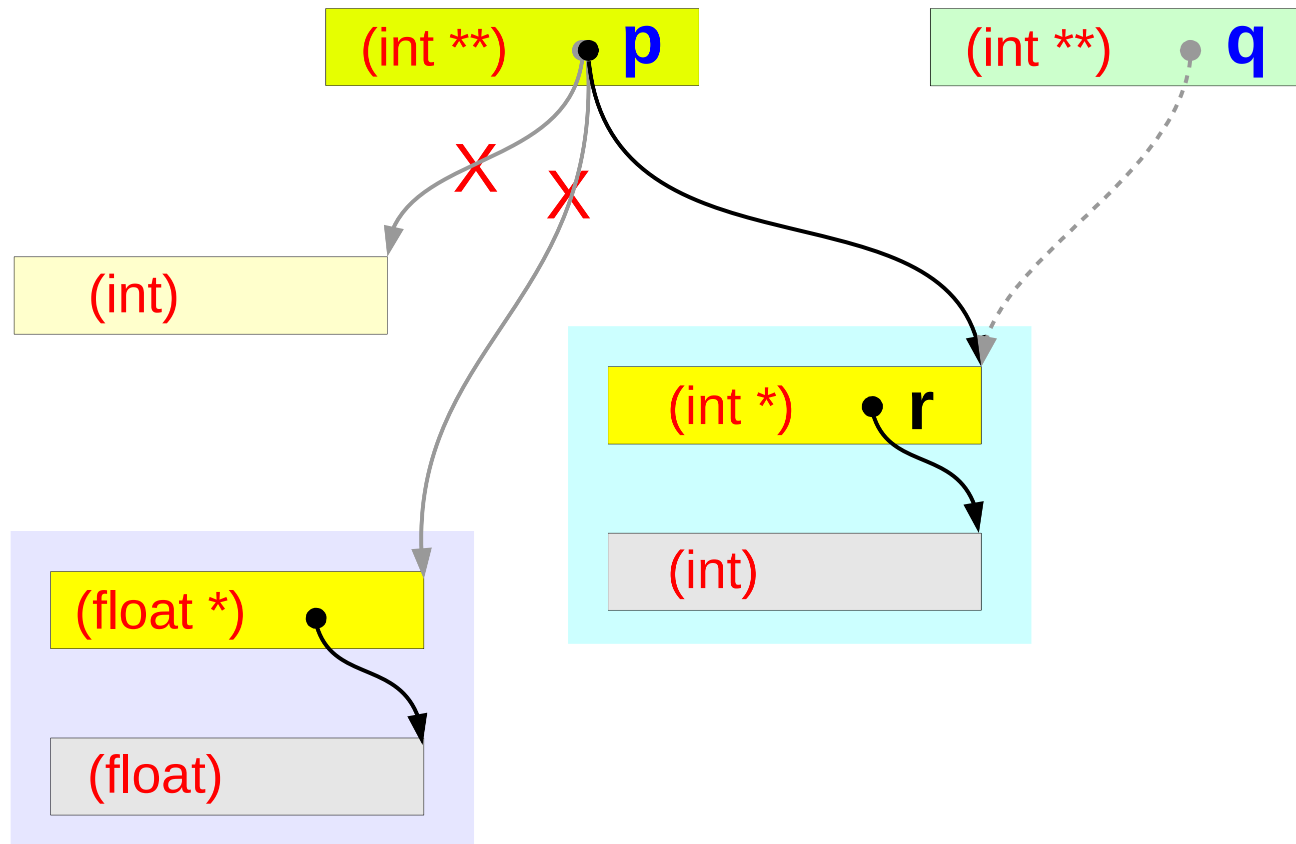
int	a	;
int *	p	;
int **	q	;



q :
double int pointer variables



Double pointer variable assignments



Pointed Addresses and Data

int a ;   &a   a =100

The variable a holds an integer data

int * p ;   &p   p → 200

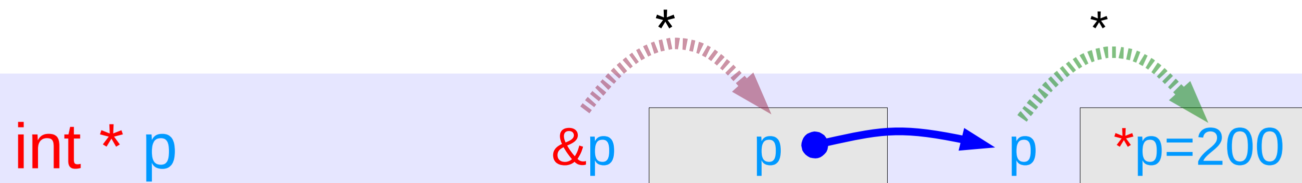
The **pointer** variable p holds an address,
at this address, an integer data is stored

int * * q ;   &q   q → *q → 30

The **pointer** variable q holds an address,
at the address q, another address *q is stored,
at the address *q, an integer data **q is stored

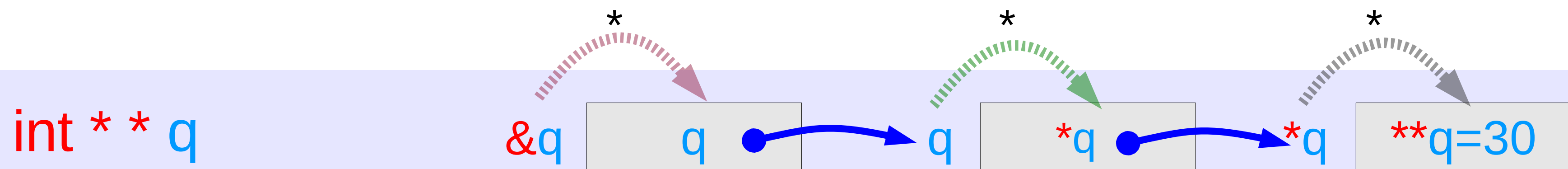
Dereferencing Operations

*(&a) = a



*(&p) = p

*(p) = *p



*(&q) = q

*(q) = *q

*(*q) = **q

Direct access to an integer **a**

```
int a ;
```

&a | a =100

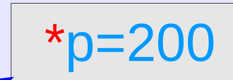
address	value	
&a	a	integer

Direct Access

1 memory access

Indirect access *p to an integer a

int * p ;

&p p 

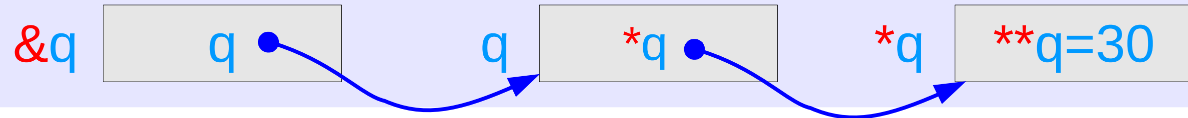
	address	value
Indirect Access	&p	p

2 memory accesses

Dereference Operator *
the content of the pointed location

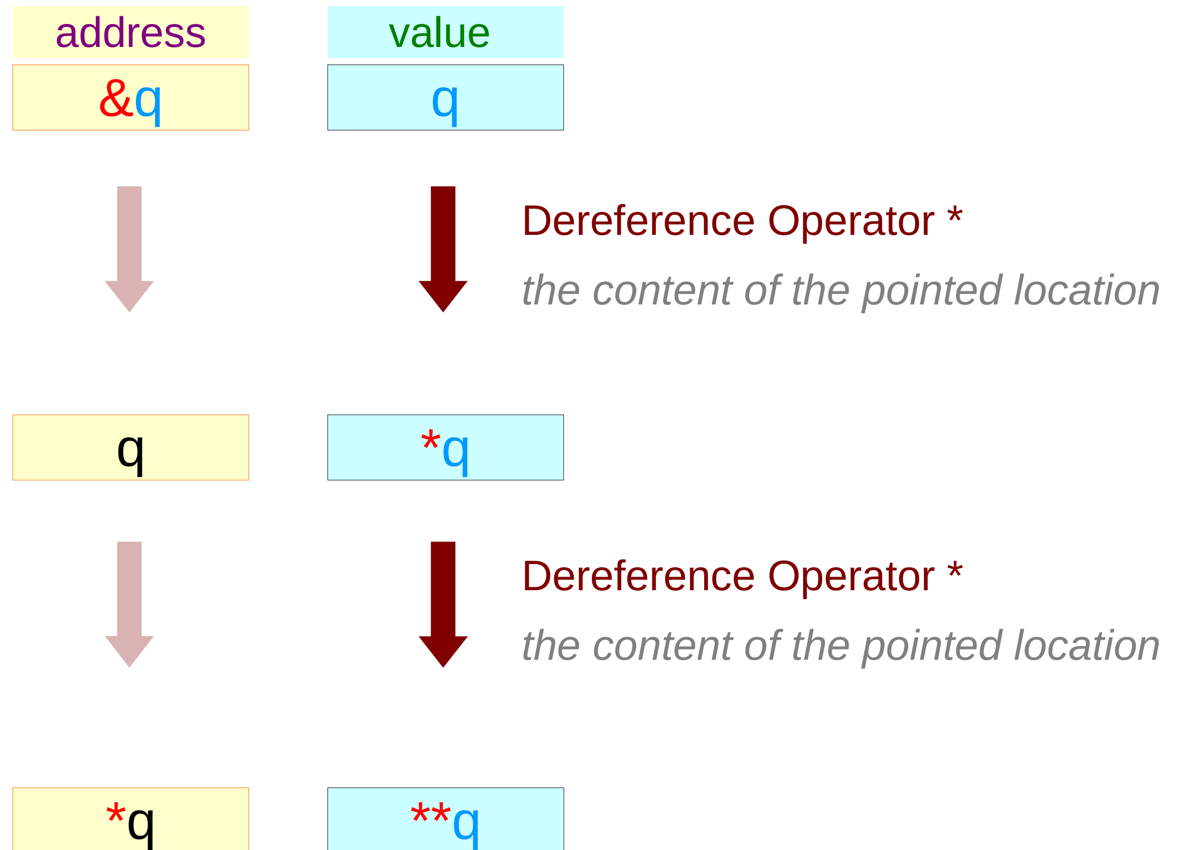
p	*p

Double indirect access **q to an integer a



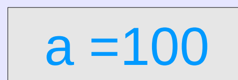
Double Indirect Access

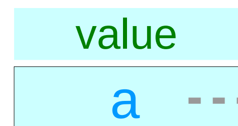
3 memory accesses



Values of variables

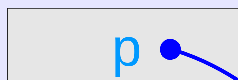
int a ;

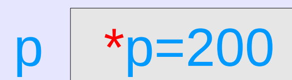
&a 

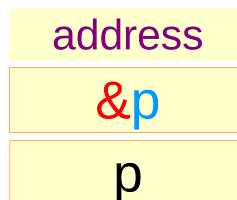


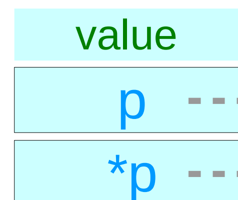
integer

int * p ;

&p 

p 

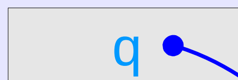


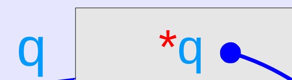


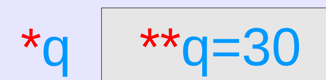
address

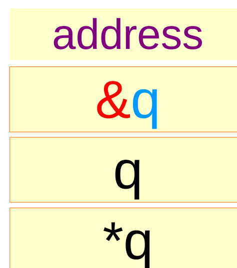
integer

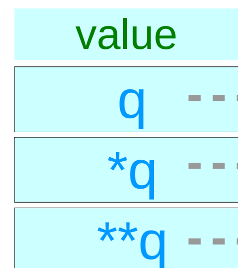
int * * q ;

&q 

q 

*q 





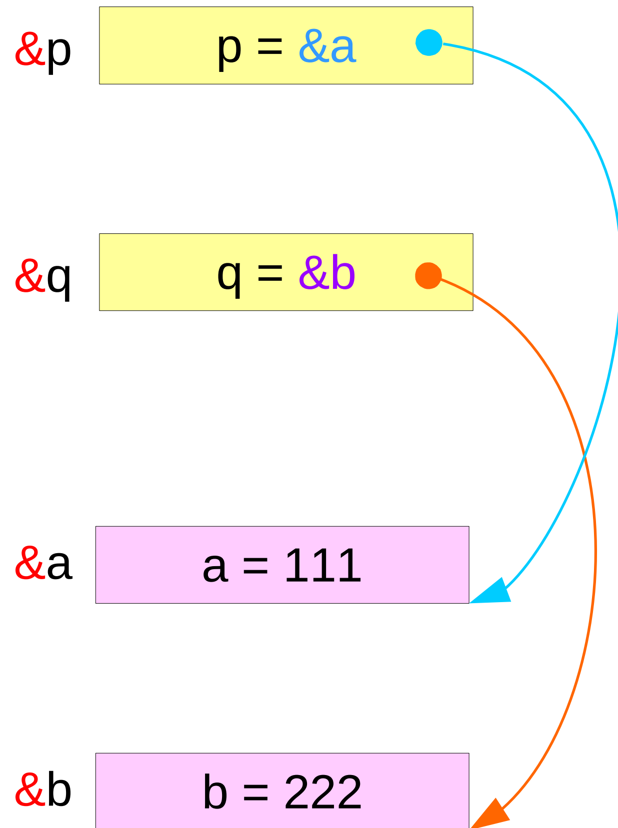
address

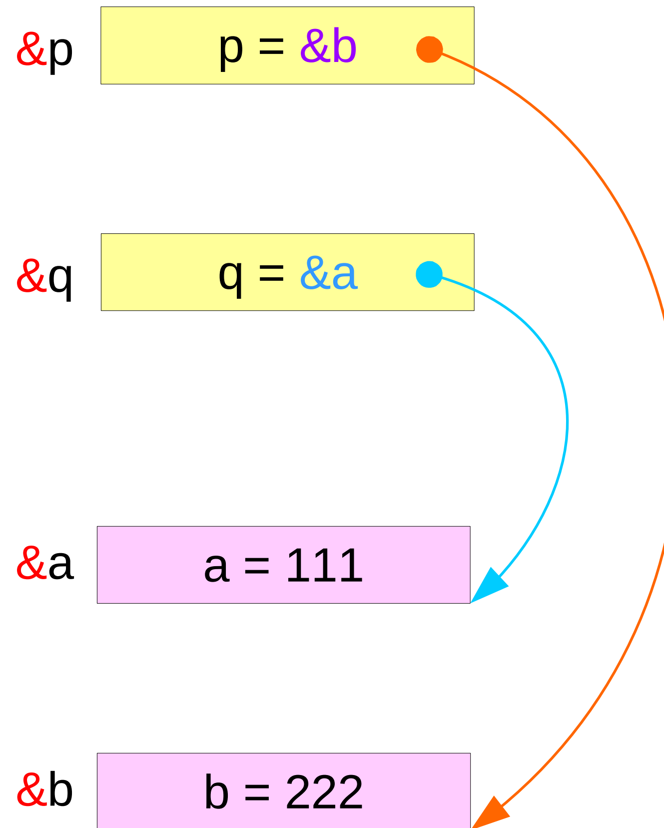
address

integer

Swapping pointers

Swapping integer pointers



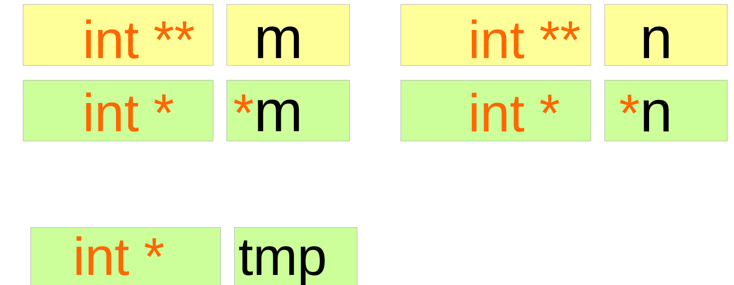


Swapping integer pointers

```
int            *p,    *q ;

swap_pointers(  &p,    &q );             function call

void swap_pointers( int **, int ** );    function prototype
```

Pass by integer pointer reference

```
void swap_pointers (int **m, int **n)
{
    int* tmp;

    tmp = *m;
    *m = *n;
    *n = tmp;
}
```

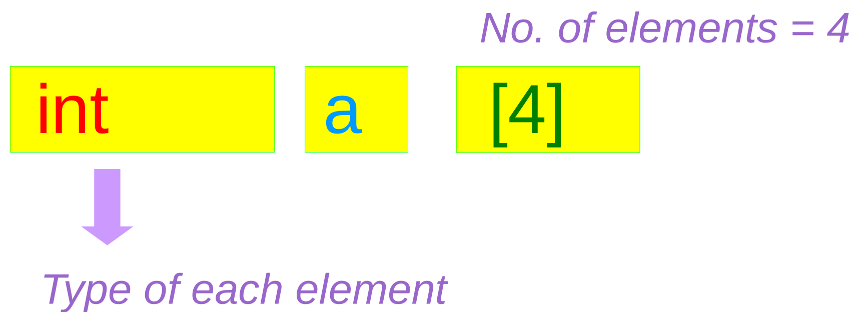


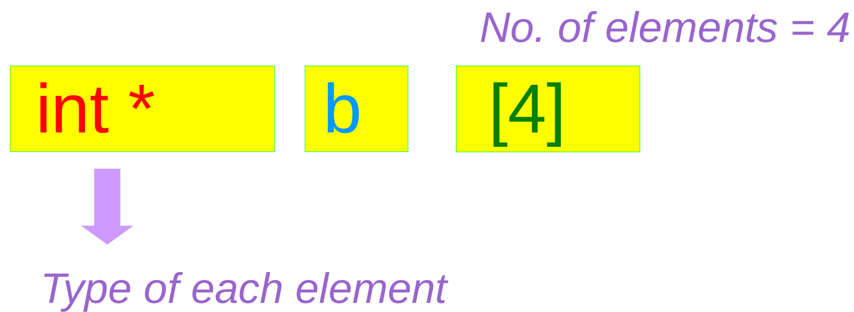

```
int   a,  b;
int  *p, *q;        p=&a, q=&b;
      …
swap_pointers( &p, &q );
```

Array of Pointers

Array of Pointers

```
int     a [4];

int *   b [4];
```

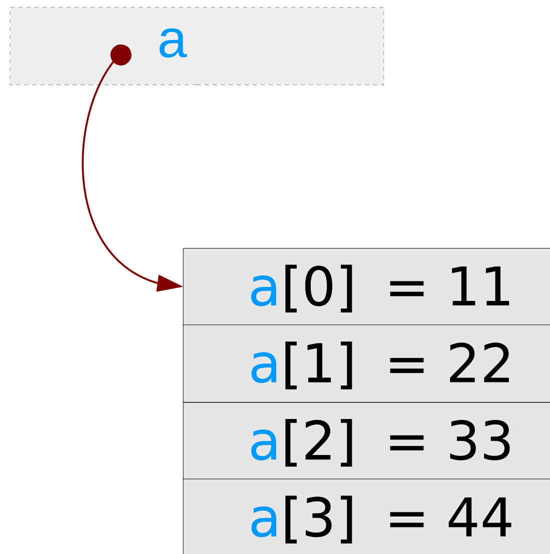




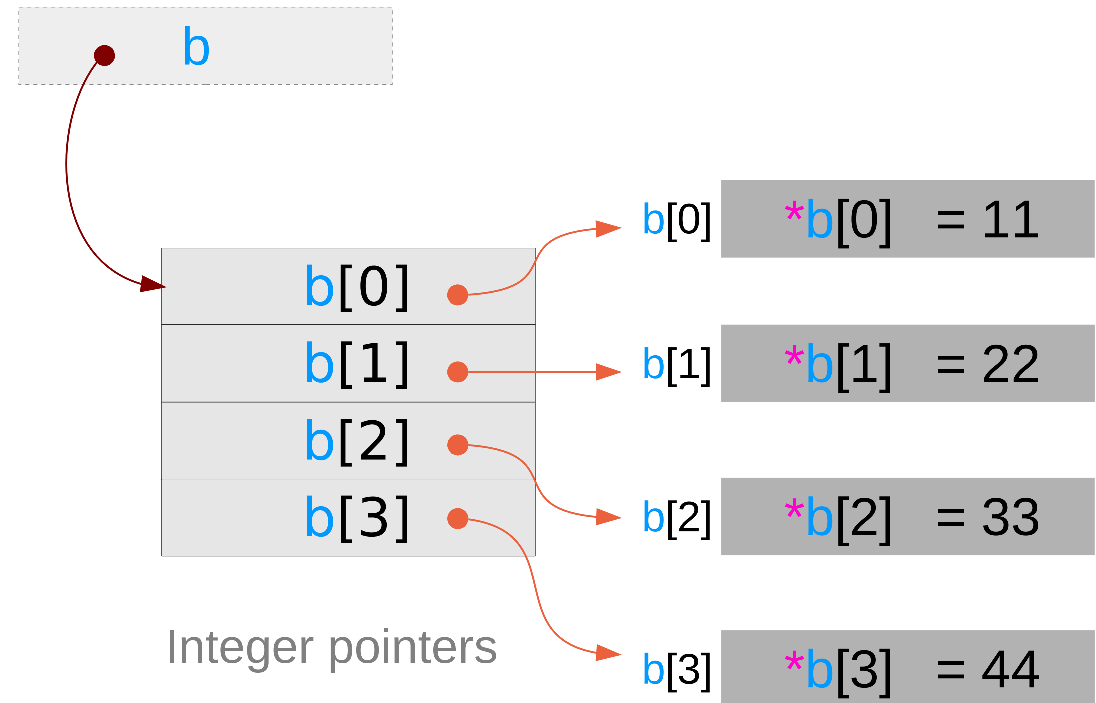
Array of Pointers – a variable view

int a [4];

int * b [4];



Integers



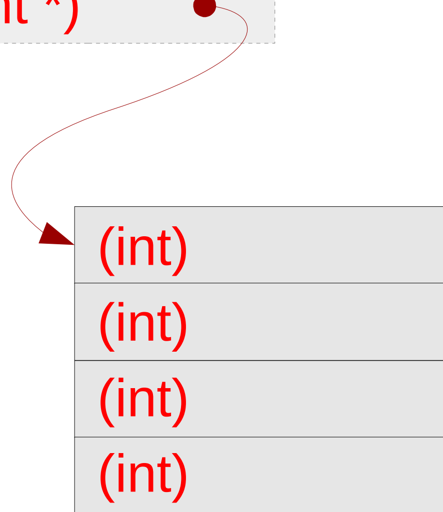
Integer pointers

taking actual
memory locations

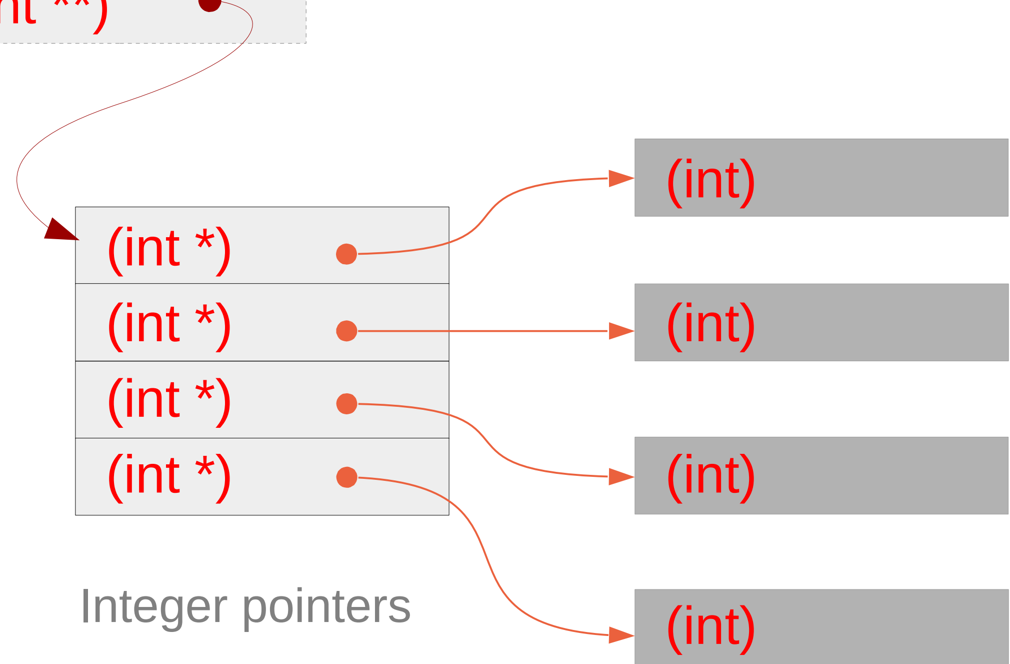
Array of Pointers – a type view

int a [4];

int * b [4];



Integers



taking actual
memory locations

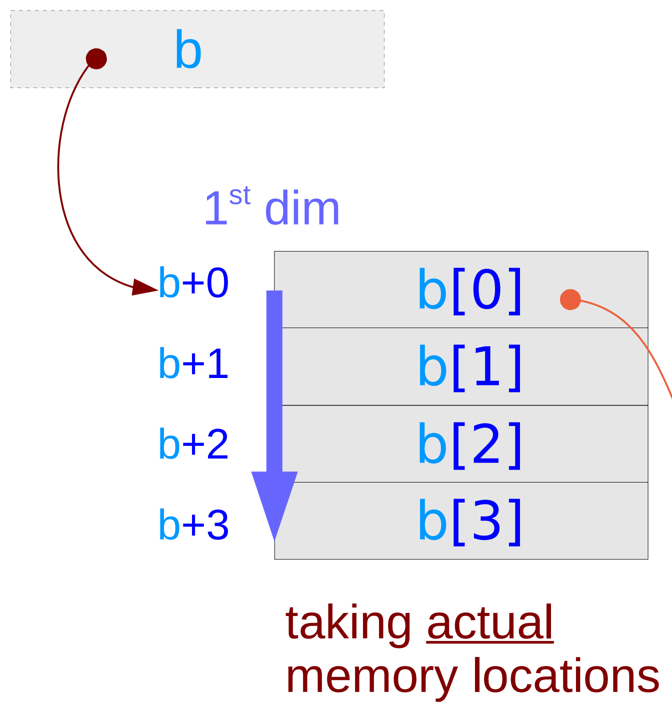
Array of Pointers – extending a dimension

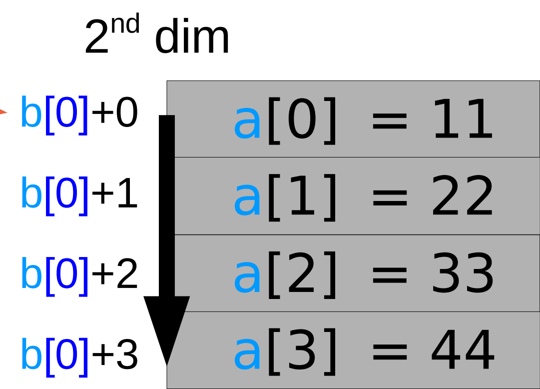
```
int *      b [4];
```

assignment b[0] = a

equivalence

a[0] ≡ b[0][0] ≡ *(*(b+0)+0)
a[1] ≡ b[0][1] ≡ *(*(b+0)+1)
a[2] ≡ b[0][2] ≡ *(*(b+0)+2)
a[3] ≡ b[0][3] ≡ *(*(b+0)+3)





Pointer to Arrays

Pointer to an array – variable declarations

```
int  m ;
int *n ;
```

an integer pointer

```
int  a    [4]
int (*p)  [4]
```

an array pointer

```
int func   (int a, int b) ;
int (* fp) (int a, int b) ;
```

a function pointer

Pointer to an array – a type view

int 4 byte data

int *

an integer pointer

array pointer:
a pointer to an array

pointer array:
an array of pointers

int [4] 4*4 byte data

int (*) [4]

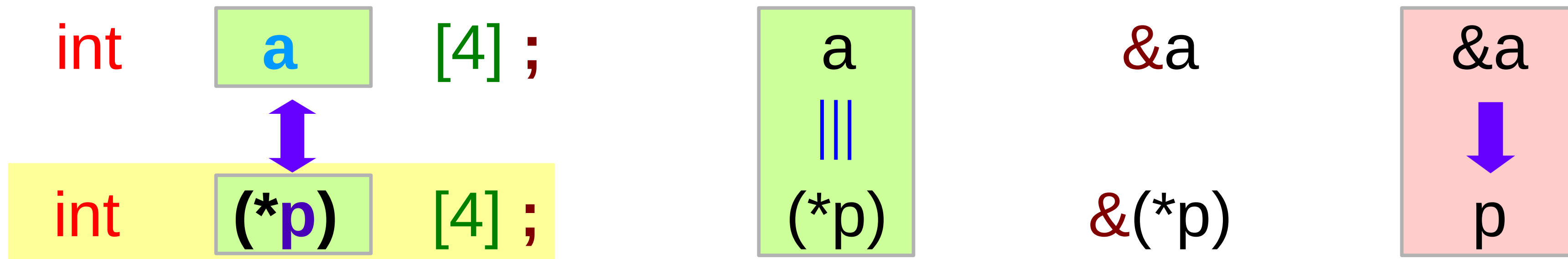
an array pointer

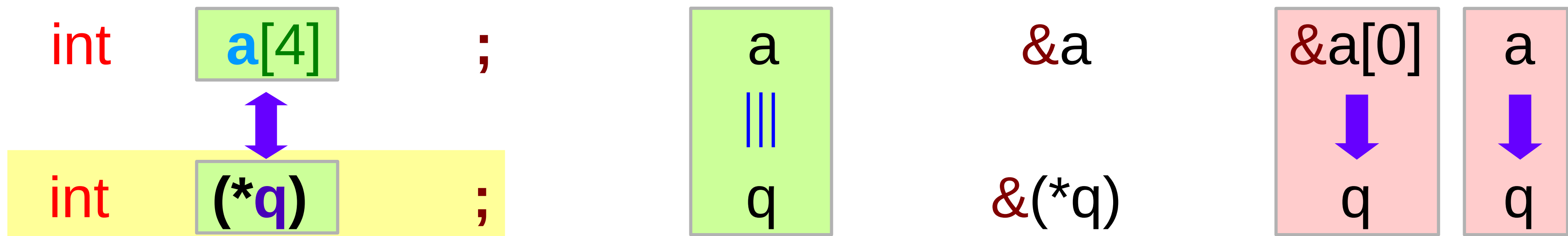
int (int, int) instructions

int (*) (int, int)

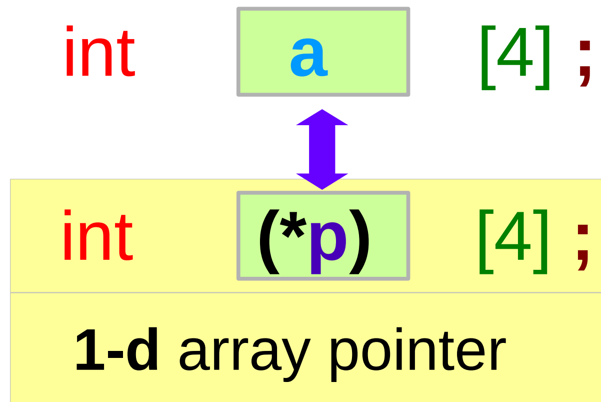
a function pointer

Pointer to an array : assignment and equivalence





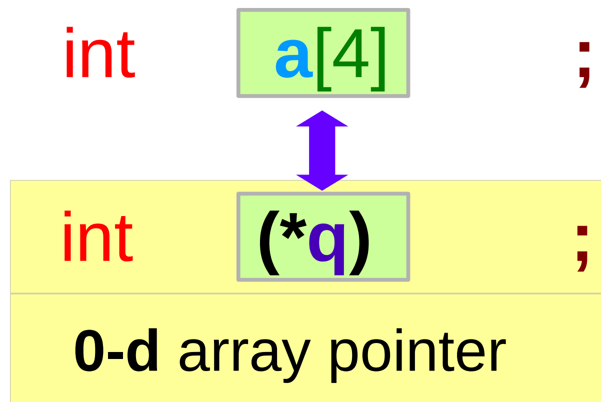
Pointer to an array : size of array



p = &a;

sizeof(p)= 8 bytes : the size of a pointer

sizeof(*p)= 4*4 bytes : the whole size of the pointed 1-d array



q = a;

sizeof(q)= 8 bytes : the size of a pointer

sizeof(*q)= 4 bytes : the whole size of the pointed 0-d array

Pointer to an array – a variable view (1)

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

assignment

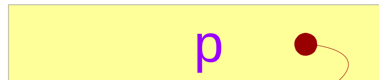
p = &a

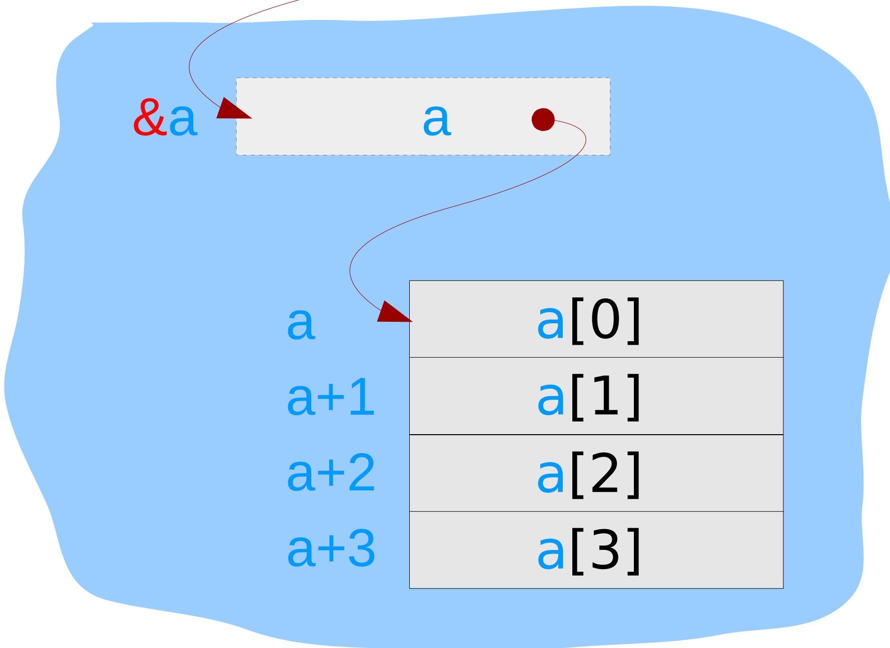
equivalence

***p ≡ a**

1-d array pointer

points to a **1-d** array –
a aggregated type data






```
int    a [4];
```

p : int (*) [4] type

Pointer to an array – a variable view (2)

int (*q) ;

assignment

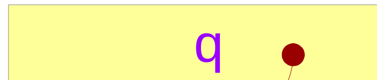
q = &a[0]

q = a

equivalence

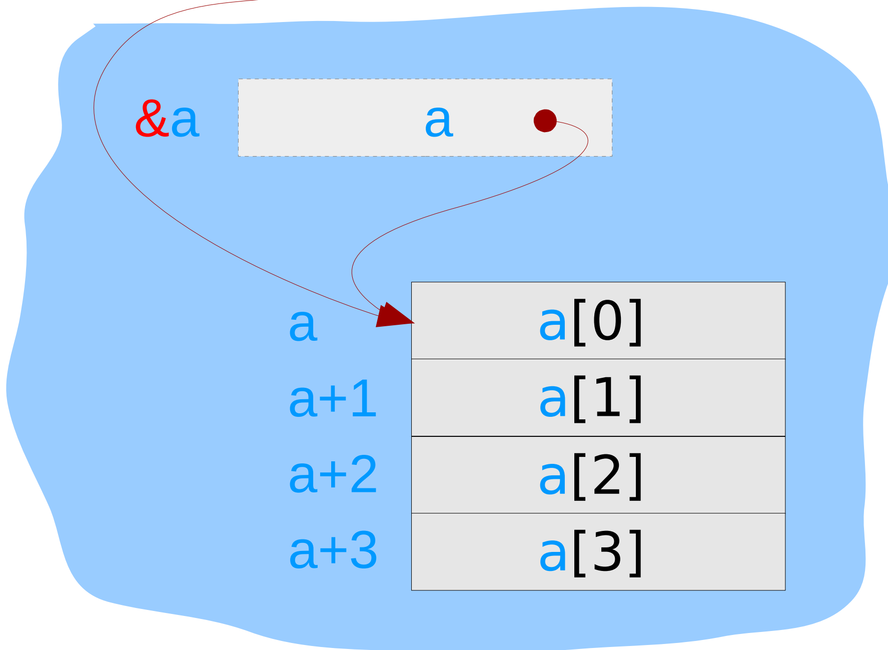
*q ≡ *a

q ≡ a



0-d array pointer

points to an array element – an integer type data



int a[4] ;

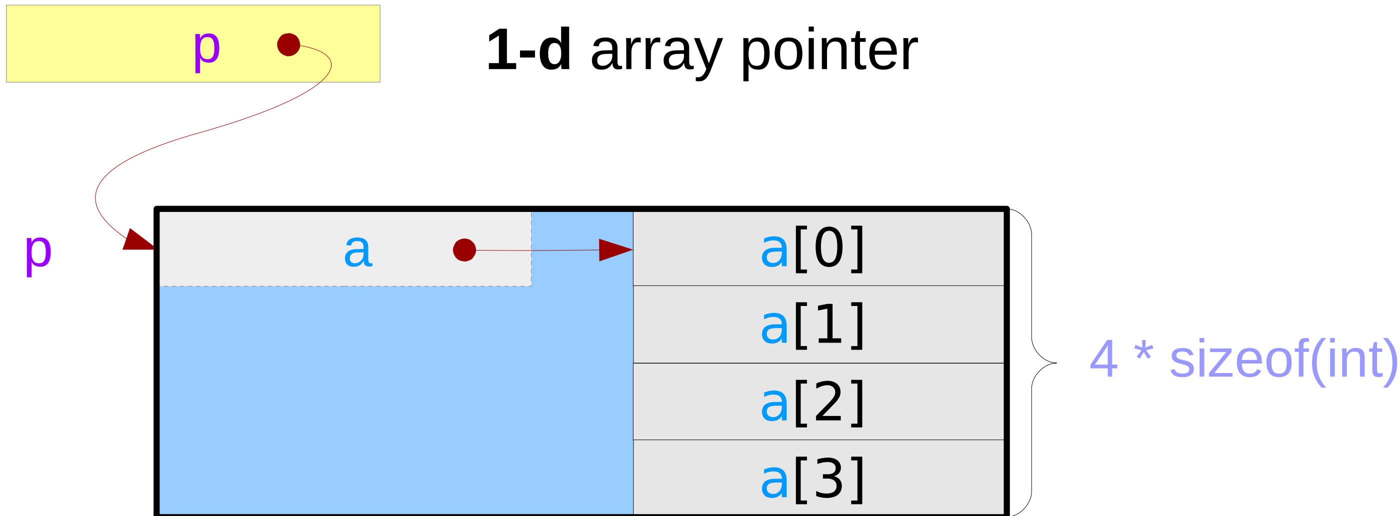
q : int (*) = int * type

Pointer to an array – an aggregated type view

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

An aggregated type
- starting address (&a)
- size of all the array elements (16 bytes)

1-d array pointer

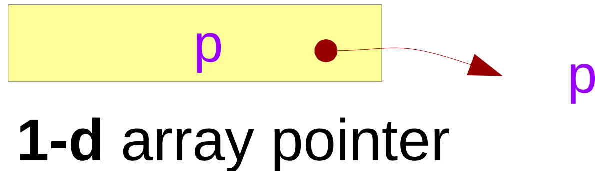


Incrementing an array pointer

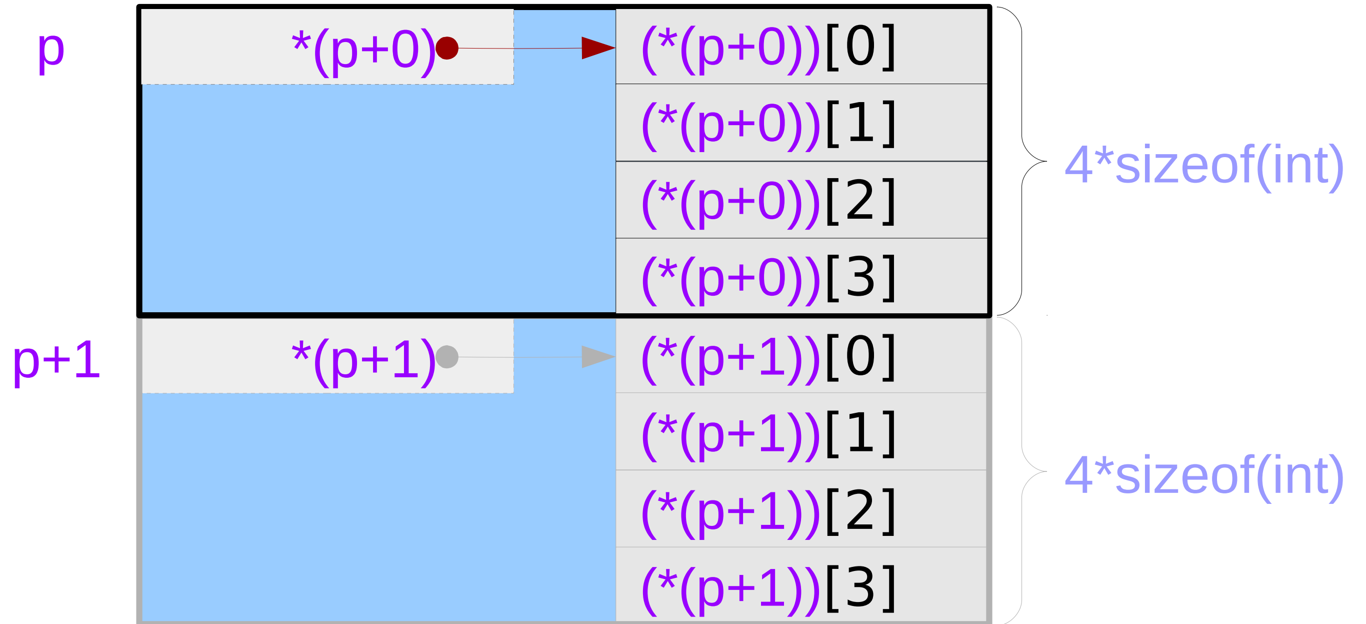
```
int     (*p) [4];
```

Aggregated Type Size

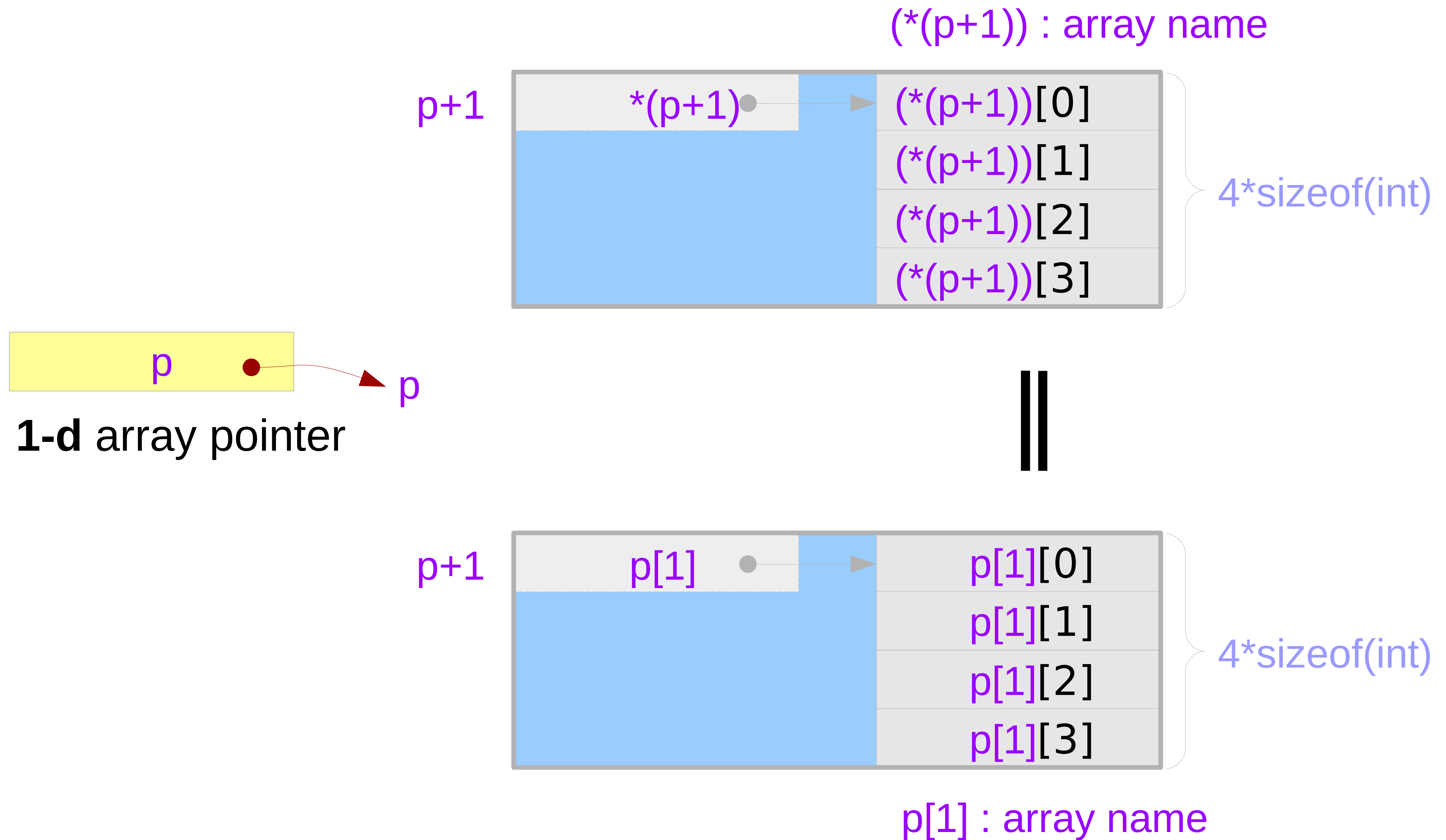
address p+1 – **address** p
= (long) (p+1) - (long) (p)   = 4 * sizeof(int)



1-d array pointer



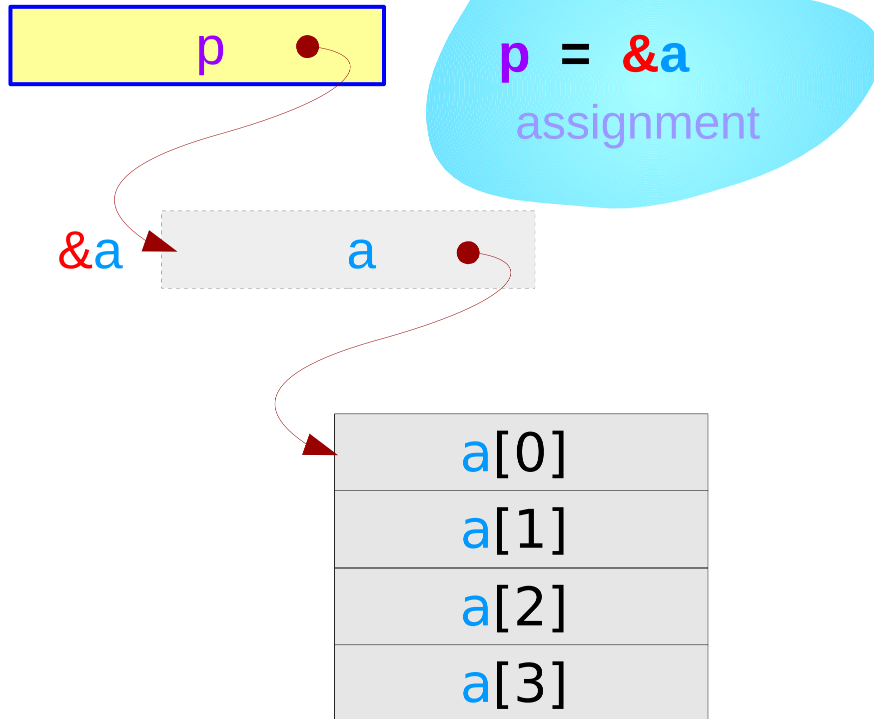
Incrementing an array pointer – extending a dimension

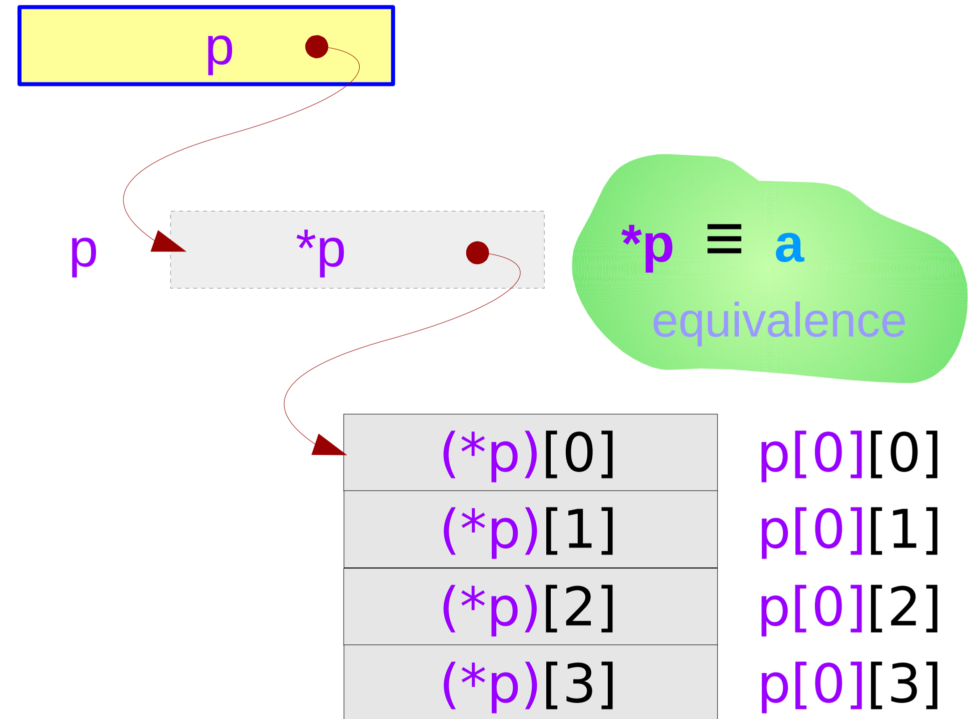


A **1-d** array pointer and a **1-d** array

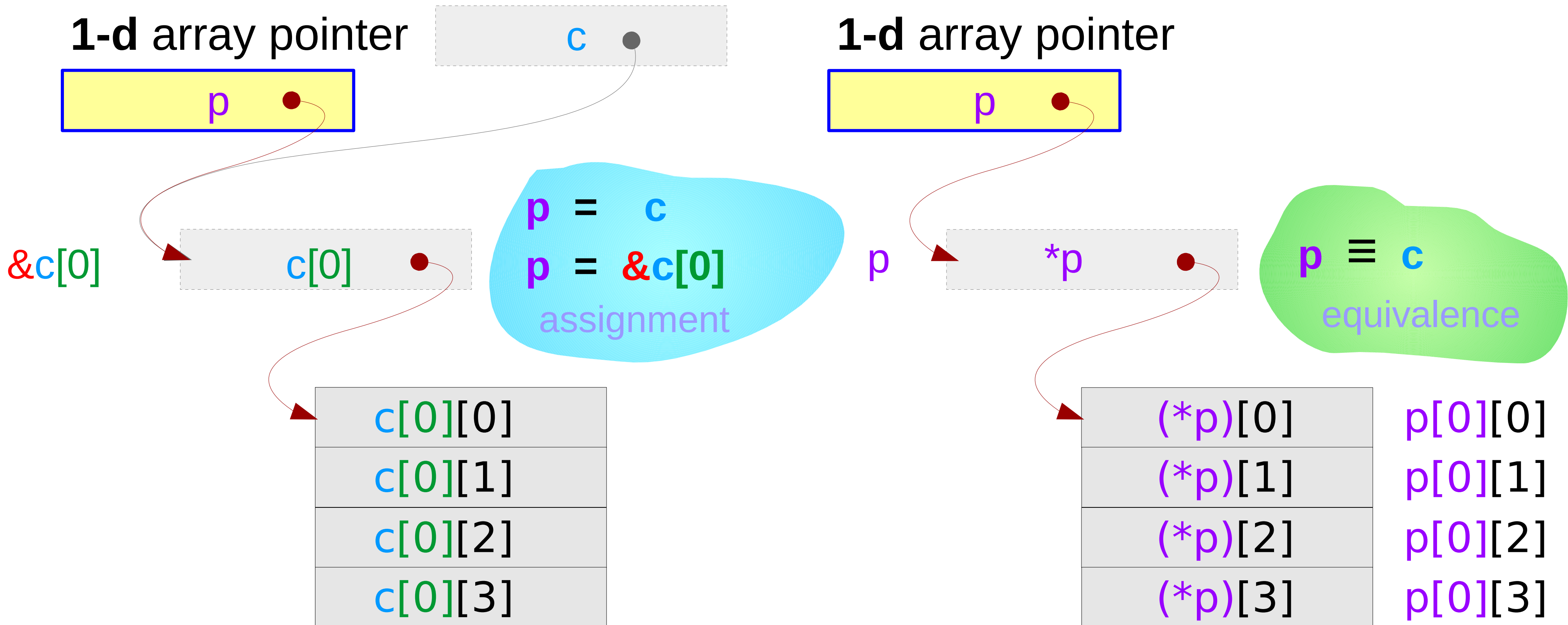
```
int        a [4];
```

```
int  (*p) [4]  = &a;
```





A **1-d** array pointer and a **2-d** array



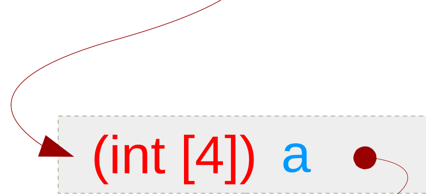
A **1-d** array pointer and a **1-d** array – a type view

```
int          a [4];
```

```
int  (*p) [4]  = &a;
```

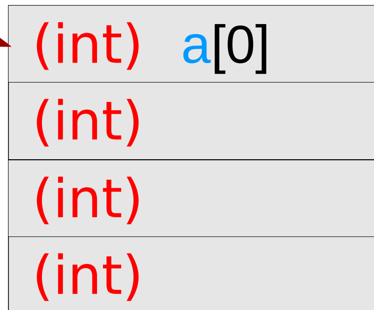
1-d array pointer

(int (*)[4])p



(int [4]) a

(int *)



(int) a[0]

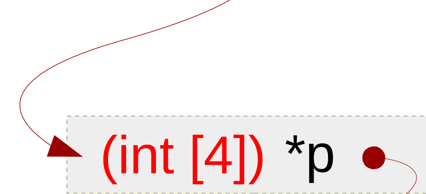
(int)

(int)

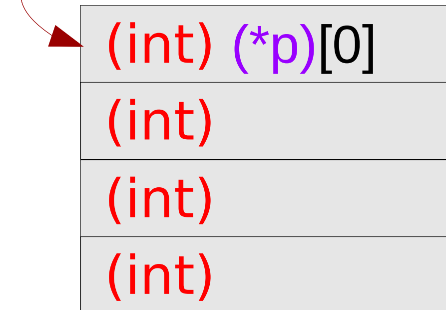
(int)

1-d array pointer

(int (*)[4])p



(int [4]) *p



(int) (*p)[0]

p[0][0]

(int)

(int)

(int)

A **1-d** array pointer and a **2-d** array – a type view

```
int          c [4][4];
```

```
int  (*p) [4]  = &c[0];
```

1-d array pointer

(int (*)[4]) p

(int (*)[4]) c

(int [4]) c[0]

(int *)

(int) c[0][0]
(int)
(int)
(int)

1-d array pointer

(int (*)[4]) p

(int [4]) *p

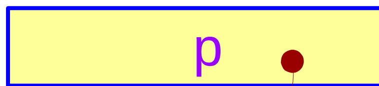
(int) p[0][0]
(int)
(int)
(int)

p[0][0]

A **1-d** array pointer – extending a dimension

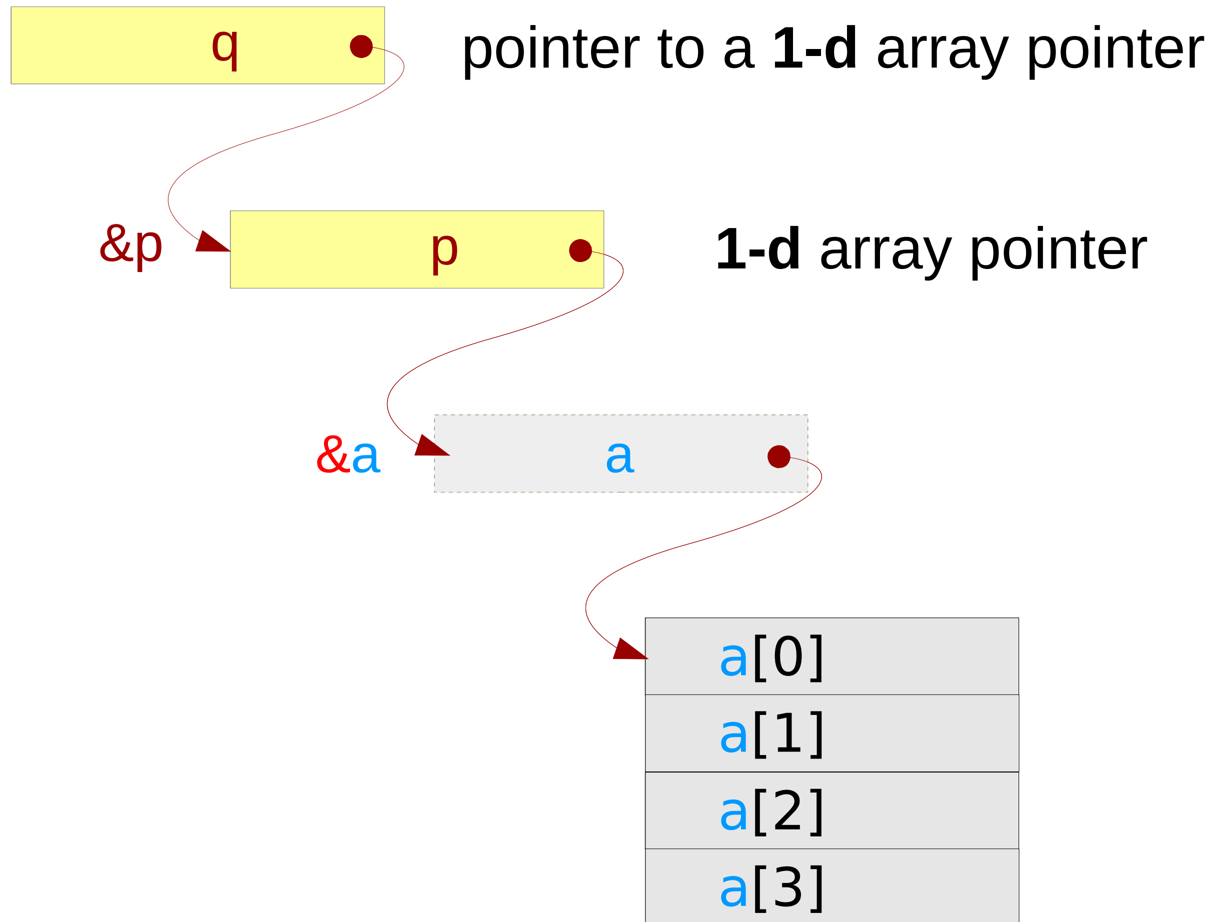
```
int  (*p) [4]  ;
```

1-d array pointer



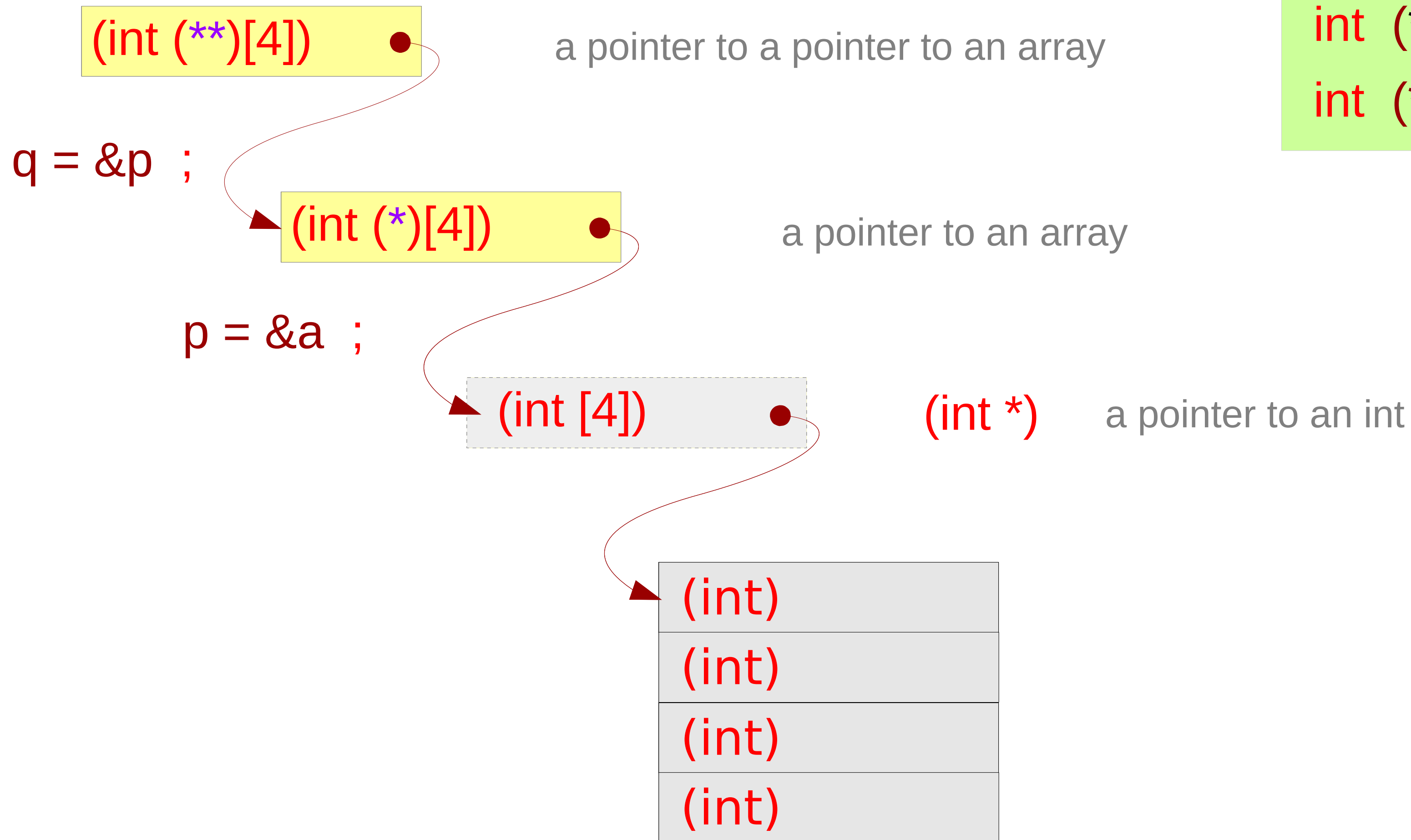
can be viewed as a 2-d array name
: an additional dimension is added

Double pointer to a **1-d** array – a variable view




```
int  a[4]  ;
int  (*p) [4] = &a ;
int  (**q) [4] = &p ;
```

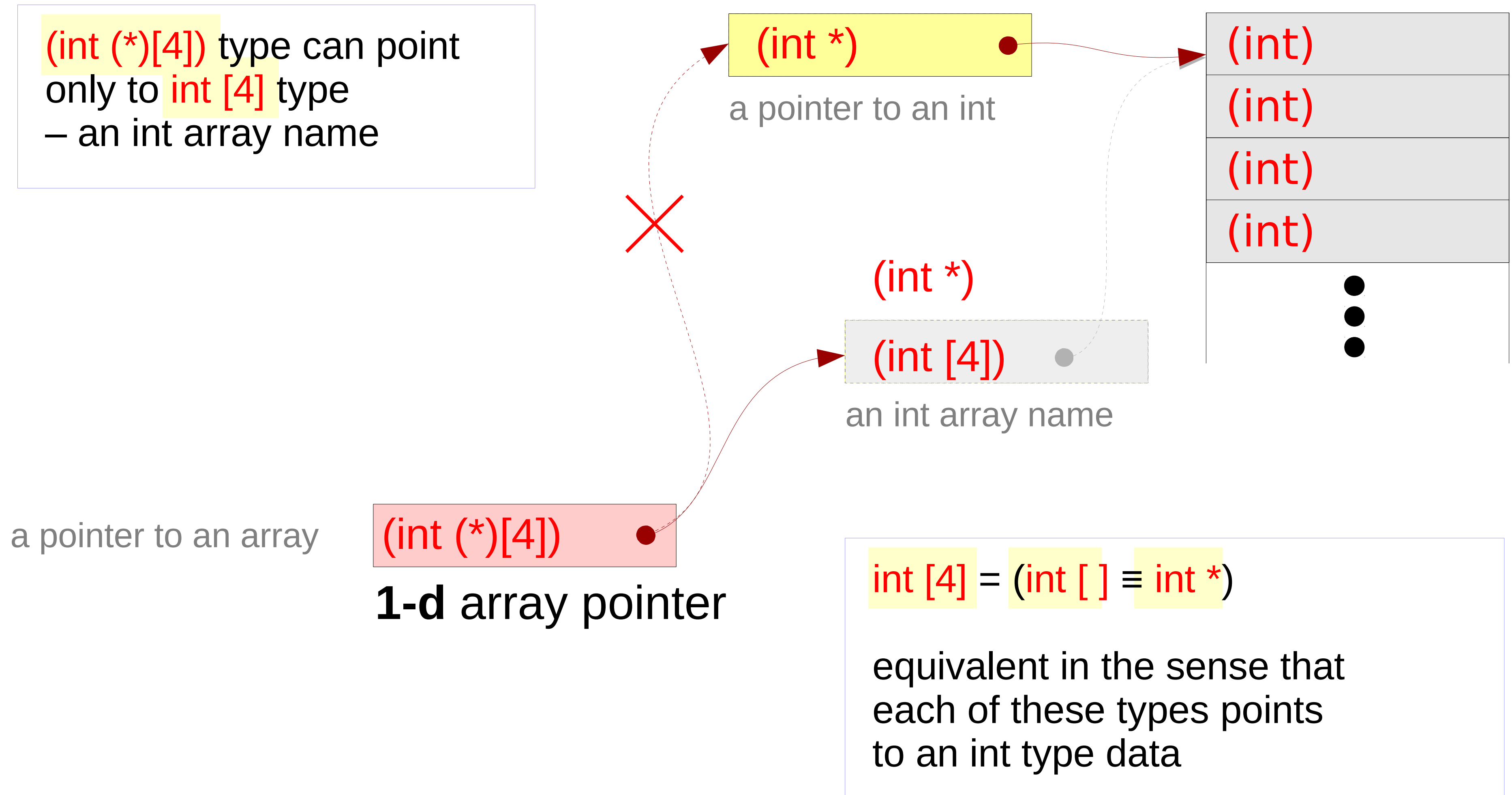
Double pointer to a **1-d** array – a type view



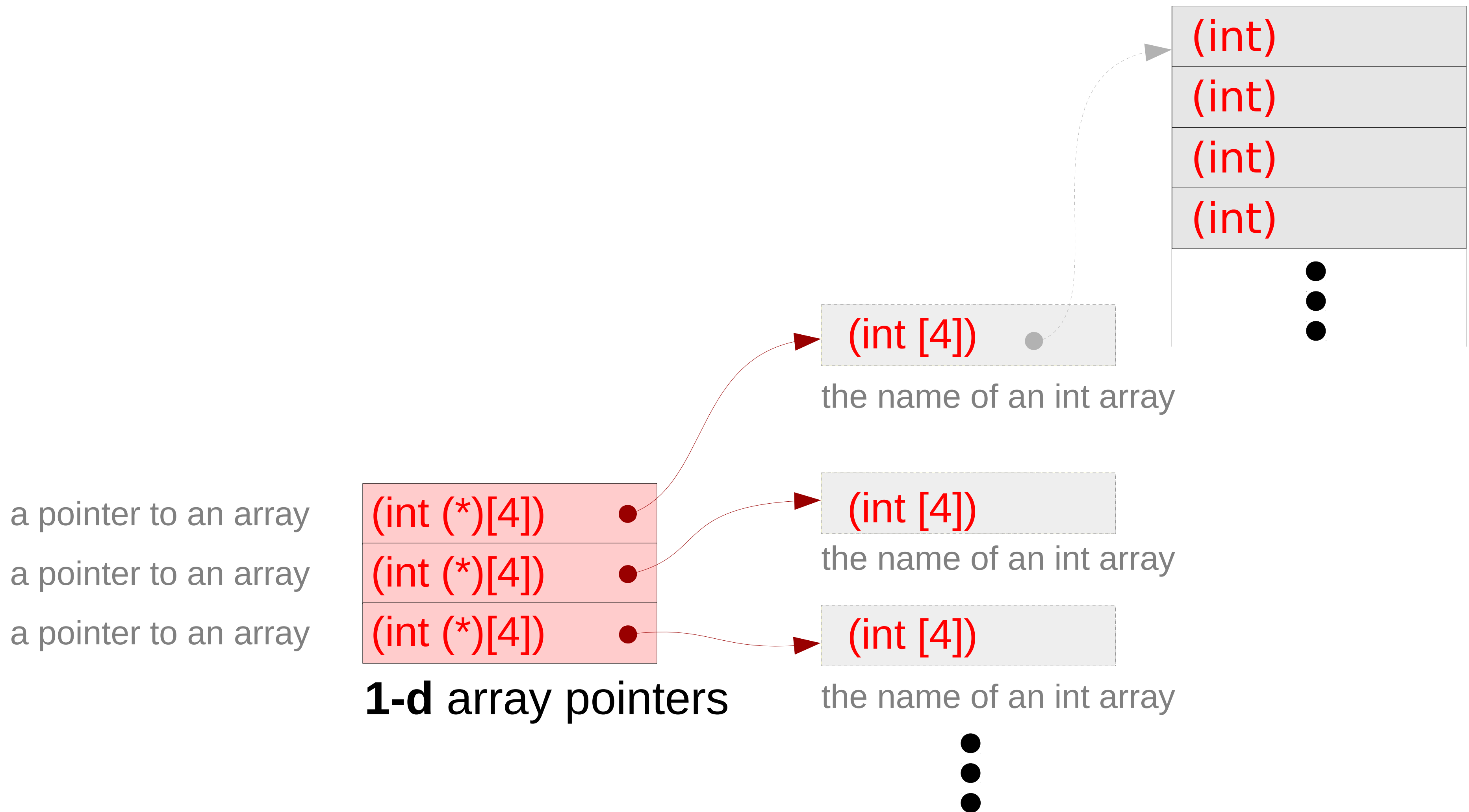

```
int a[4] ;
int (*p) [4] = &a ;
int (**q) [4] = &p ;
```

Pointer to Multi-dimensional Arrays

Integer pointer type



Series of array pointers – a type view

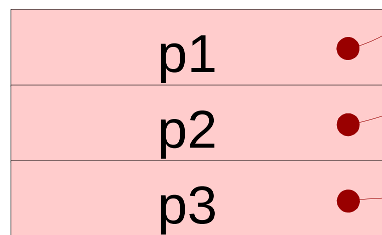


Series of array pointers – a variable view

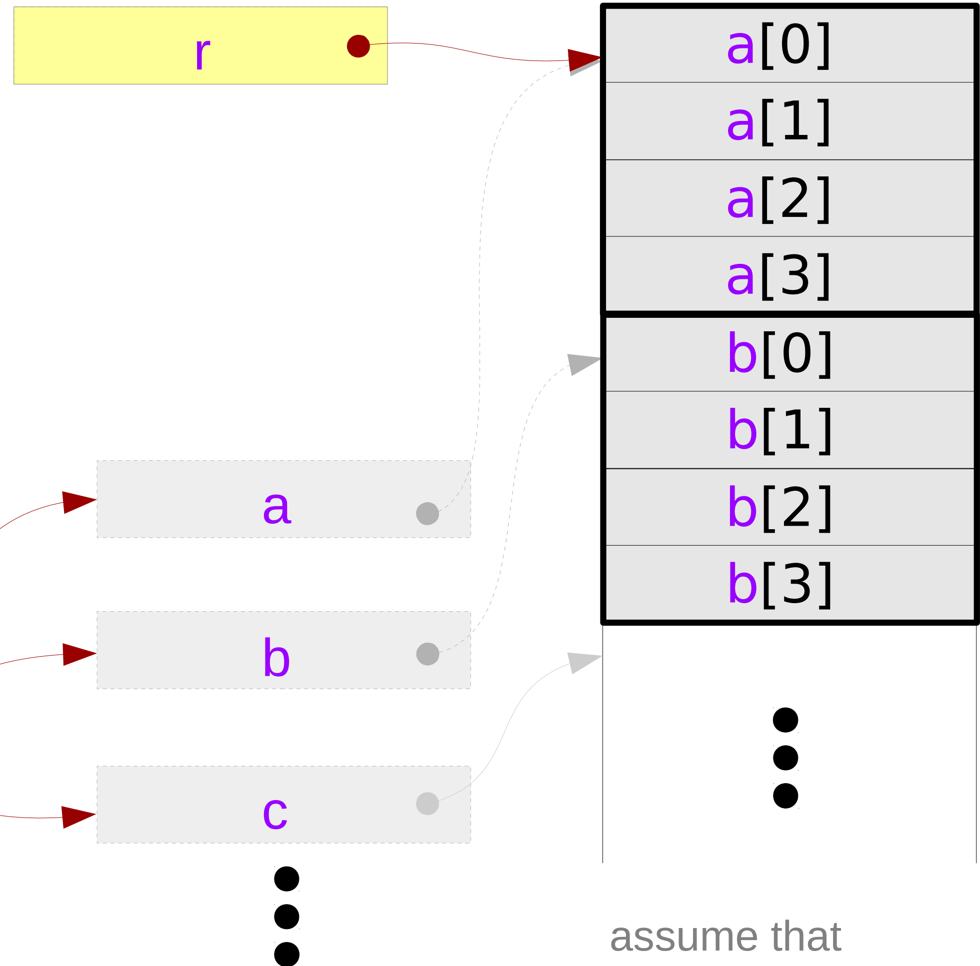
```
int a[4];  int (*p1)[4];   int  (*r);
int b[4];  int (*p2)[4];
int c[4];  int (*p3)[4];
```

assignment	equivalence
p1 = &a	(*p1) ≡ p1[0] ≡ a
p2 = &b	(*p2) ≡ p2[0] ≡ b
p3 = &c	(*p3) ≡ p3[0] ≡ c

a pointer to an array
a pointer to an array
a pointer to an array

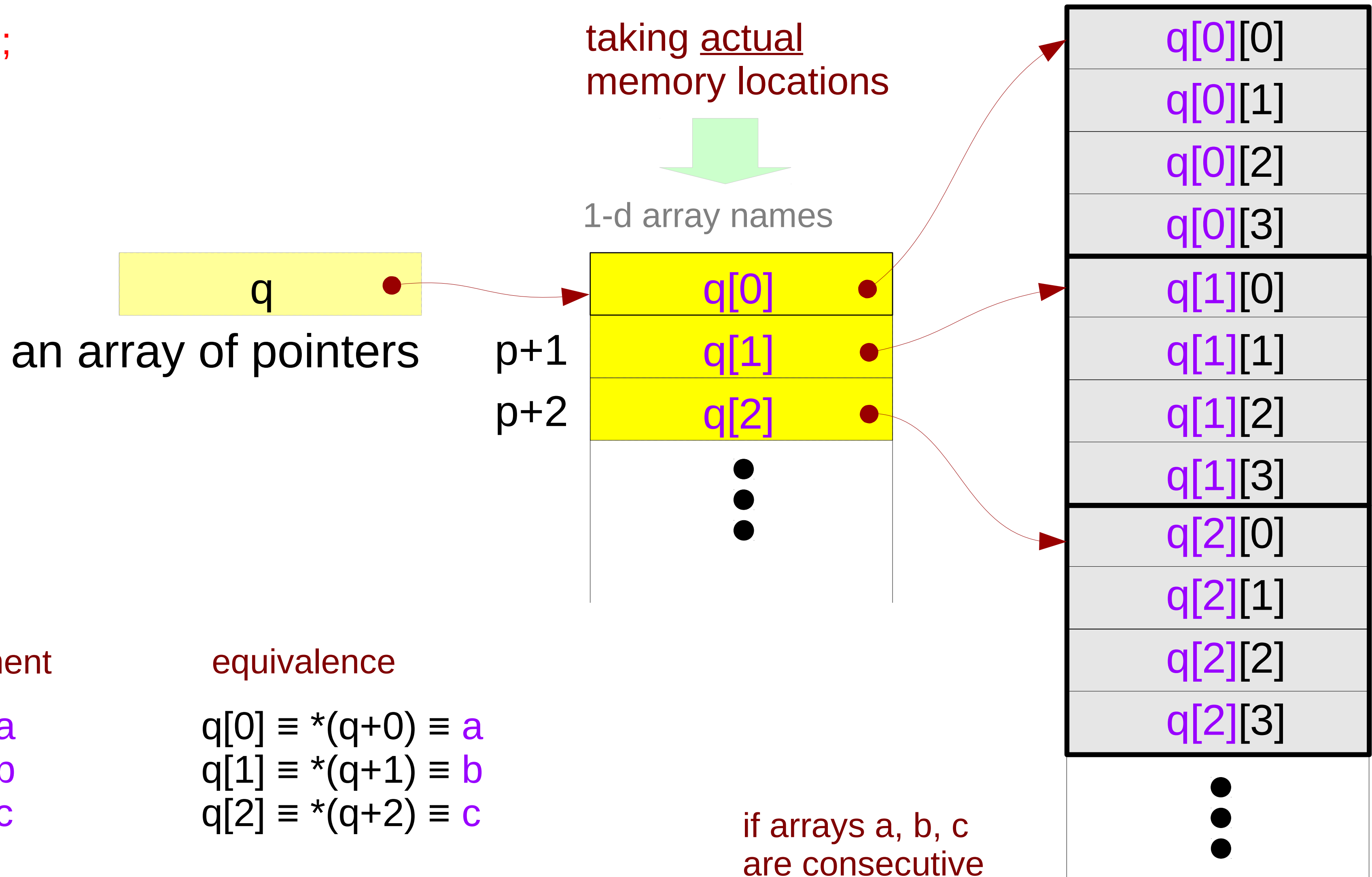


1-d array pointers



assume that array a, b, and c are contiguous in the memory

Pointer array – a variable view

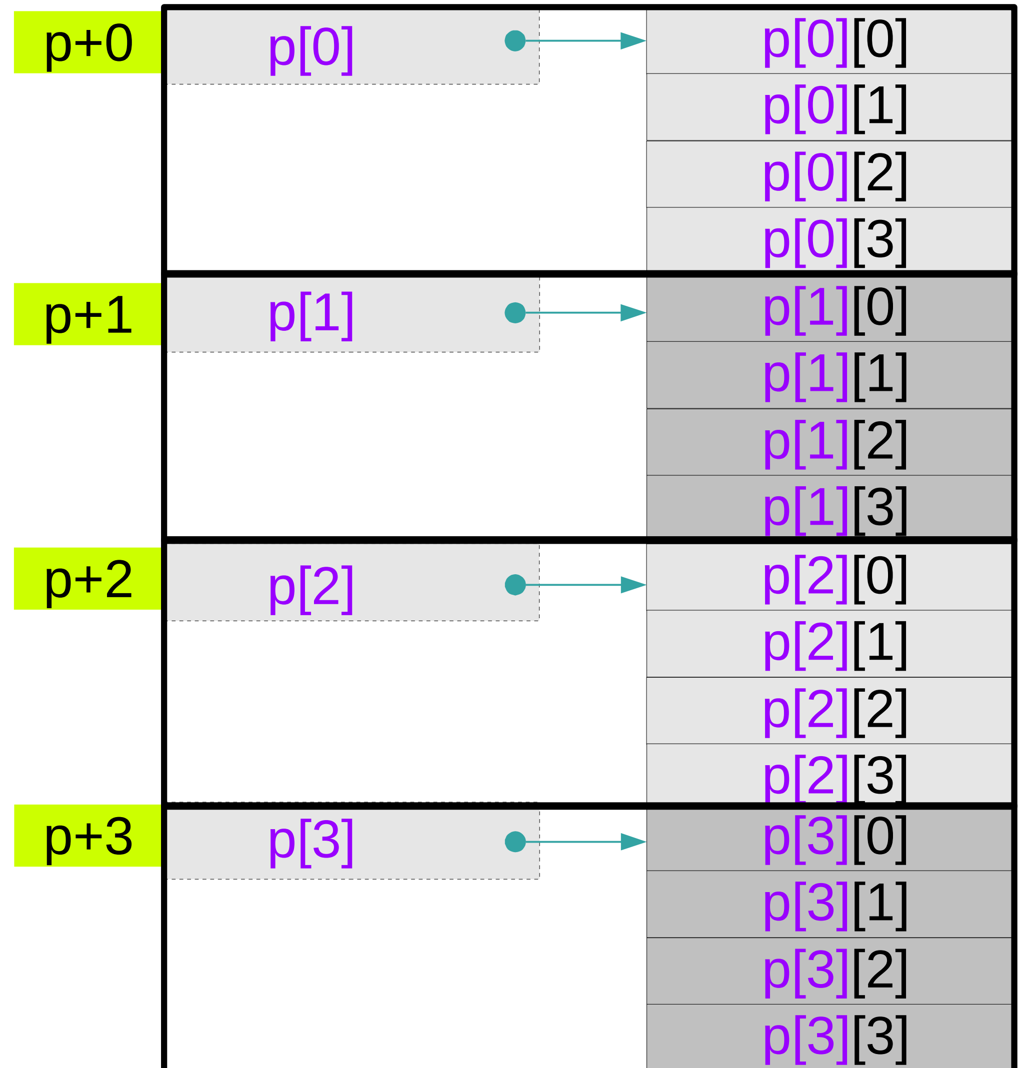


Array pointer to consecutive **1-d** arrays

int (*p)[4];

a pointer to an array

1-d array pointer



assignment

p = &a

equivalence

*(p+0) ≡ p[0] ≡ a
*(p+1) ≡ p[1] ≡ b
*(p+2) ≡ p[2] ≡ c
*(p+2) ≡ p[2] ≡ d

if arrays a, b, c, d
are consecutive

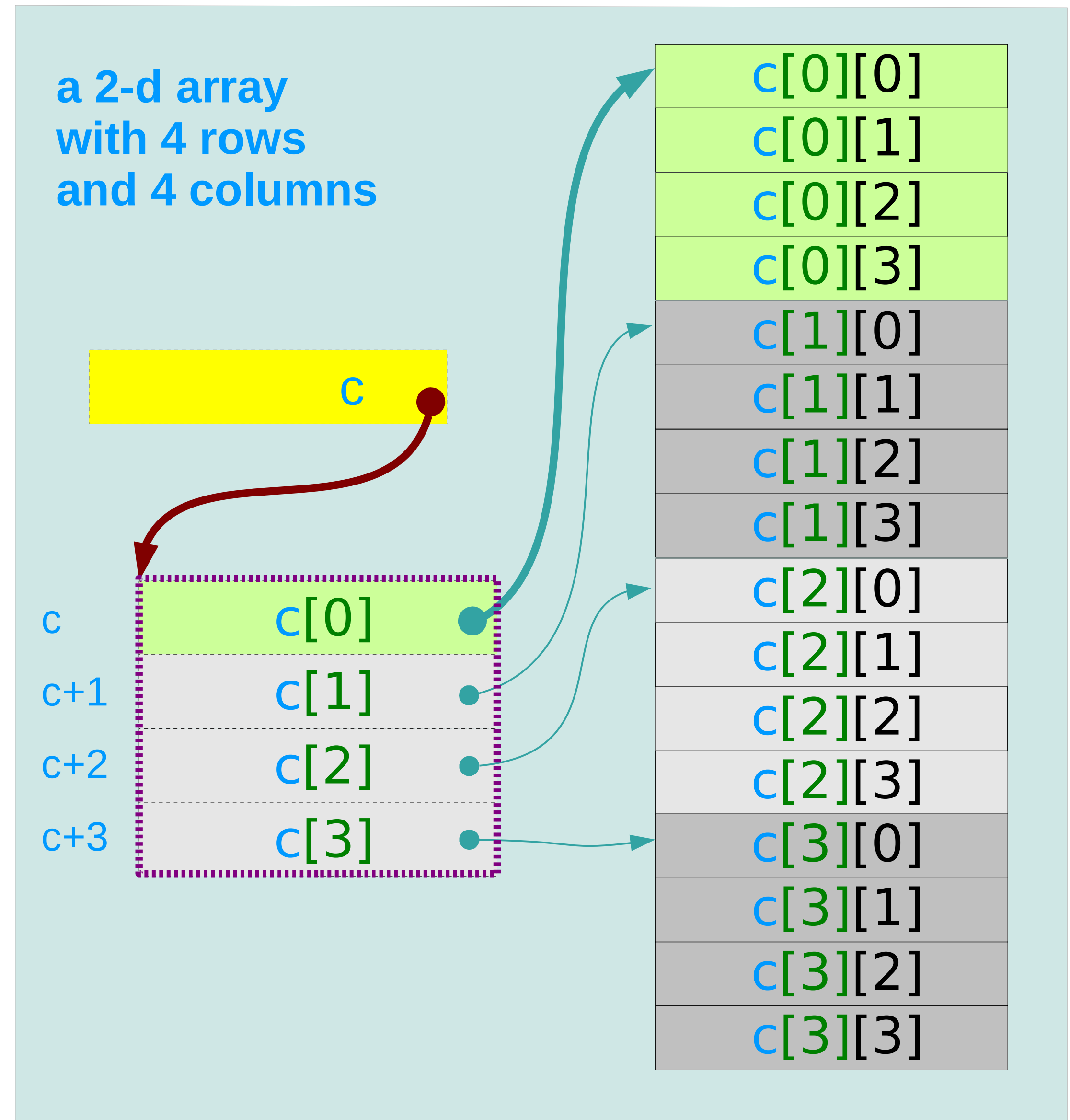
A **2-d** array and its sub-arrays – a variable view

the array name **c** of a **2-d** array as a **1-d** array pointer which points to its 1st **1-d** sub-array

c is the **1-d** array pointer
c[i]'s are the **1-d** sub-array name

c[0]	the 1st	1-d sub-array name
c[1]	the 2nd	1-d sub-array name
c[2]	the 3rd	1-d sub-array name
c[3]	the 4th	1-d sub-array name

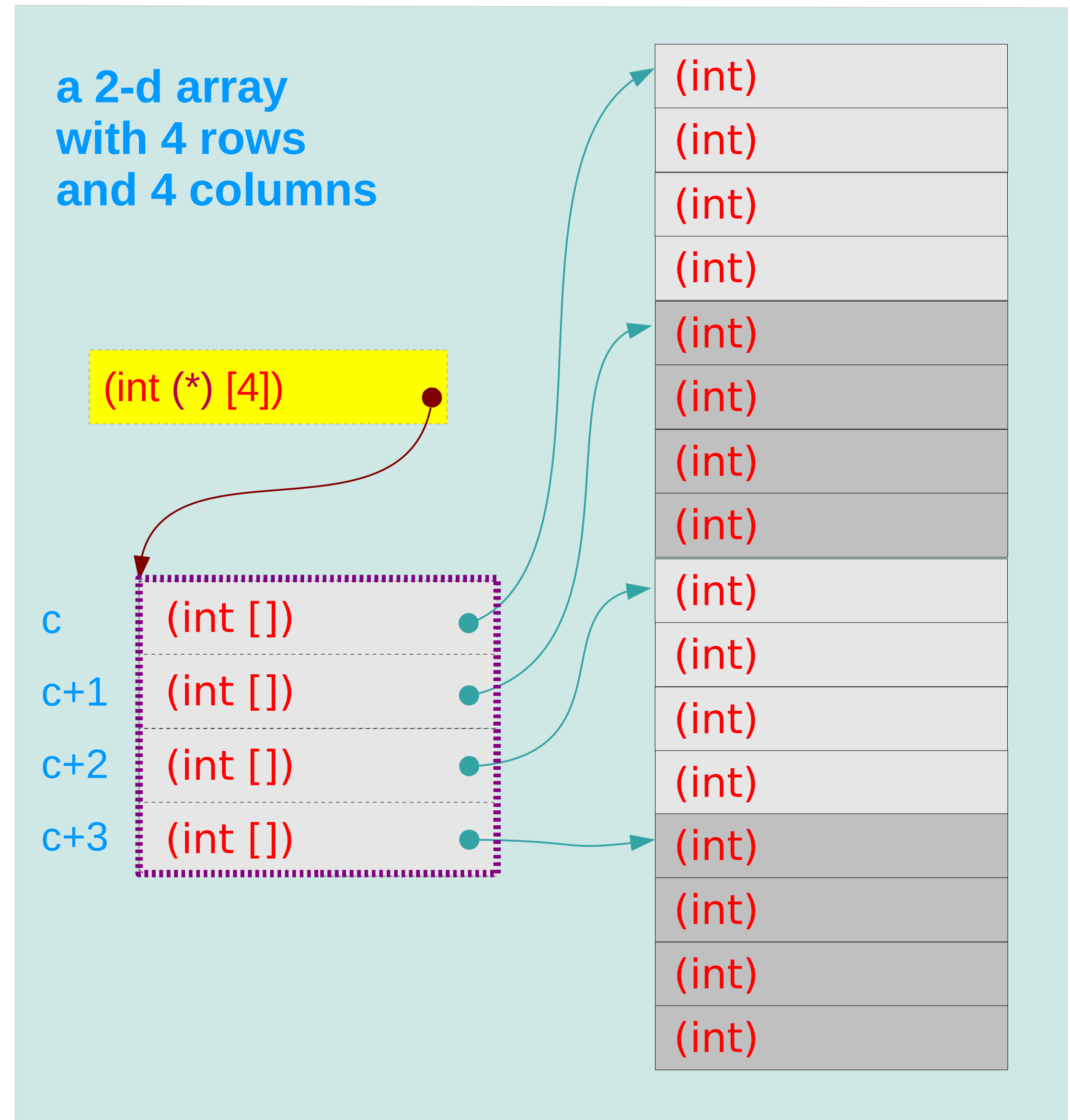
Compilers can make **c[i]**'s require no actual memory locations



A **2-d** array and its sub-arrays – a type view

1-d array pointer

1-d array name
1-d array name
1-d array name
1-d array name



1-d subarray aggregated data type

The 1st subarray c[0]

sizeof(c[0]) = 16 bytes

The 2nd subarray c[1]

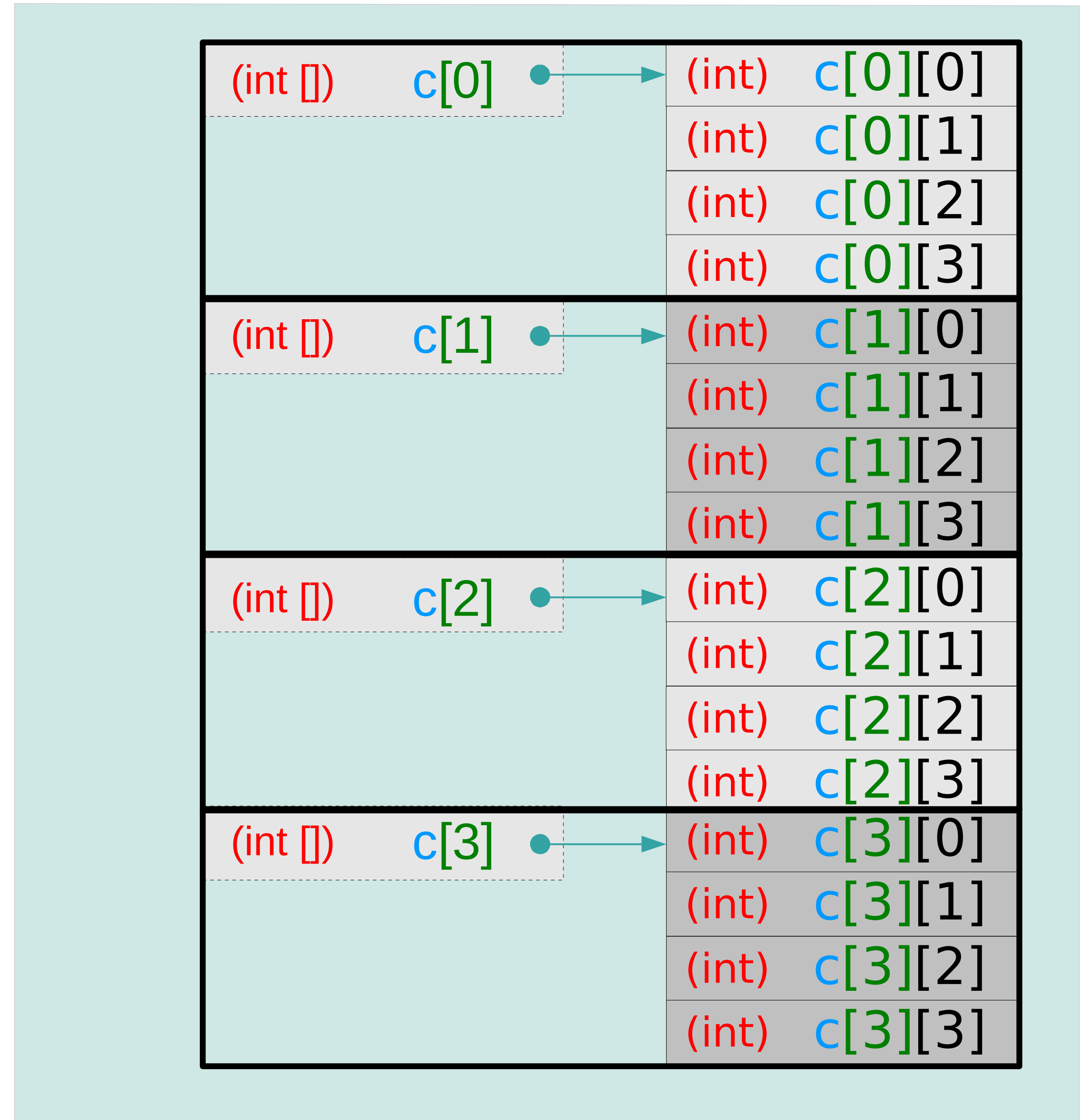
sizeof(c[0]) = 16 bytes

The 3rd subarray c[2]

sizeof(c[0]) = 16 bytes

The 4th subarray c[3]

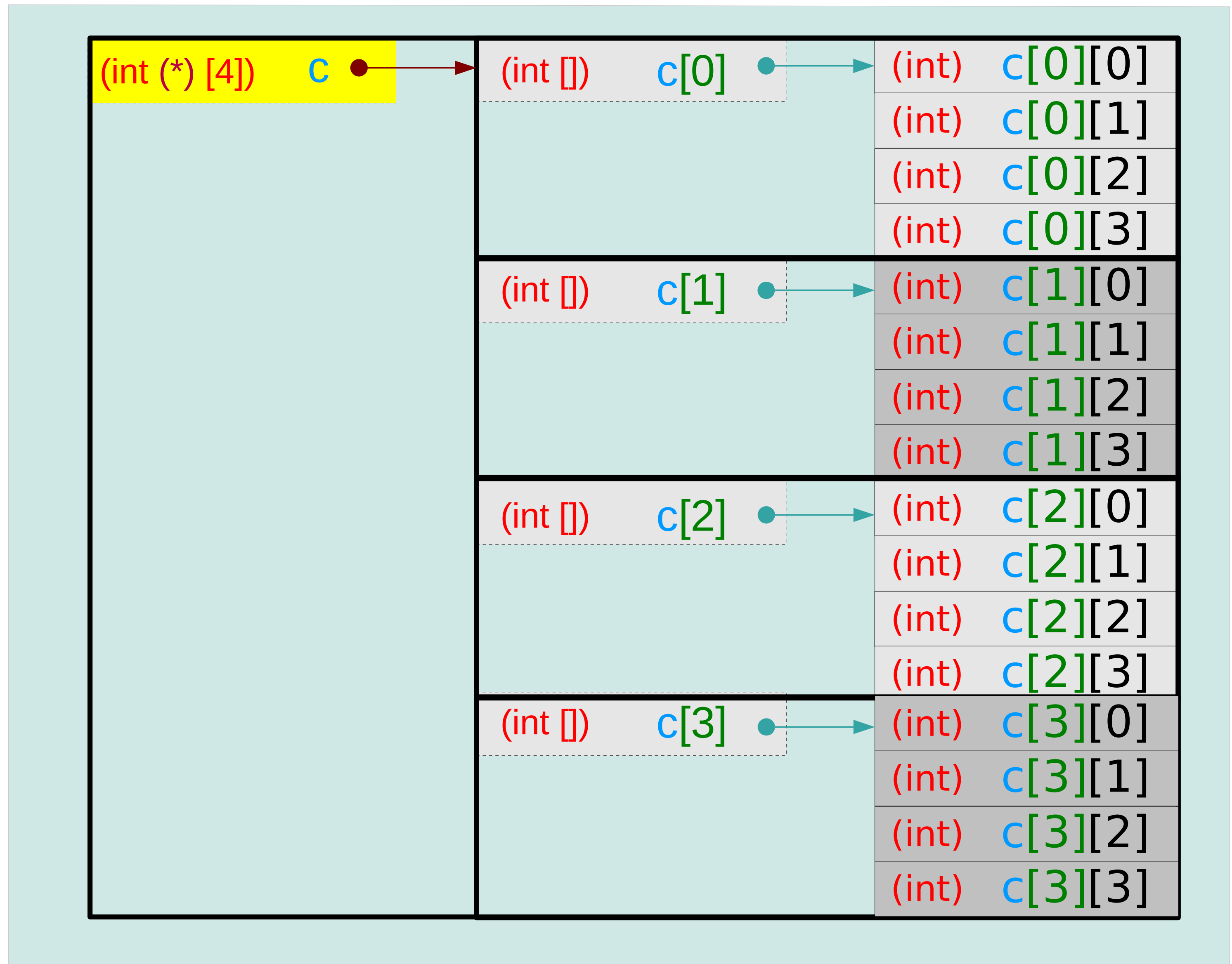
sizeof(c[0]) = 16 bytes



2-d array aggregated data type

2-d array :
sizeof(c) = 64 bytes

1-d sub-arrays :
sizeof(*c) = 16 bytes



2-d array name as a pointer to a 1-d subarray

1-d array pointer

(int (*) [4]) c

1-d array pointer

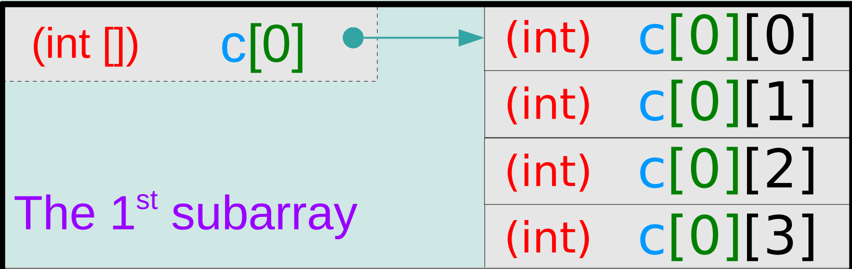
(int (*) [4]) c+1

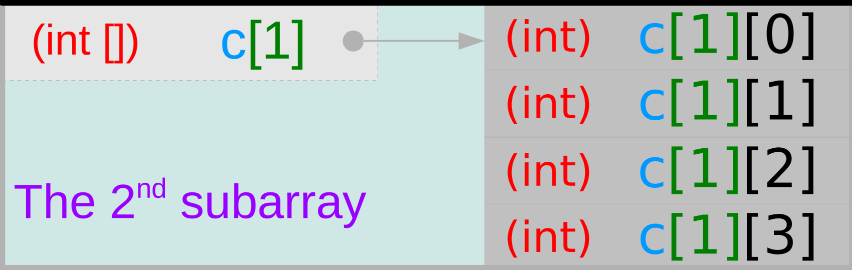
1-d array pointer

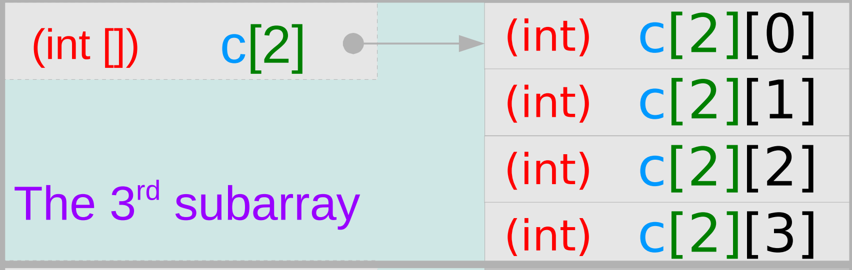
(int (*) [4]) c+2

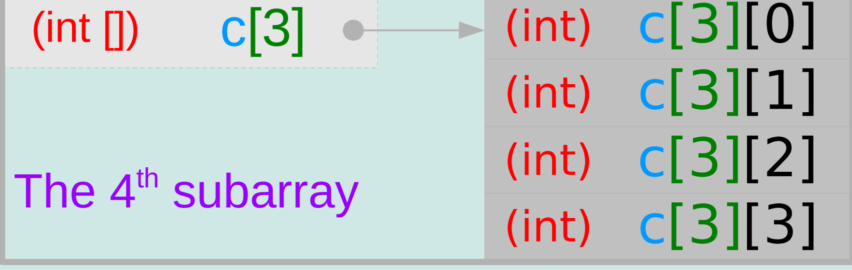
1-d array pointer

(int (*) [4]) c+3



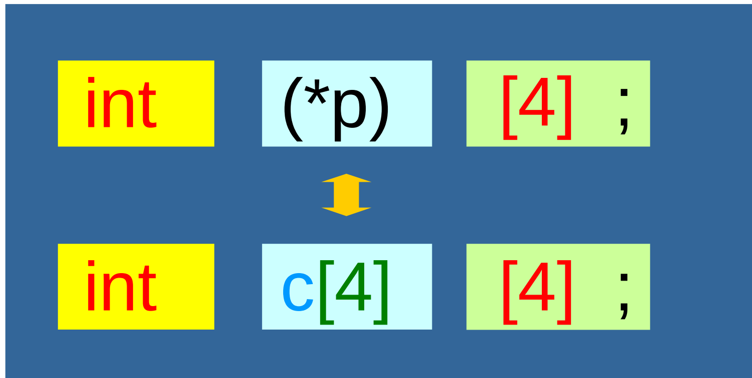




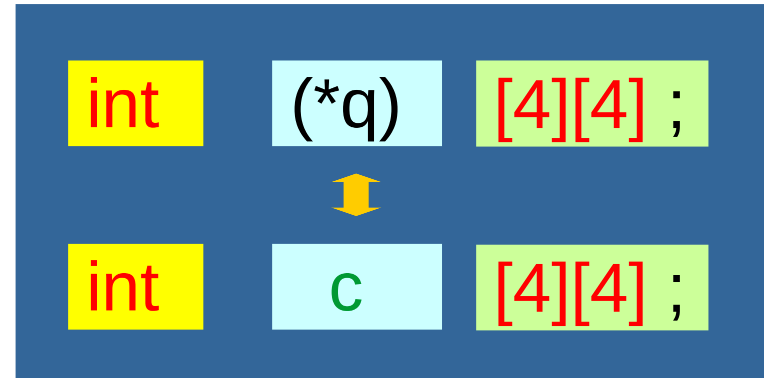


2-d array and 1-d and 2-d array pointers

1-d array pointer



2-d array pointer



(int (*) [4])

p = &c[0];

p = c;

(int(*)[4][4])

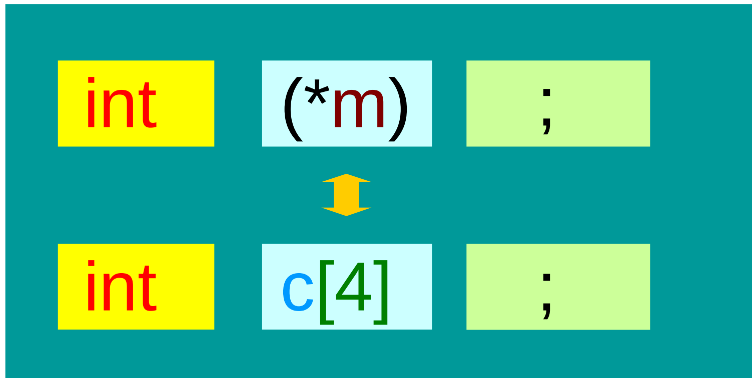
q = &c;

p[0] ≡ c[0]
p[1] ≡ c[1]
p[2] ≡ c[2]
p[3] ≡ c[3]

(*q)[0] ≡ q[0][0] ≡ c[0]
(*q)[1] ≡ q[0][0] ≡ c[1]
(*q)[2] ≡ q[0][0] ≡ c[2]
(*q)[3] ≡ q[0][0] ≡ c[3]

1-d array and 0-d and 1-d array pointers

0-d array pointer : int pointer



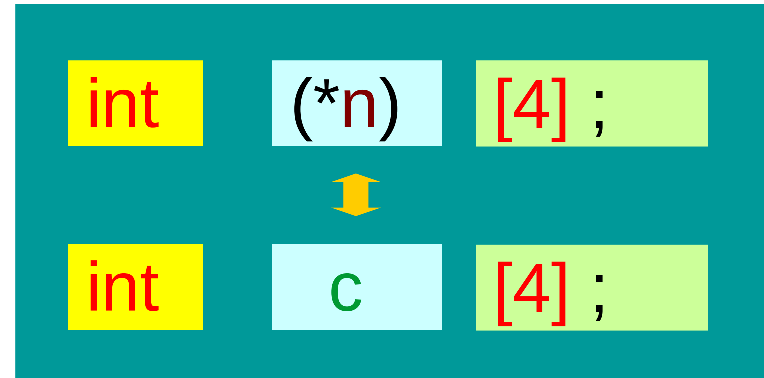
(int (*))

m = &c[0];

m = c;

m[0] ≡ c[0]
m[1] ≡ c[1]
m[2] ≡ c[2]
m[3] ≡ c[3]

1-d array pointer

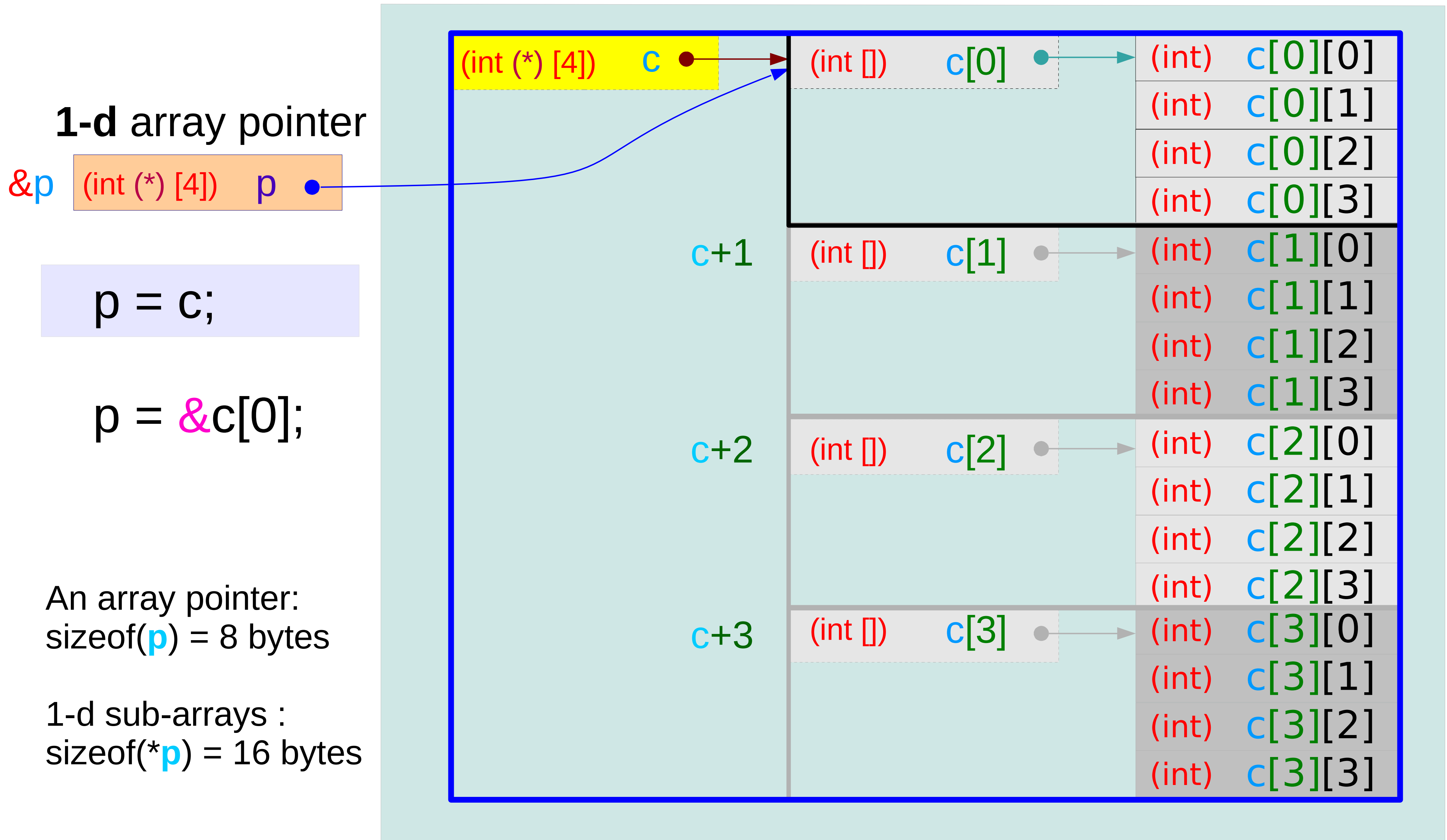


(int(*)[4])

n = &c;

(*n)[0] ≡ n[0][0] ≡ c[0]
(*n)[1] ≡ n[0][0] ≡ c[1]
(*n)[2] ≡ n[0][0] ≡ c[2]
(*n)[3] ≡ n[0][0] ≡ c[3]

2-d array and **1-d** array pointer

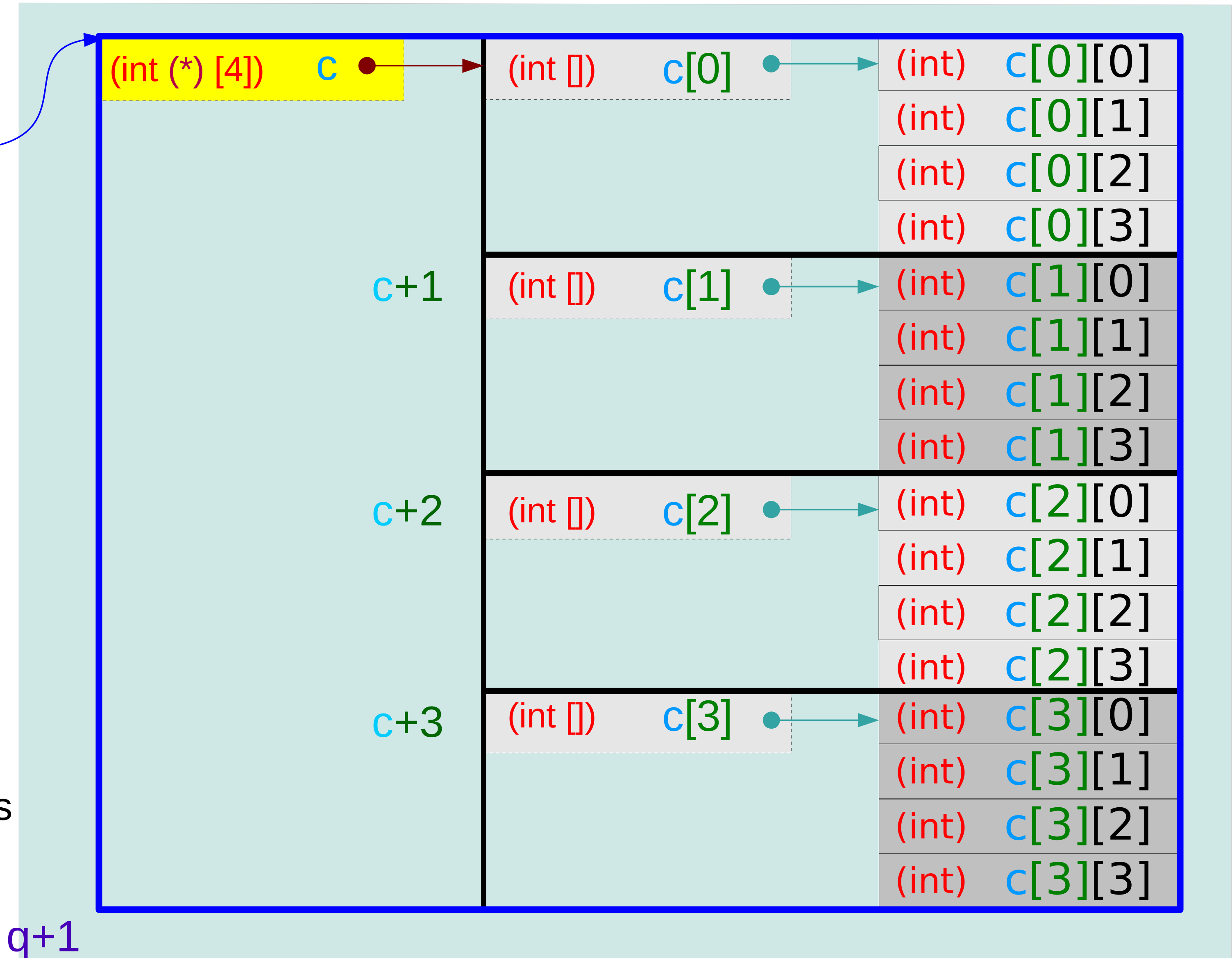


2-d array and 2-d array pointer

2-d array pointer

&q (int(*)[4][4]) q

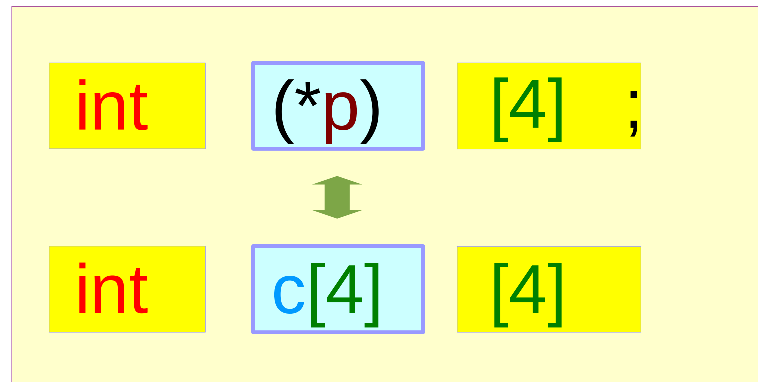
q = &c;



An array pointer:
sizeof(**q**) = 8 bytes

1-d sub-arrays :
sizeof(***q**) = 64 bytes

Using a **1-d** array pointer to a **2-d** array



1-d array pointer

&p (int (*) [4]) p

p = c;

p[0] ≡ c[0]
p[1] ≡ c[1]
p[2] ≡ c[2]
p[3] ≡ c[3]

a 2-d array with 4 rows and 4 columns

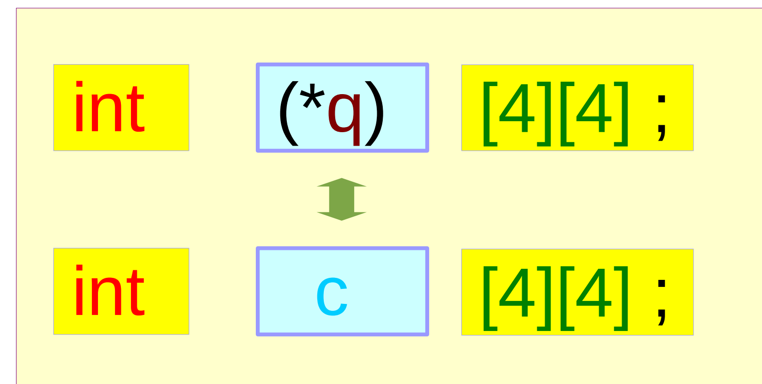
(int (*) [4]) c

p

p[0]
p[1]
p[2]
p[3]

p[0][0]
p[0][1]
p[0][2]
p[0][3]
p[1][0]
p[1][1]
p[1][2]
p[1][3]
p[2][0]
p[2][1]
p[2][2]
p[2][3]
p[3][0]
p[3][1]
p[3][2]
p[3][3]

Using a **2-d** array pointer to a **2-d** array



2-d array pointer

&p (int(*)[4][4]) q

q = &c;

(*q)[0] ≡ c[0]
(*q)[1] ≡ c[1]
(*q)[2] ≡ c[2]
(*q)[3] ≡ c[3]

a 2-d array with 4 rows and 4 columns

(int (*) [4]) c

*q

(*q)[0]
(*q)[1]
(*q)[2]
(*q)[3]

(*q)[0][0]
(*q)[0][1]
(*q)[0][2]
(*q)[0][3]
(*q)[1][0]
(*q)[1][1]
(*q)[1][2]
(*q)[1][3]
(*q)[2][0]
(*q)[2][1]
(*q)[2][2]
(*q)[2][3]
(*q)[3][0]
(*q)[3][1]
(*q)[3][2]
(*q)[3][3]

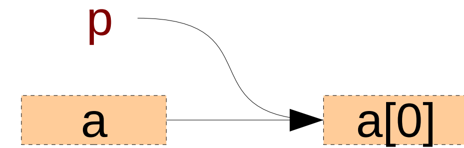
Pointer to multi-dimensional arrays (1)

```
int a[4] ;              1-d array
int (*p) ;              0-d array pointer

int b[4]  [2];          2-d array
int (*q)  [2];          1-d array pointer

int c[4]  [2][3];       3-d array
int (*r)  [2][3];       2-d array pointer

int d[4]  [2][3][4];    4-d array
int (*s)  [2][3][4];    3-d array pointer
```

Pointer to multi-dimensional arrays (2)

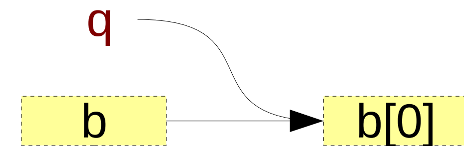
```
int a[4] ;
int (*p) ;
```

```
p = &a[0];
p = a;
```



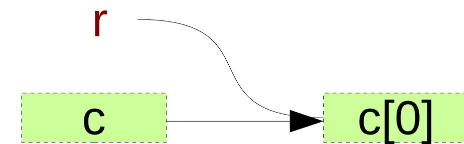

```
int b[4]  [2];
int (*q)  [2];
```

```
q = &b[0];
q = b;
```



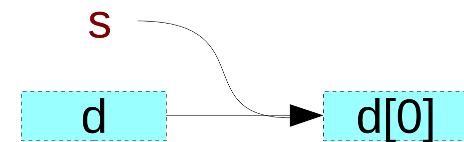

```
int c[4]  [2][3];
int (*r)  [2][3];
```

```
r = &c[0];
r = c;
```

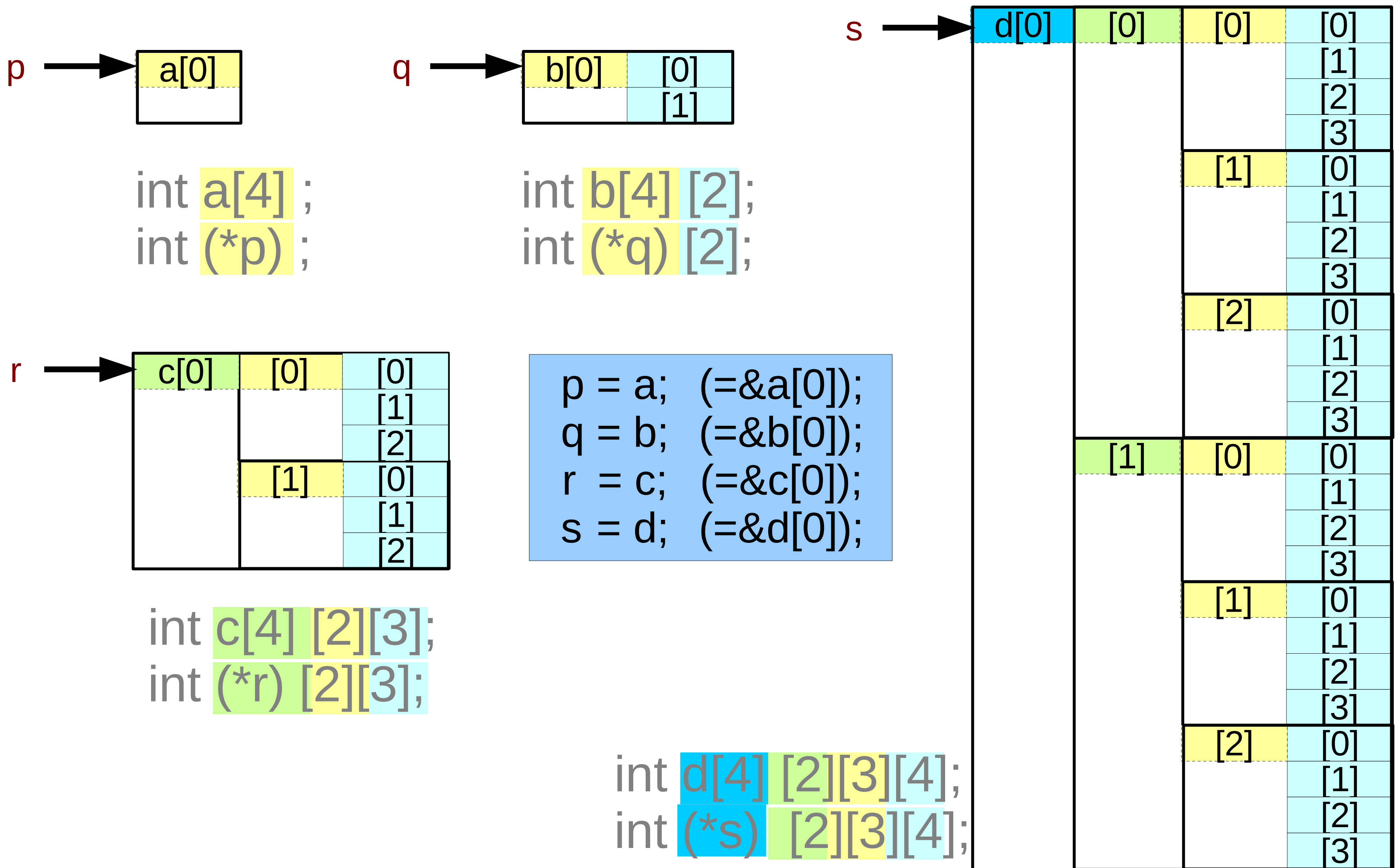



```
int d[4]  [2][3][4];
int (*s)  [2][3][4];
```

```
s = &d[0];
s = d;
```



Pointer to multi-dimensional arrays (3)



To pass array name

e
ll
e
ll
e
ll

	call	prototype
int a[4] ; int (*p) ;	funa(a, …);	void funa(int (*p), …);
int b[4] [2]; int (*q) [2];	funb(b, …);	void funb(int (*q)[2], …);
int c[4] [2][3]; int (*r) [2][3];	func(c, …);	void func(int (*r)[2][3], …);
int d[4] [2][3][4]; int (*s) [2][3][4];	fund(d, …);	void fund(int (*s)[2][3][4], …);

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