Applications of Pointers (1A)

Young Won Lim
6/7/18

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.

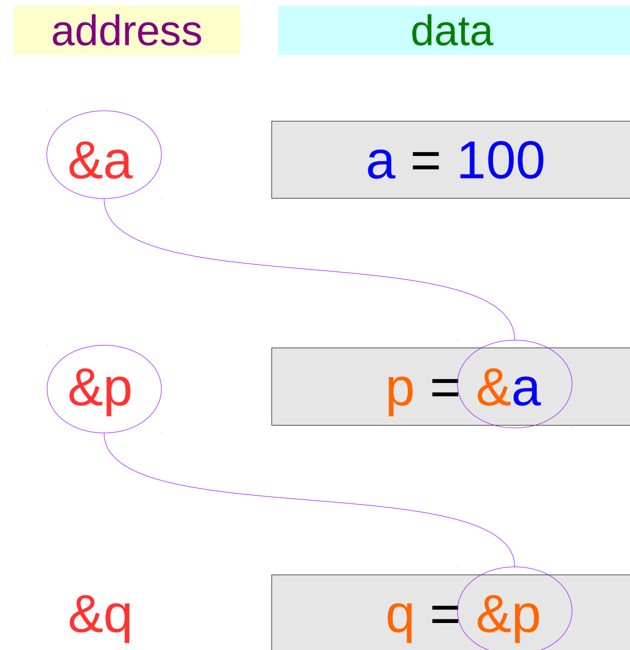
This document was produced by using LibreOffice.

Variables and their addresses

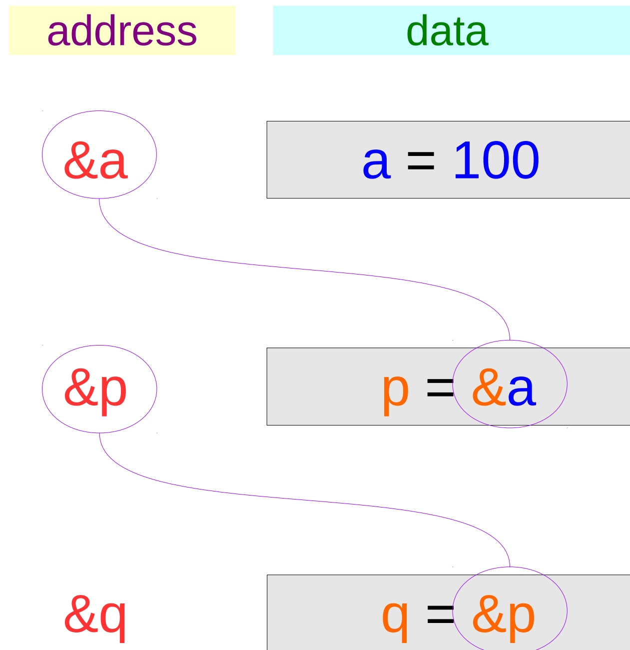
	address	data
int a;	&a	a
int * p;	&p	p
int ** q;	&q	q

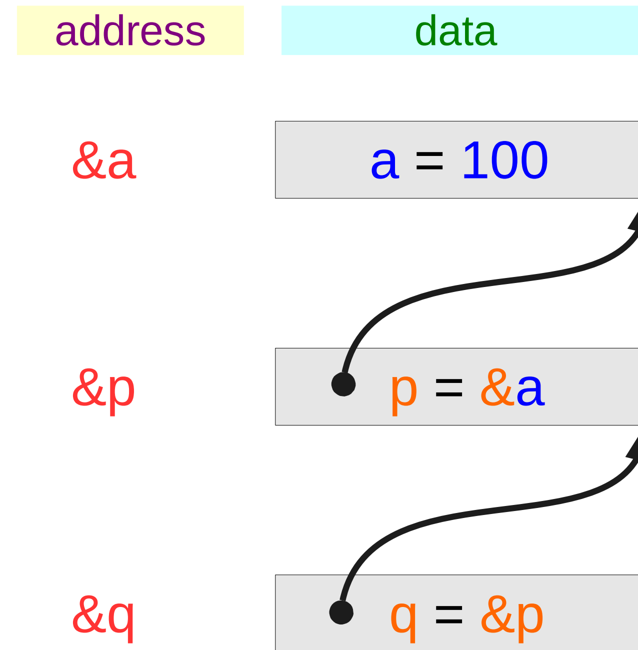
Initialization of Variables

```
int     a = 100;

int   * p = &a;

int  ** q = &p;
```



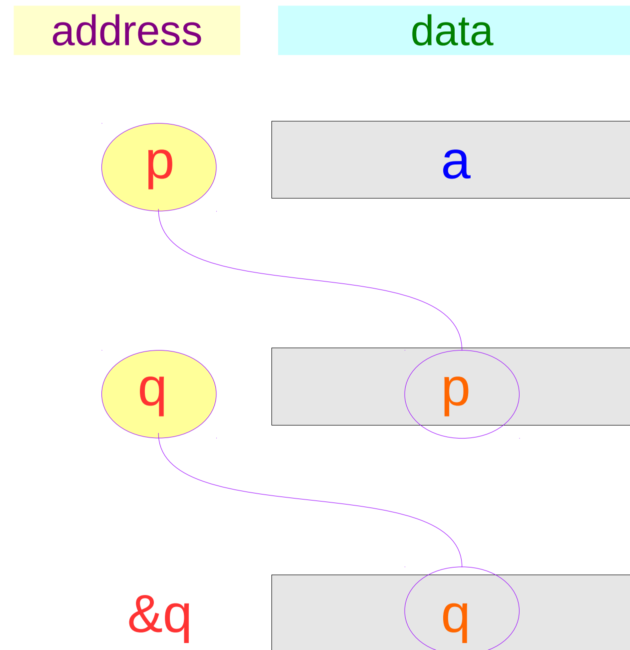
Traditional arrow notations





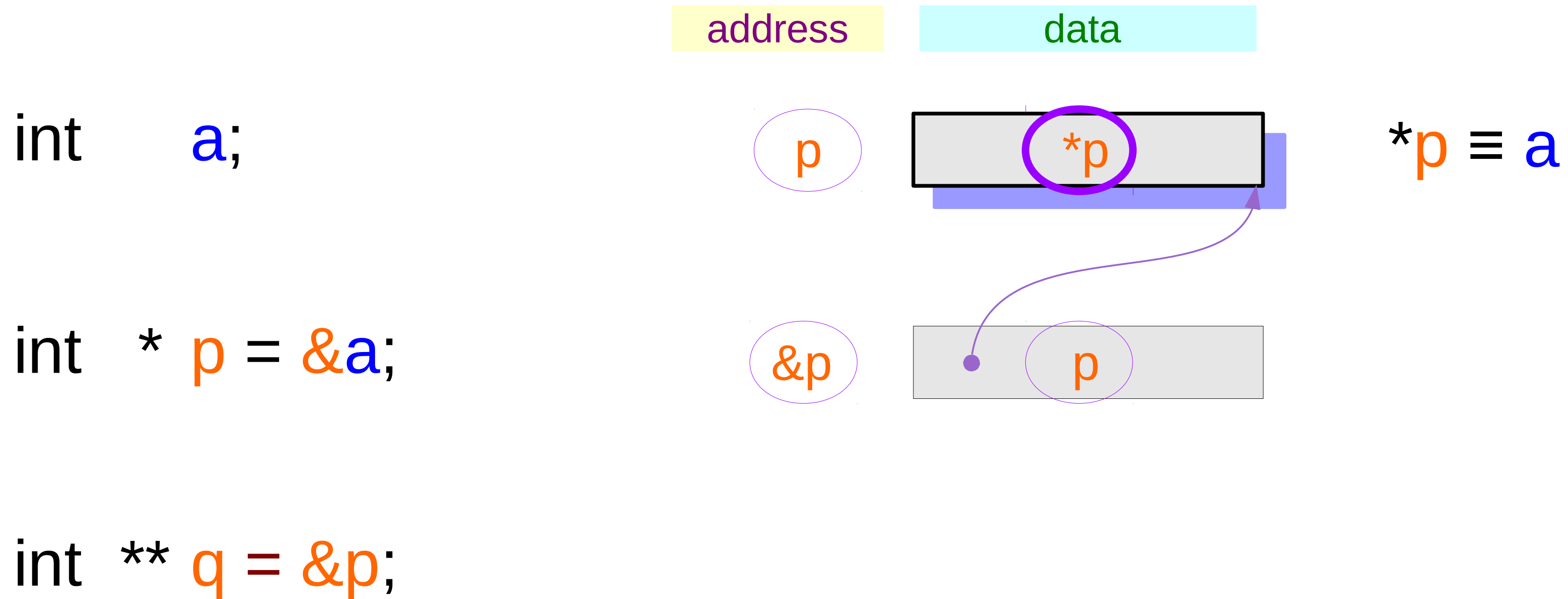
Pointed addresses : p, q

```
int     a;

int   * p = &a;

int  ** q = &p;
```



p = &a
q = &p

Dereferenced Variables : *p



Dereferenced Variables : *p

```
int     a;

int   * p = &a;

int  ** q = &p;
```

Address assignment

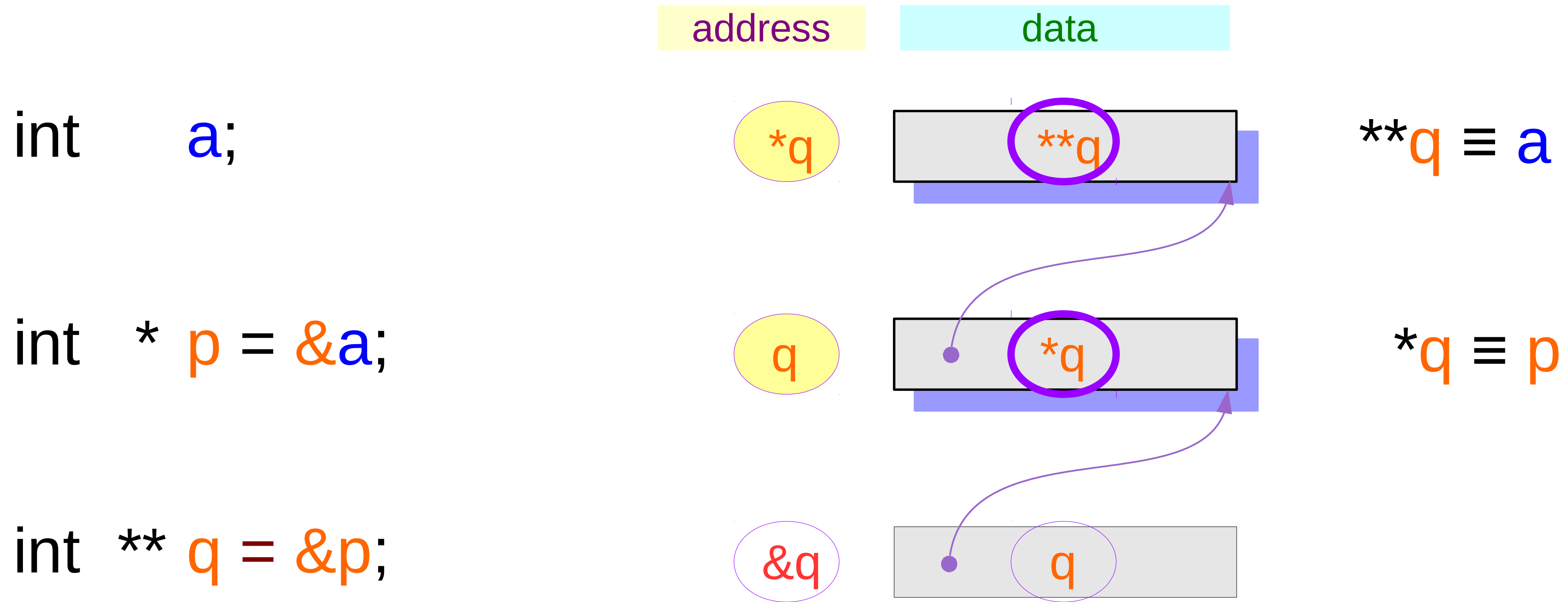
Variable aliasing

p = &a ➡ *p ≡ a

```
  p  ≡   &a
*(p) ≡ *(&a)
* p  ≡    a
```

Relations after address assignment

Dereferenced Variables : *q, **q



Dereferenced Variables : *q, **q

```
int     a;

int   * p = &a;

int  ** q = &p;
```

Address assignment

Variable aliasing

p = &a ➡ *p ≡ a

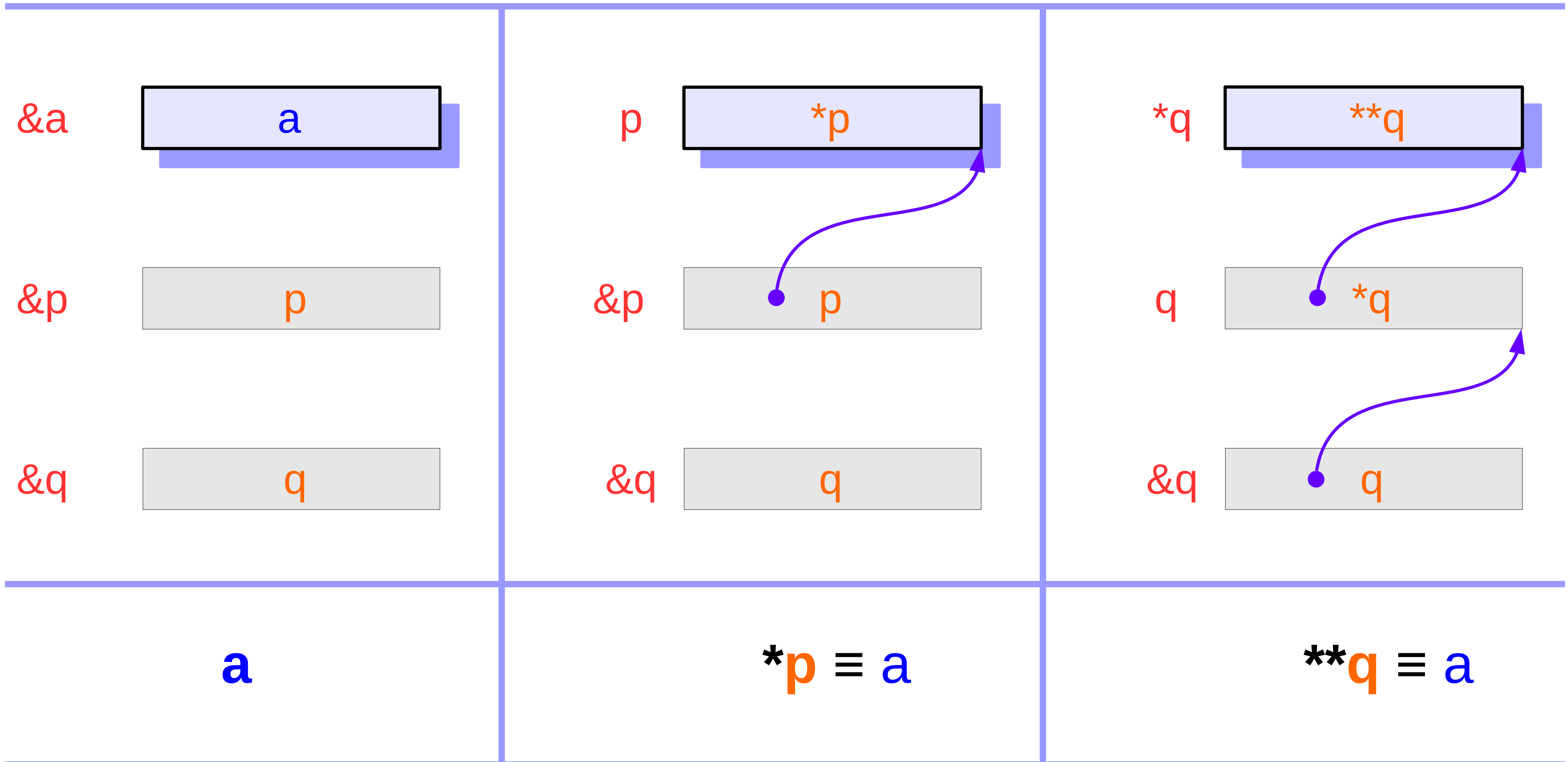
q = &p ➡ *q ≡ p

➡ **q ≡ a

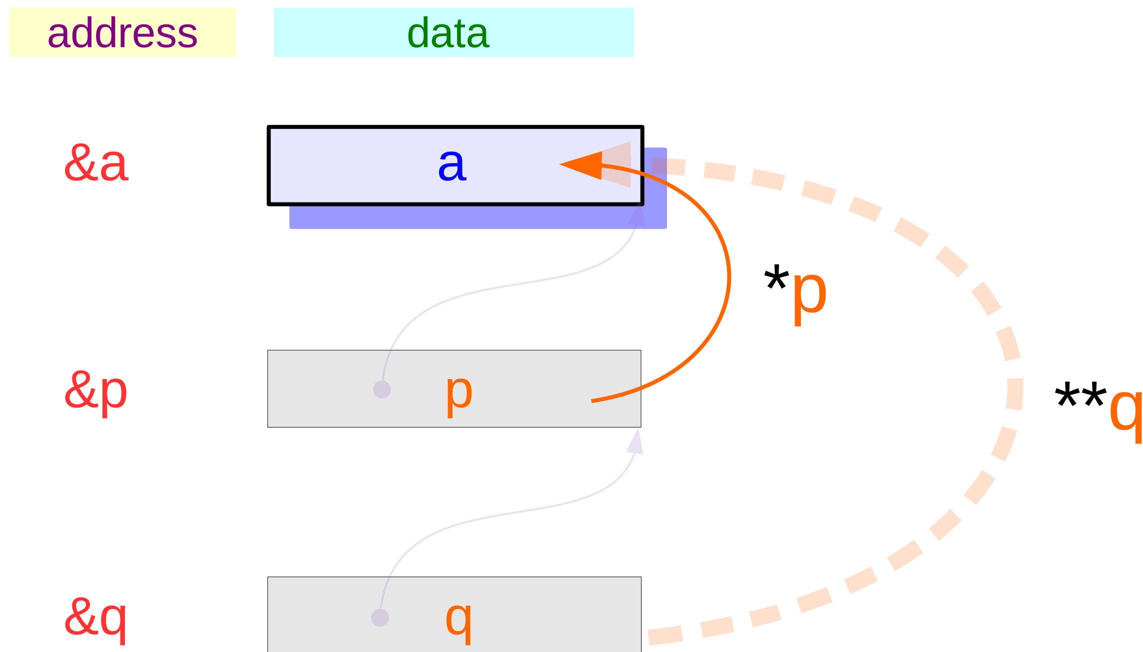
q	≡	&p
*(q)	≡	*(&p)
* q	≡	p
**q	≡	*p
**q	≡	a

Relations after address assignment

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



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



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

Variables

```
int      a;
```

a can hold an *integer*

address	data
&a	a

```
a = 100;
```

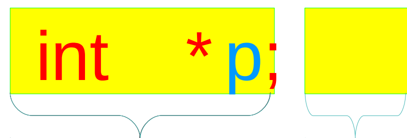
a holds 100

address	data
&a	a ⬅ 100

Pointer Variables

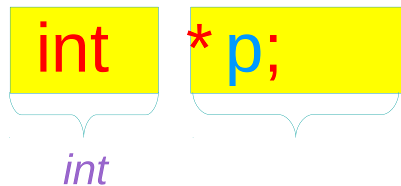
int * p;

p can hold an ***address***

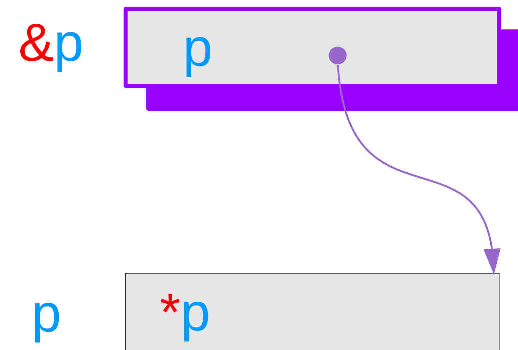


pointer to int

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



***p** holds
a int type ***data***



Pointer to Pointer Variable

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;

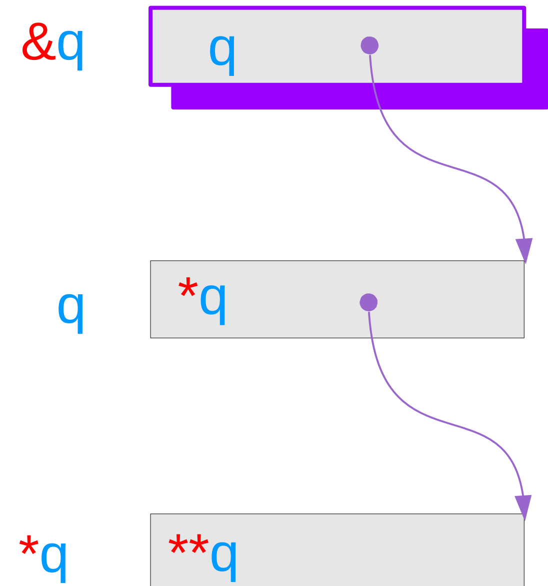
pointer to int

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

int **q;

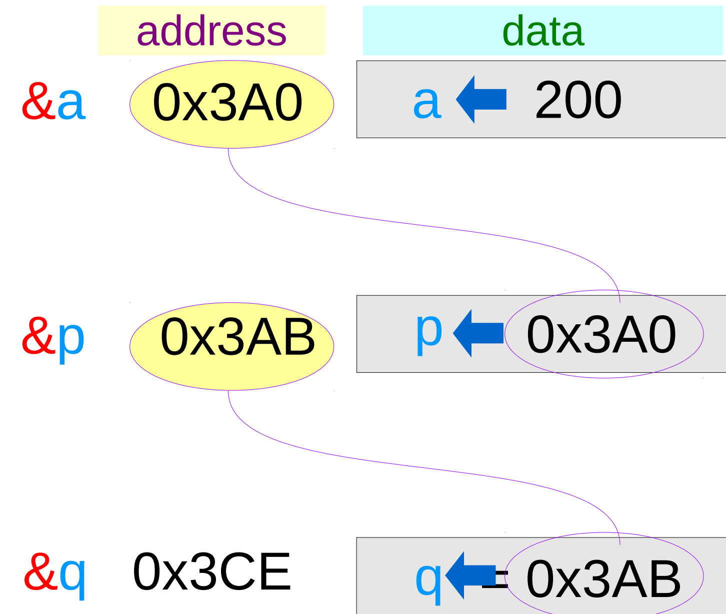
int

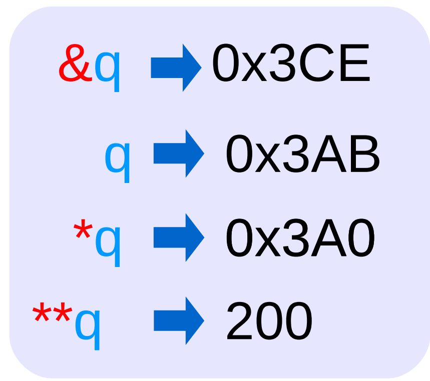
****q** holds a int type data



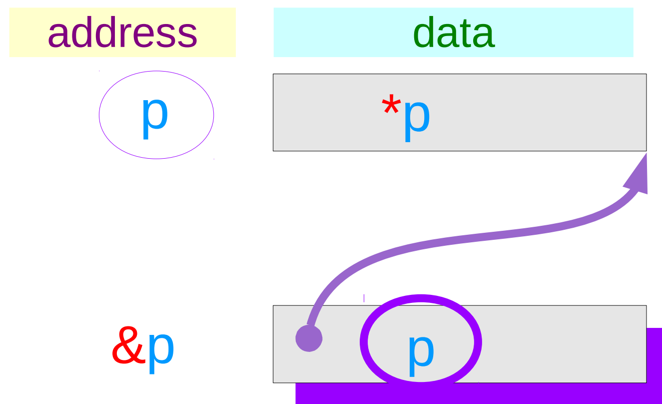
Pointer Variables Examples

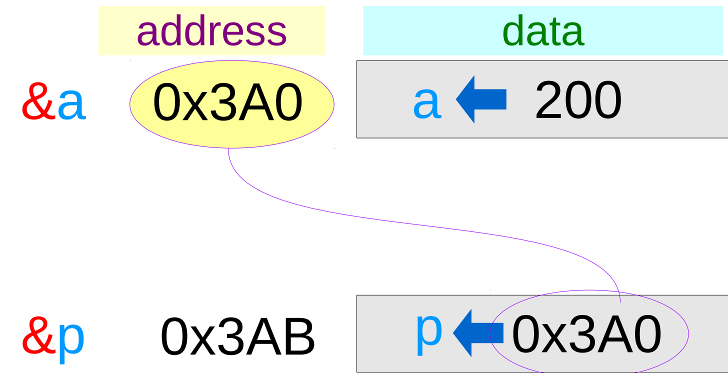
```
int      a = 200;

int *       p = & a;

int **      q = & p;
```



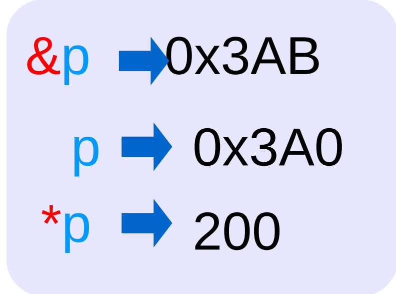


Pointer Variable **p** with an arrow notation

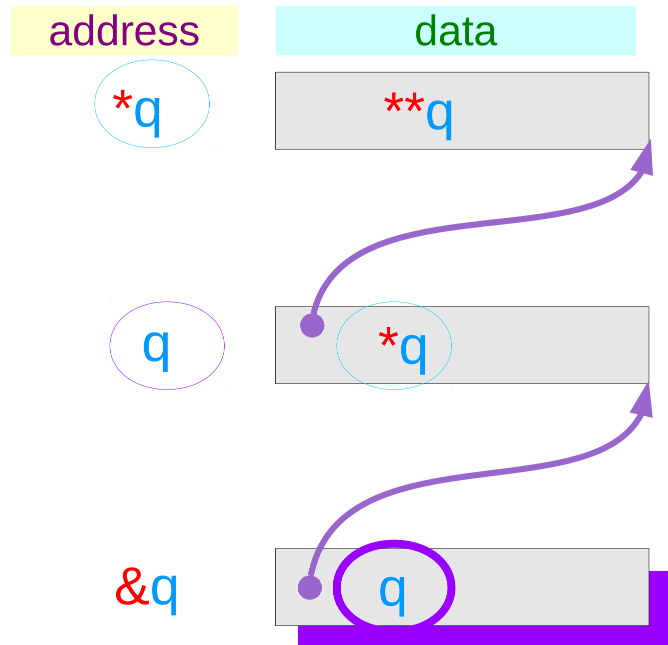




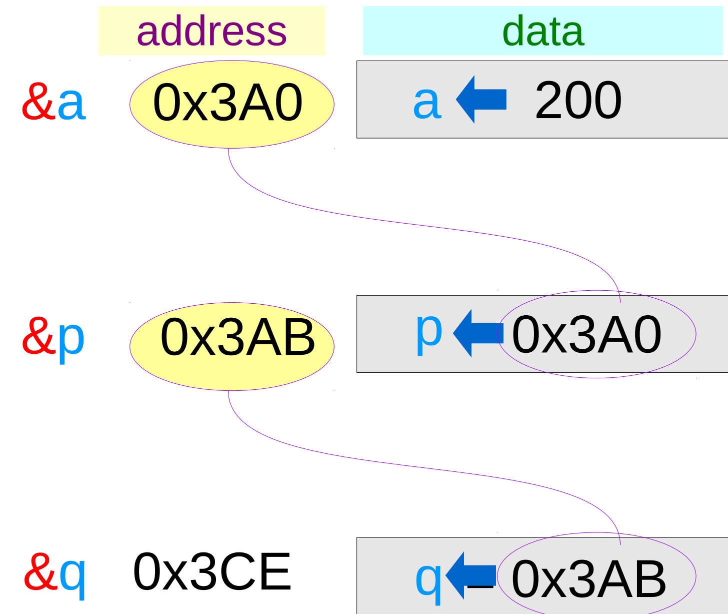
using an arrow notation



Pointer Variable **q** with an arrow notation

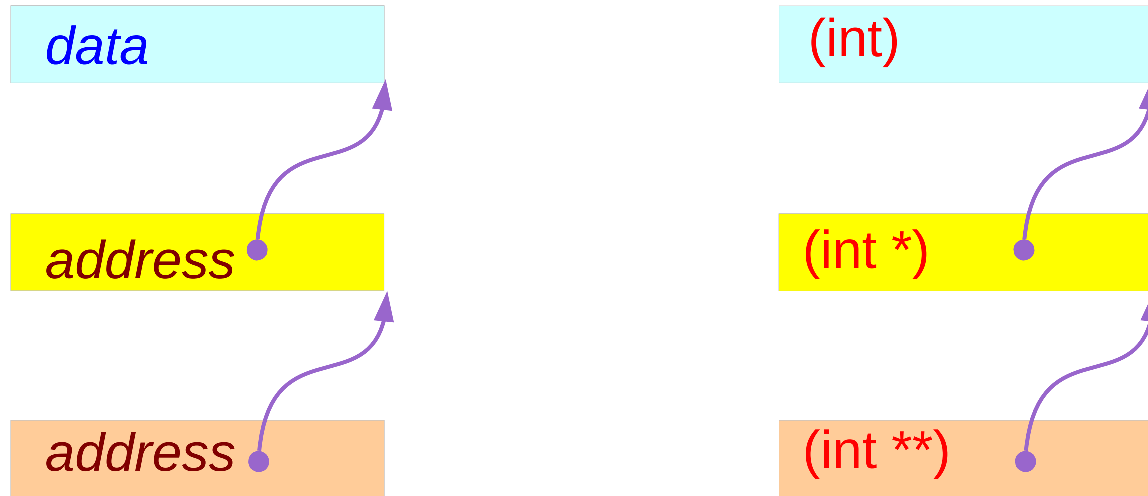


using an arrow notation



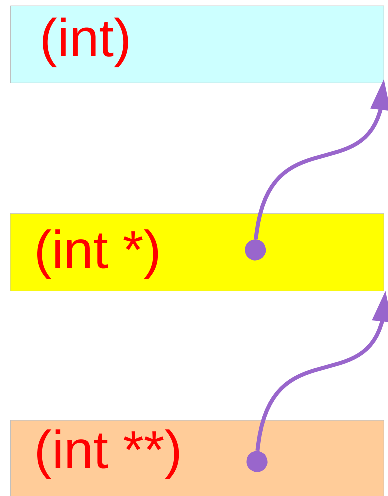
&q ➡ 0x3CE
q ➡ 0x3AB
*q ➡ 0x3A0
**q ➡ 200

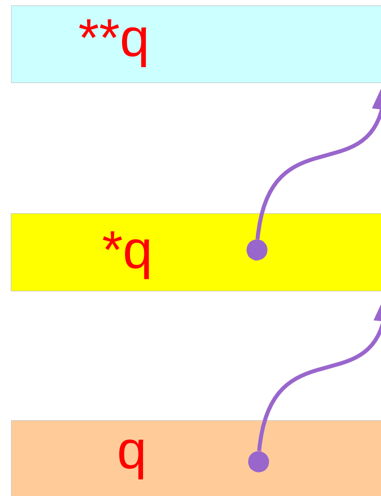
The type view point of pointers

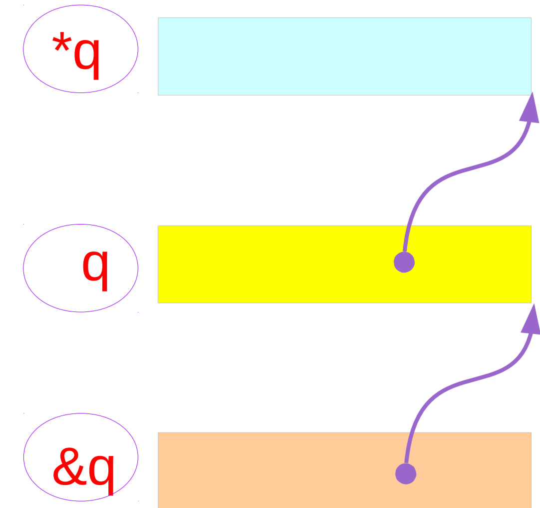


Types

The different view points of pointers



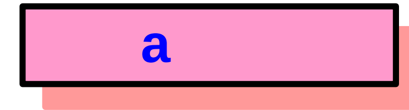


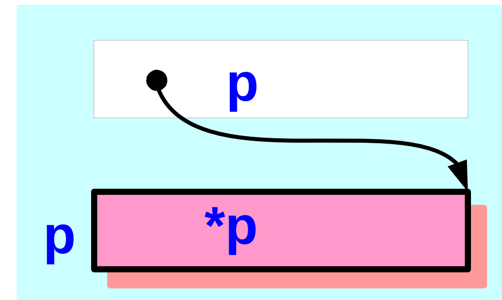


Types **Variables** **Addresses**

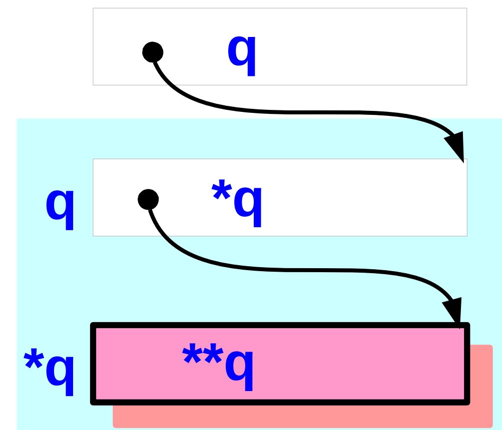
Single and Double Pointer Examples (1)

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



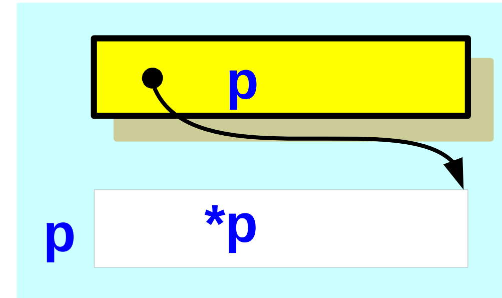


a, ***p**, and ****q**:
int variables

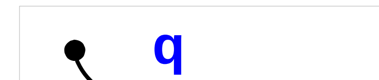


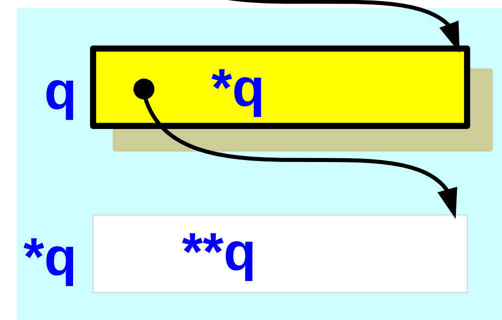
Single and Double Pointer Examples (2)

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



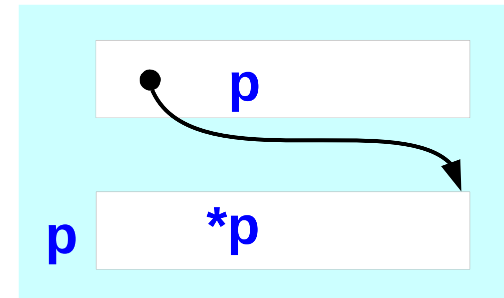
p and ***q** :
int pointer variables
(singlepointers)



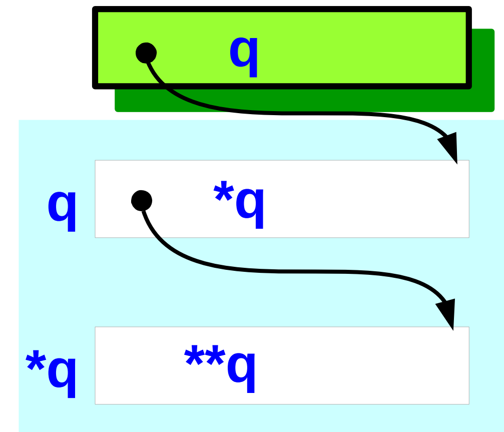


Single and Double Pointer Examples (3)

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

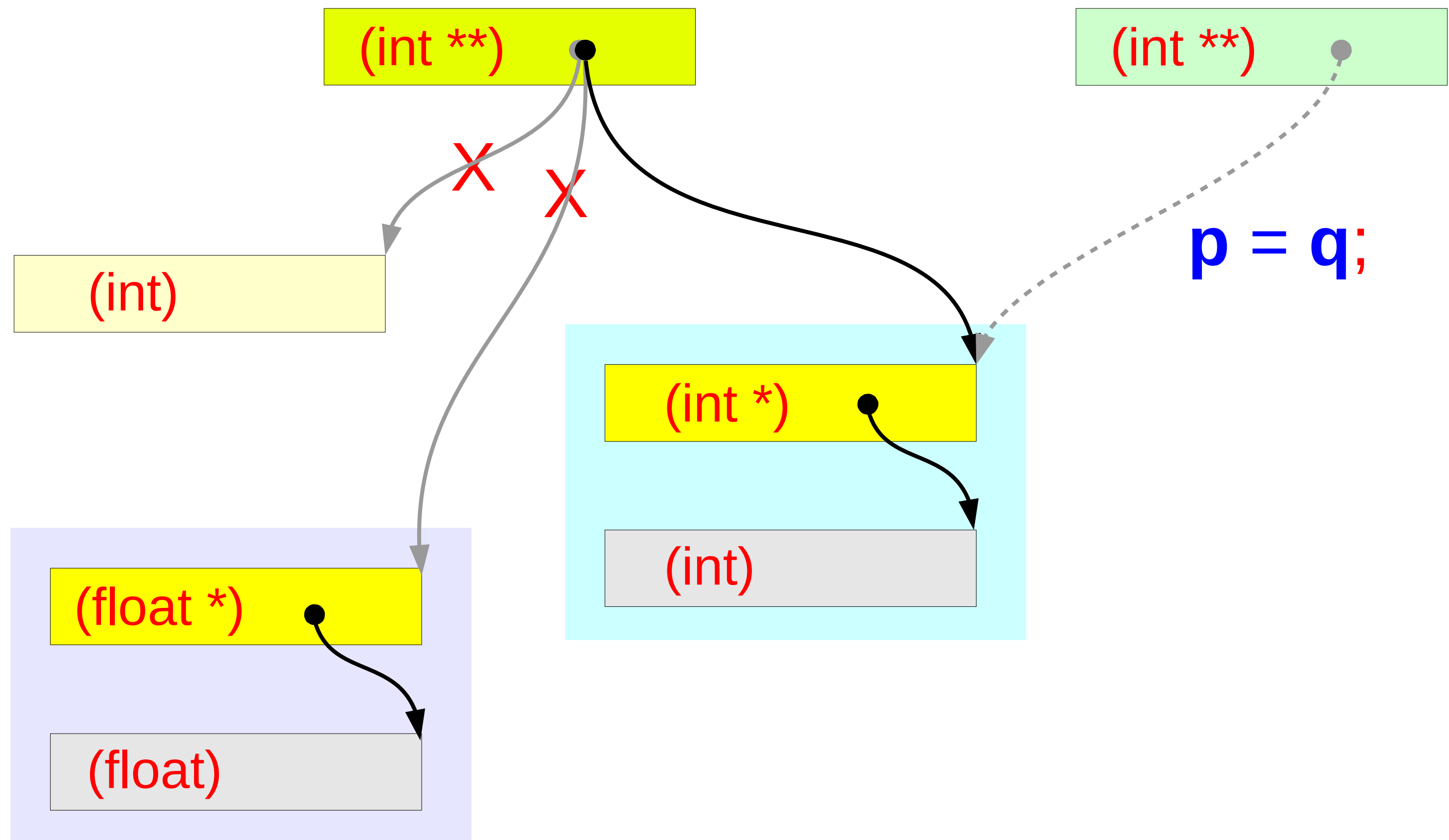


q :
double int pointer variables

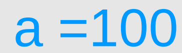


Values of double pointer variables

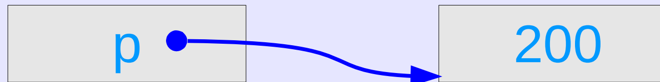
```
int ** p, **q ;
```



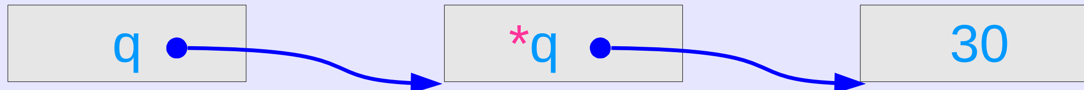
Pointed Addresses and Data

int a ; &a 

The variable a holds an integer data

int * p ; &p 

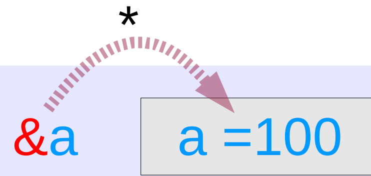
The **pointer** variable p holds an address,
at this address, an integer data is stored

int * * q ; &q 

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

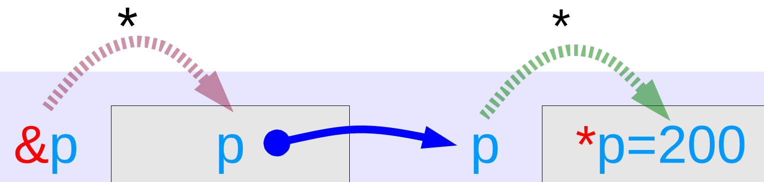
Dereferencing Operations

int a



*(&a) = a

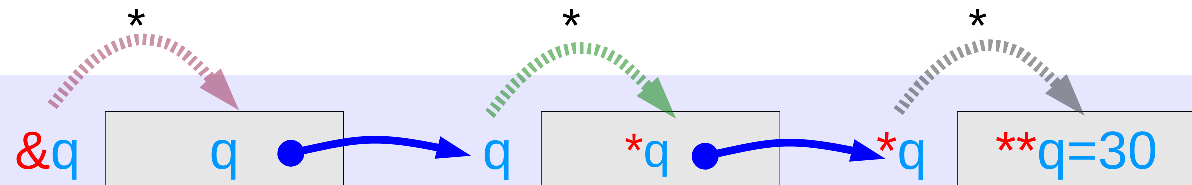
int * p



*(&p) = p

*(p) = *p

int * * q



*(&q) = q

*(q) = *q

*(*q) = **q

Direct Access to an integer **a**

int a ;

&a | a =100

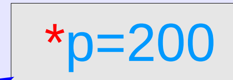
address	value	
&a	a	integer

Direct Access

1 memory access

Indirect Access *p to an integer a

int * p ;

&p p 

address	value
&p	p

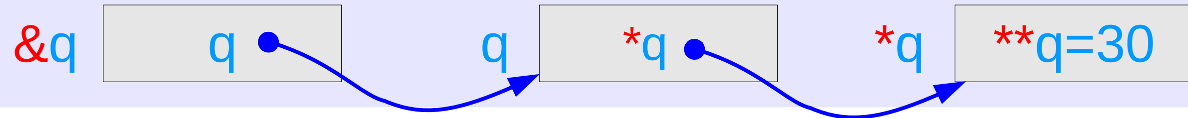
Indirect Access

2 memory accesses

Dereference Operator *
the content of the pointed location

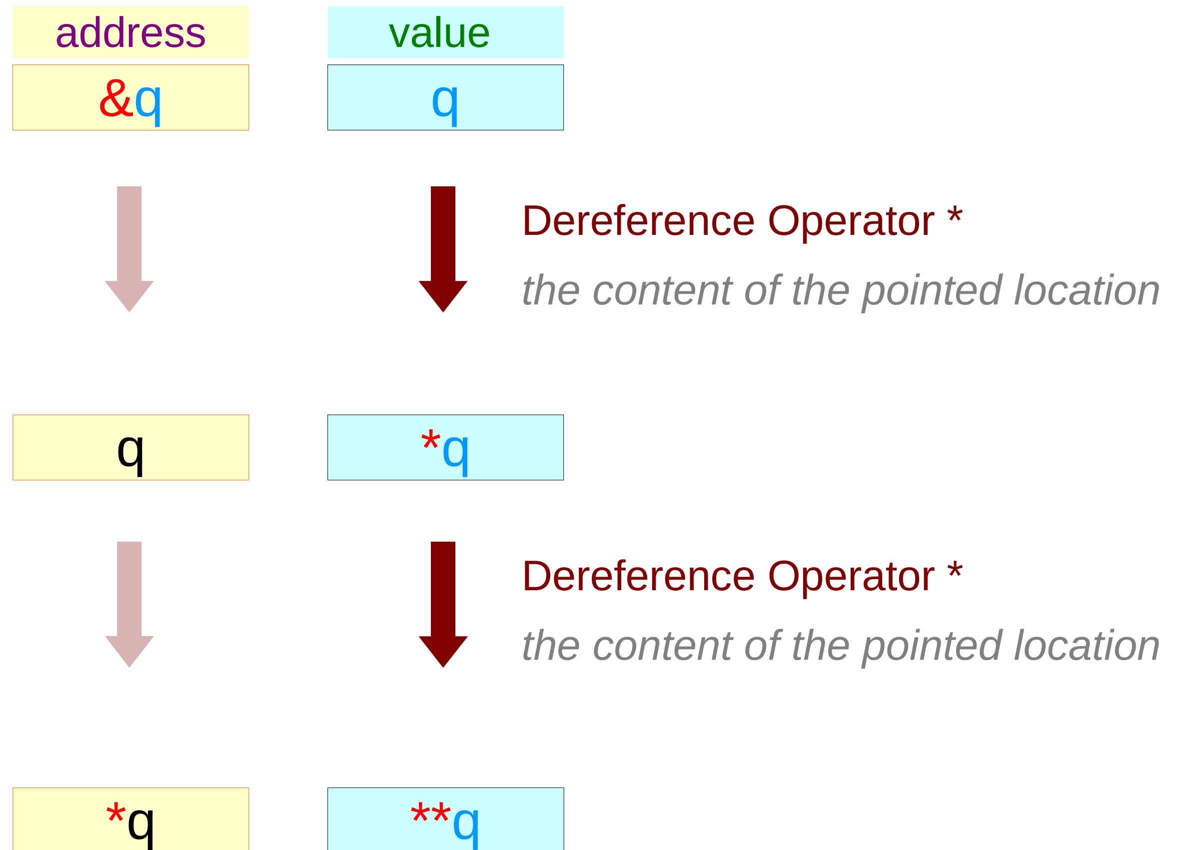
p	*p

Double Indirect Access **q to an integer a



Double Indirect Access

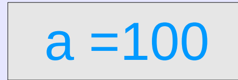
3 memory accesses

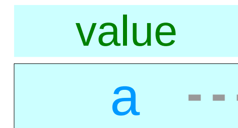


Values of Variables

int a ;

&a

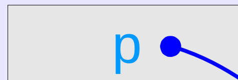




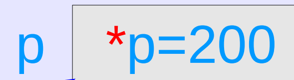
integer

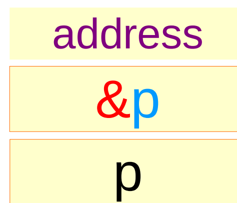
int * p ;

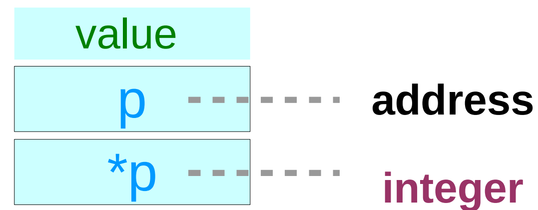
&p



p

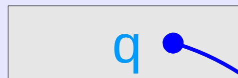




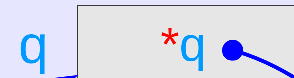


int * * q ;

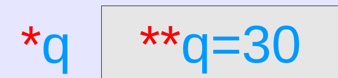
&q

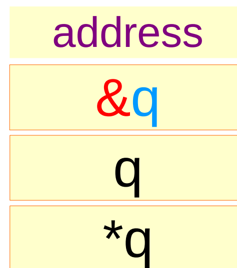


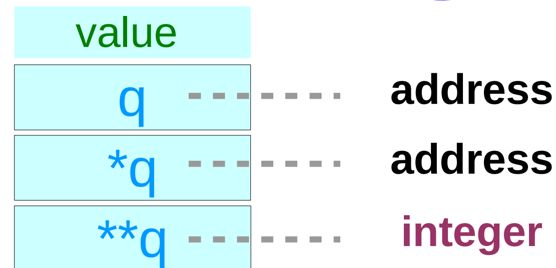
q



*q



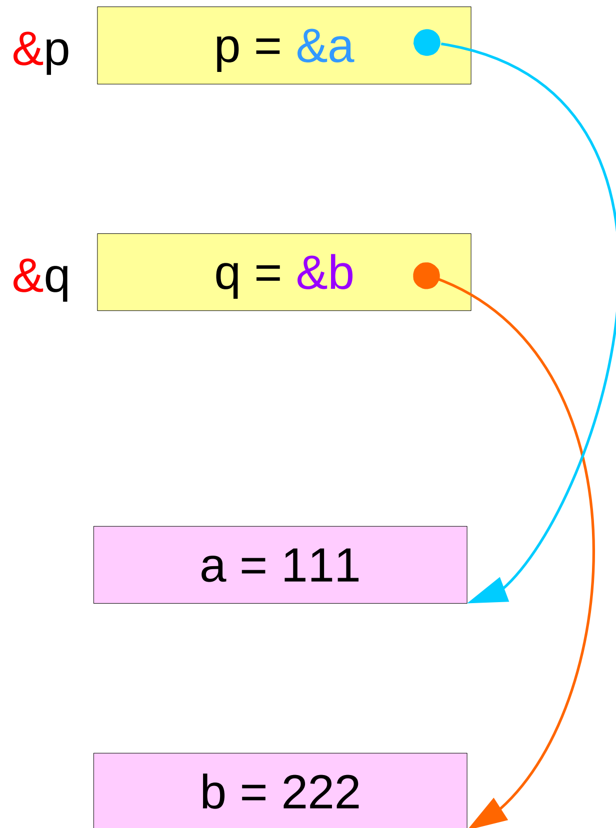


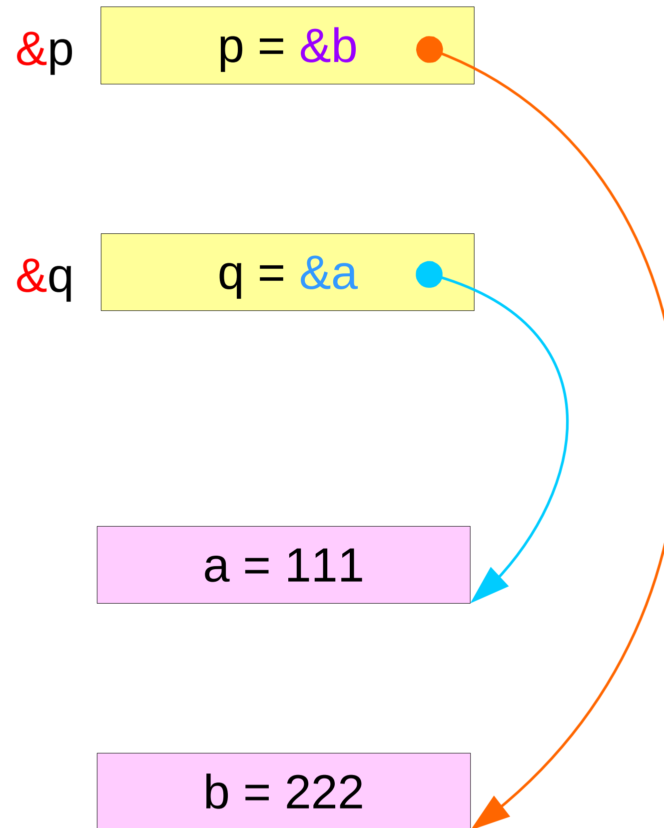


Swapping pointers

- pass by reference
- double pointers

Swapping integer pointers



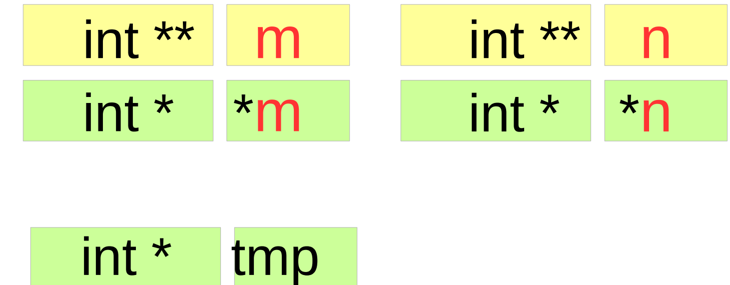


Swapping integer pointers

```
int *p, *q;

swap_pointers( &p, &q );              function call

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;
}
```

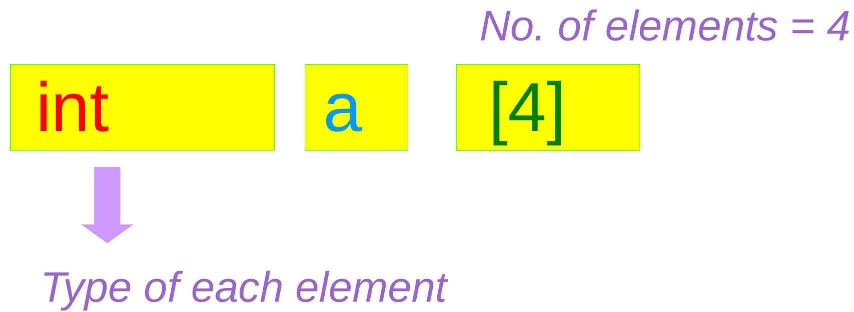


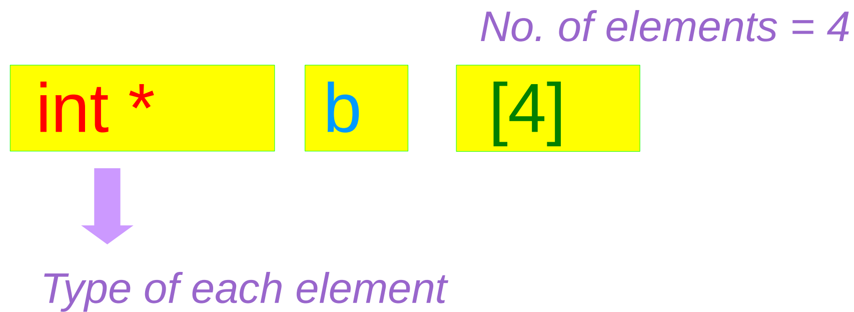

```
int   a,  b;
int *p, *q;        p=&a, q=&b;
     …
swap_pointers( &p, &q );
```

Array of Pointers

Array of Pointers

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

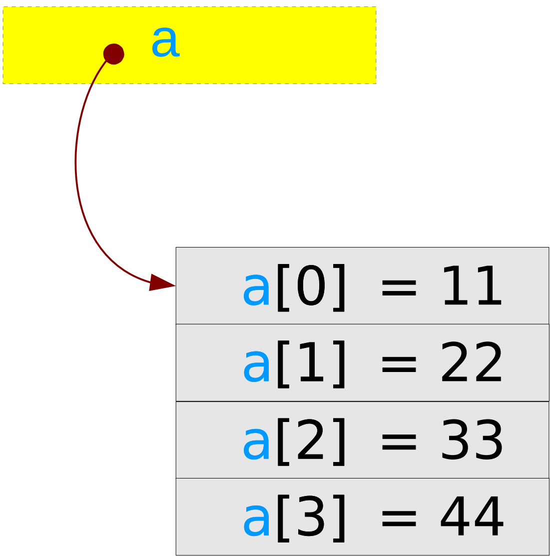


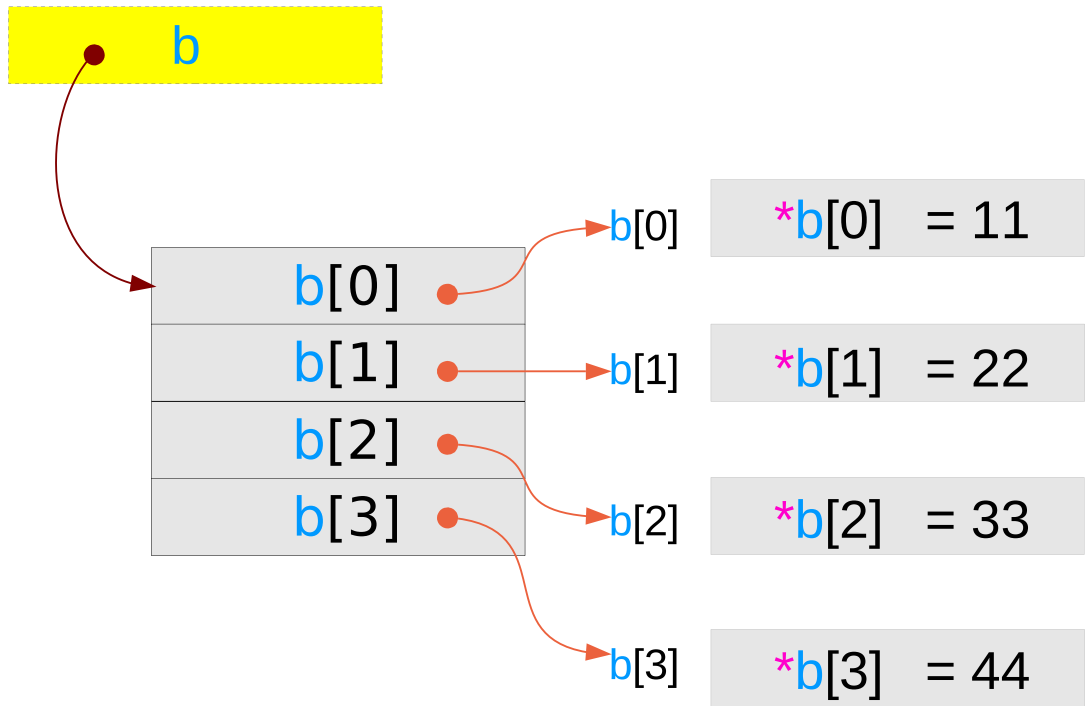


Array of Pointers – variable view

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

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

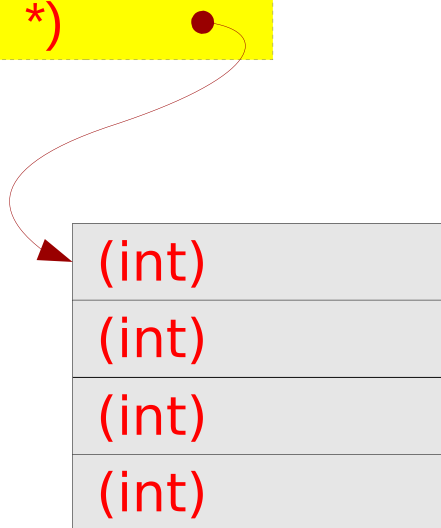


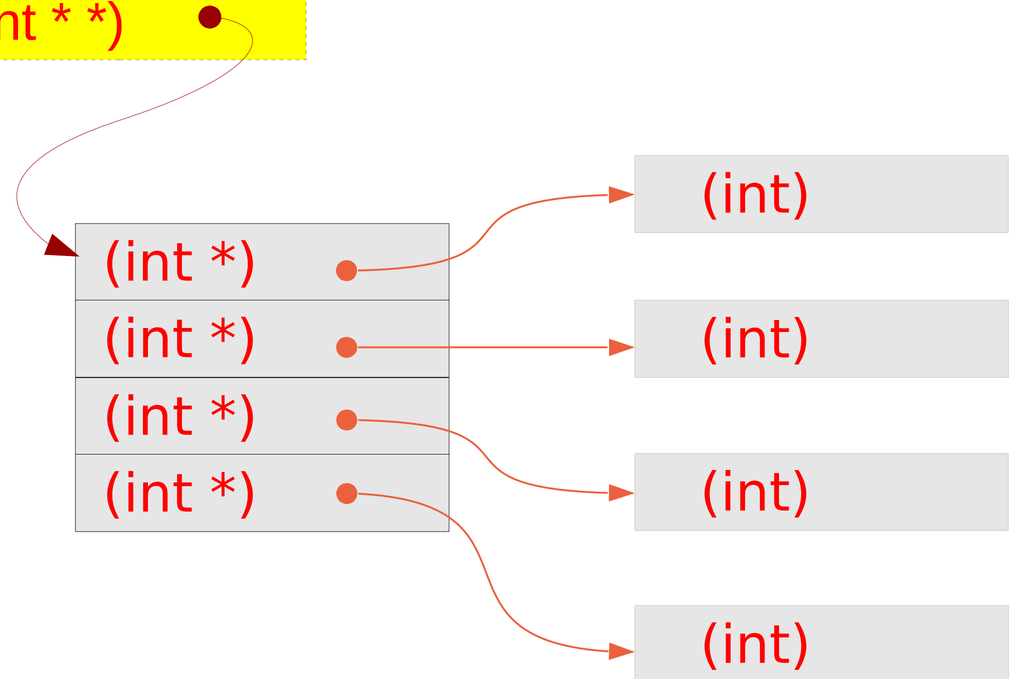


Array of Pointers – type view

int a [4];

int * b [4];





Pointer to Arrays

Pointer to an array – variable declarations

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

an integer pointer

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

an integer 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

int *

an integer pointer

Int [4] ≡ int [ ]

int (*) [4]

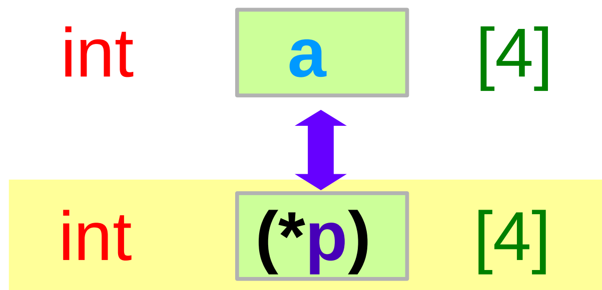
an integer array pointer

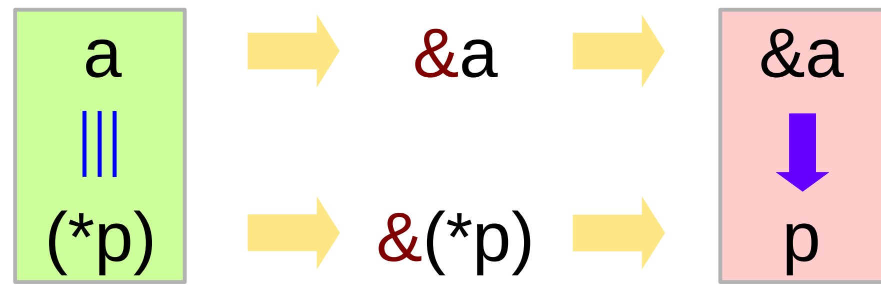
int (int, int)

int (*) (int, int)

a function pointer

Pointer to an Array : Assignment and Dereference



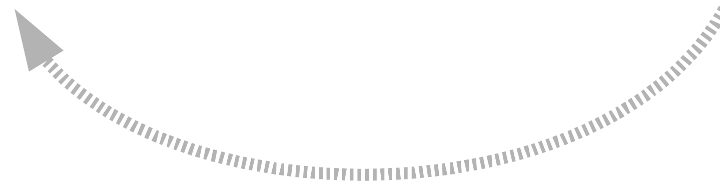


equivalence

usages

assignment

initialization



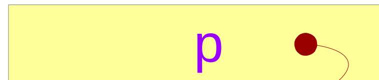
sizeof(p)= 8 bytes : the size of a pointer

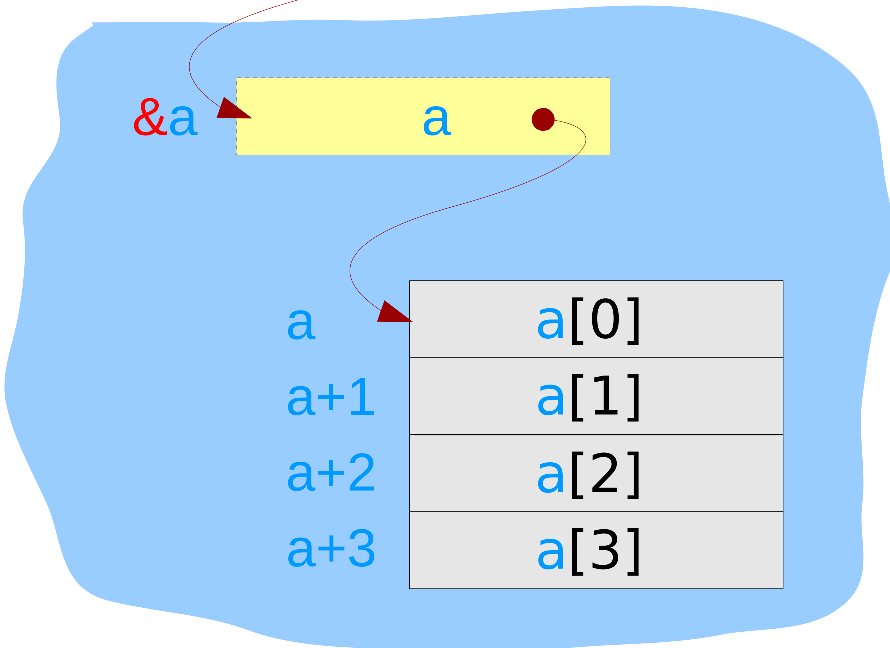
sizeof(*p)= 16 bytes : the whole size of the pointed array

Pointer to an array – a variable view

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

an array pointer points to an array –
a aggregated type data





assignment

p = &a

equivalence

***p ≡ a**

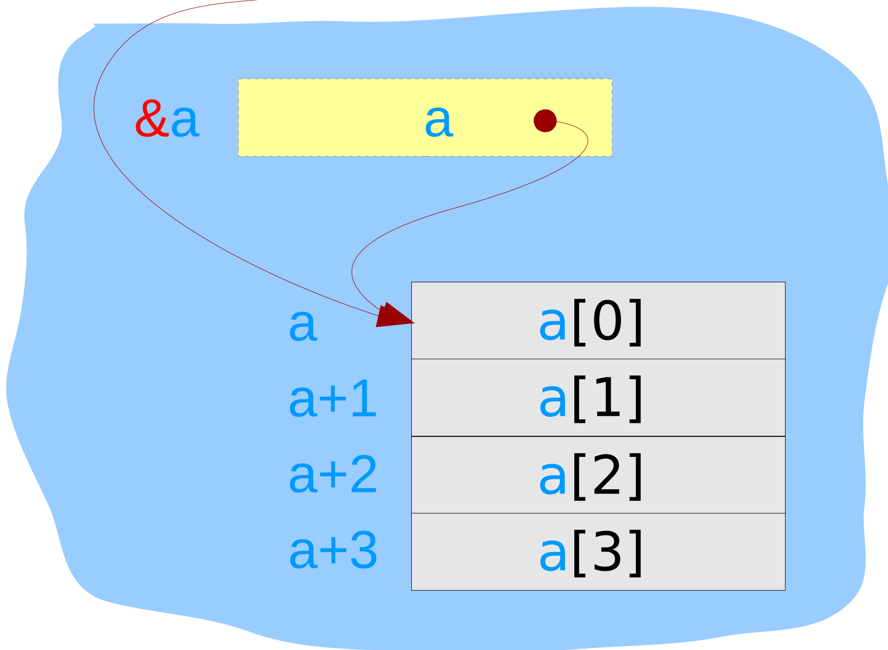
```
int a [4];
```

p : int (*) [4] type

Pointer to an array – a variable view

int (*q) ;

an array pointer points to an array –
a aggregated type data



assignment

q = a

equivalence

***q ≡ *a**

q[0] ≡ a[0]

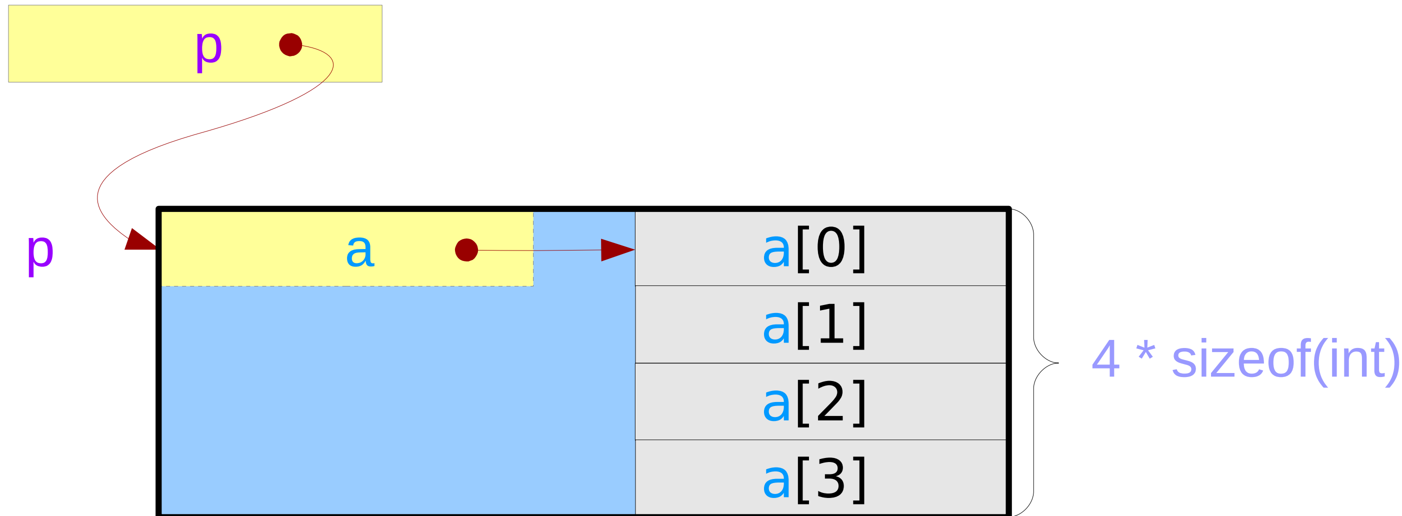
int a[4] ;

q : int (*) = int * type

Pointer to an array – a aggregated type view

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

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

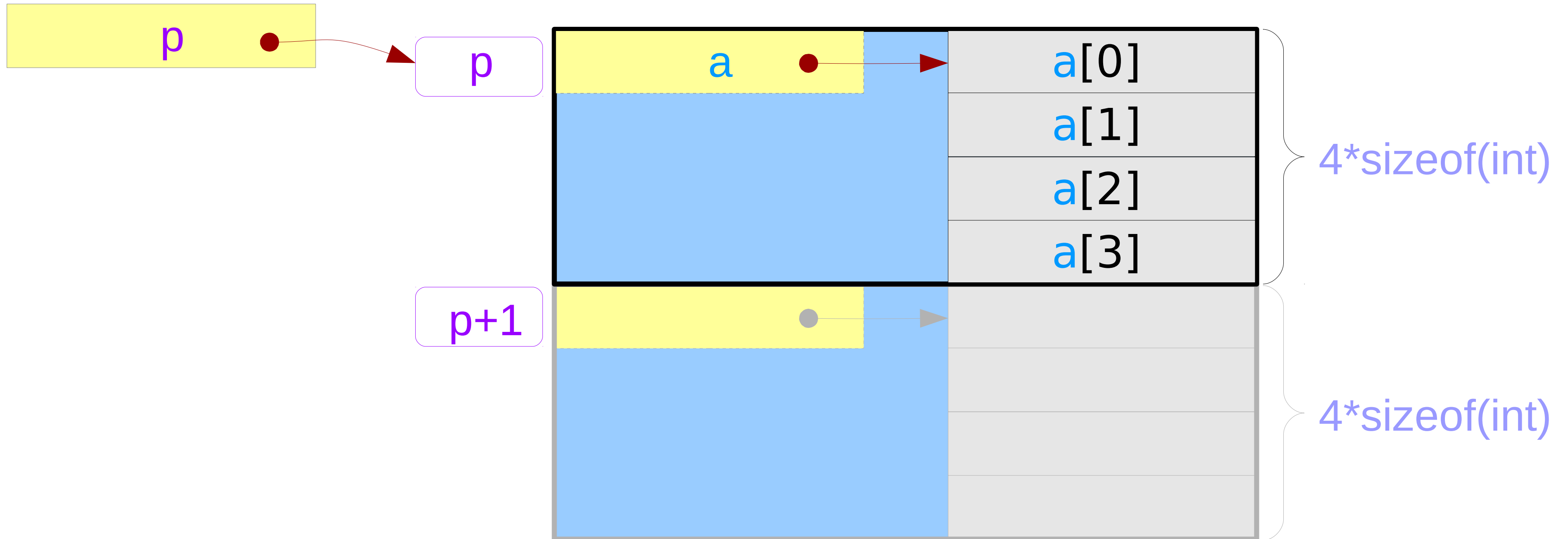


Incrementing a pointer to an array

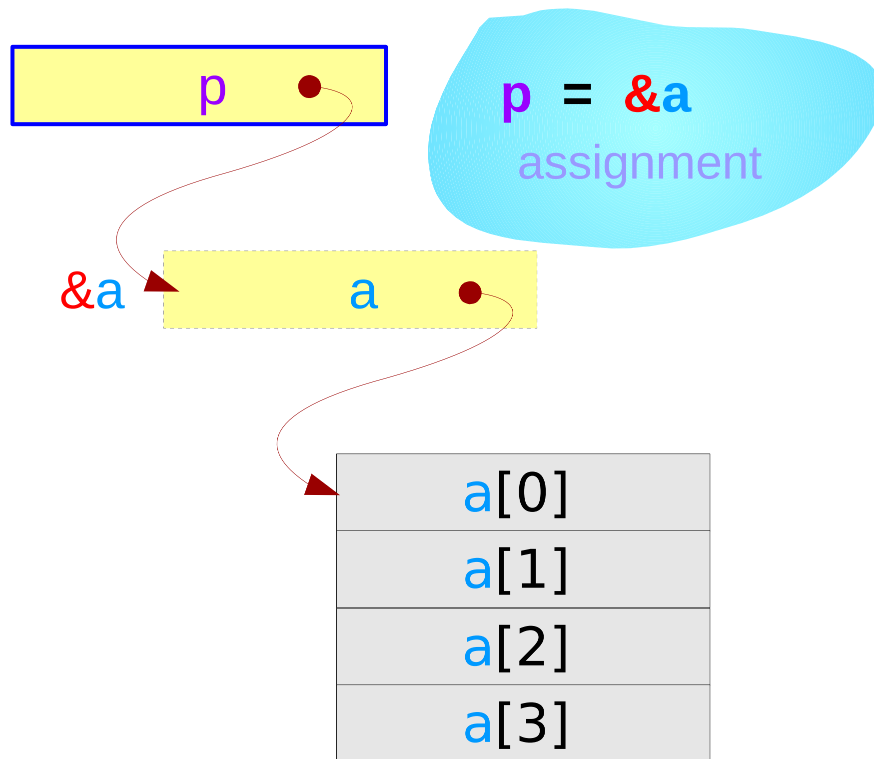
```
int     (*p) [4];
```

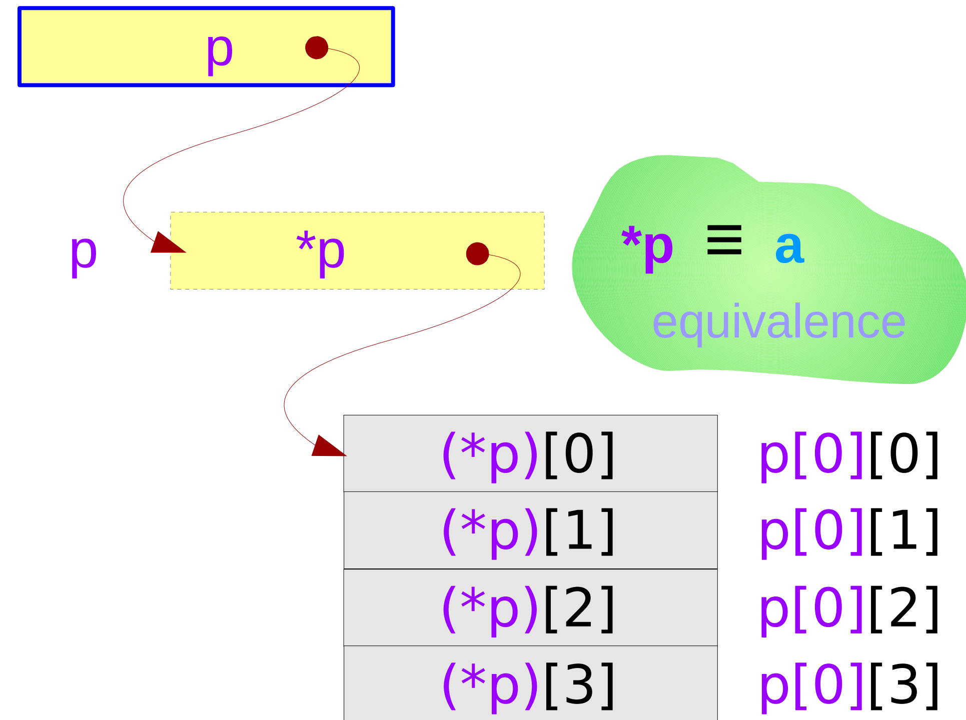
Address value (p+1) – **Address value** (p)
= (long) (p+1) - (long) (p) = 4 * sizeof(int)

Aggregated Type Size



Pointer to an array – a variable view



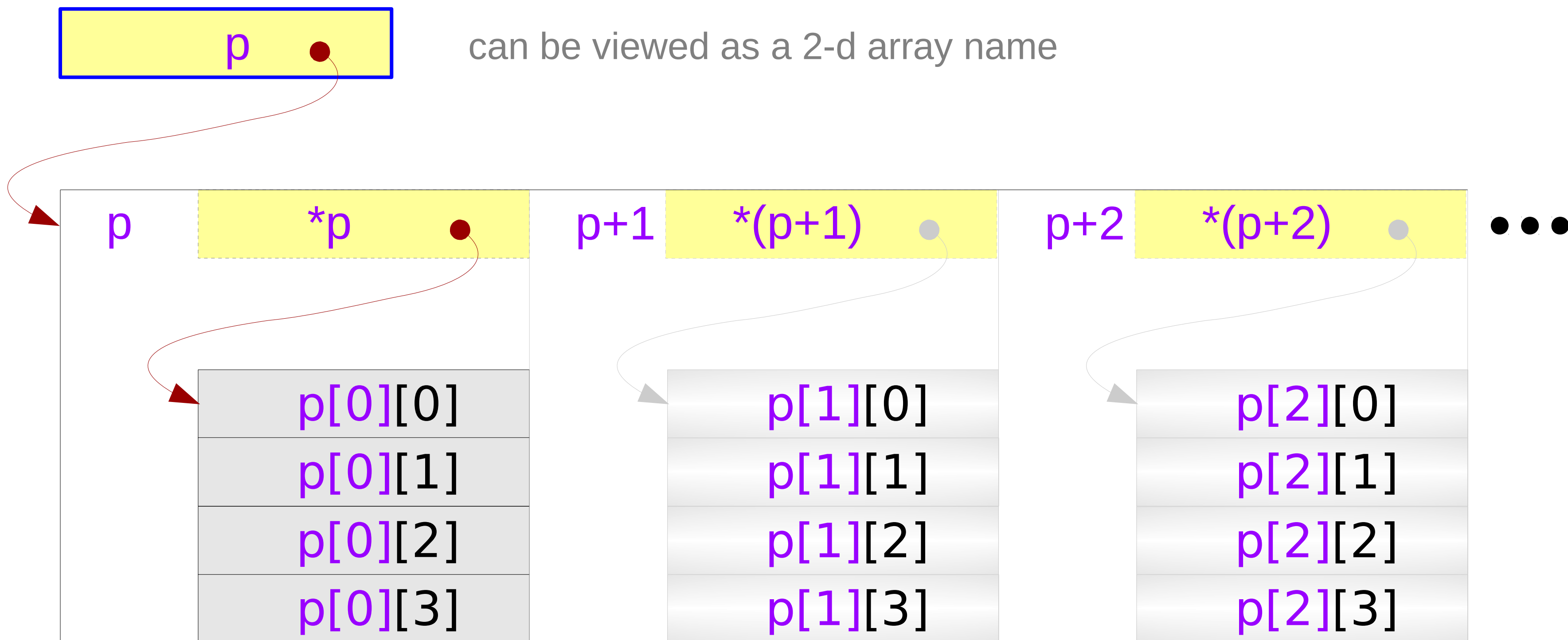


Pointer to an array – an extended variable view

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

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

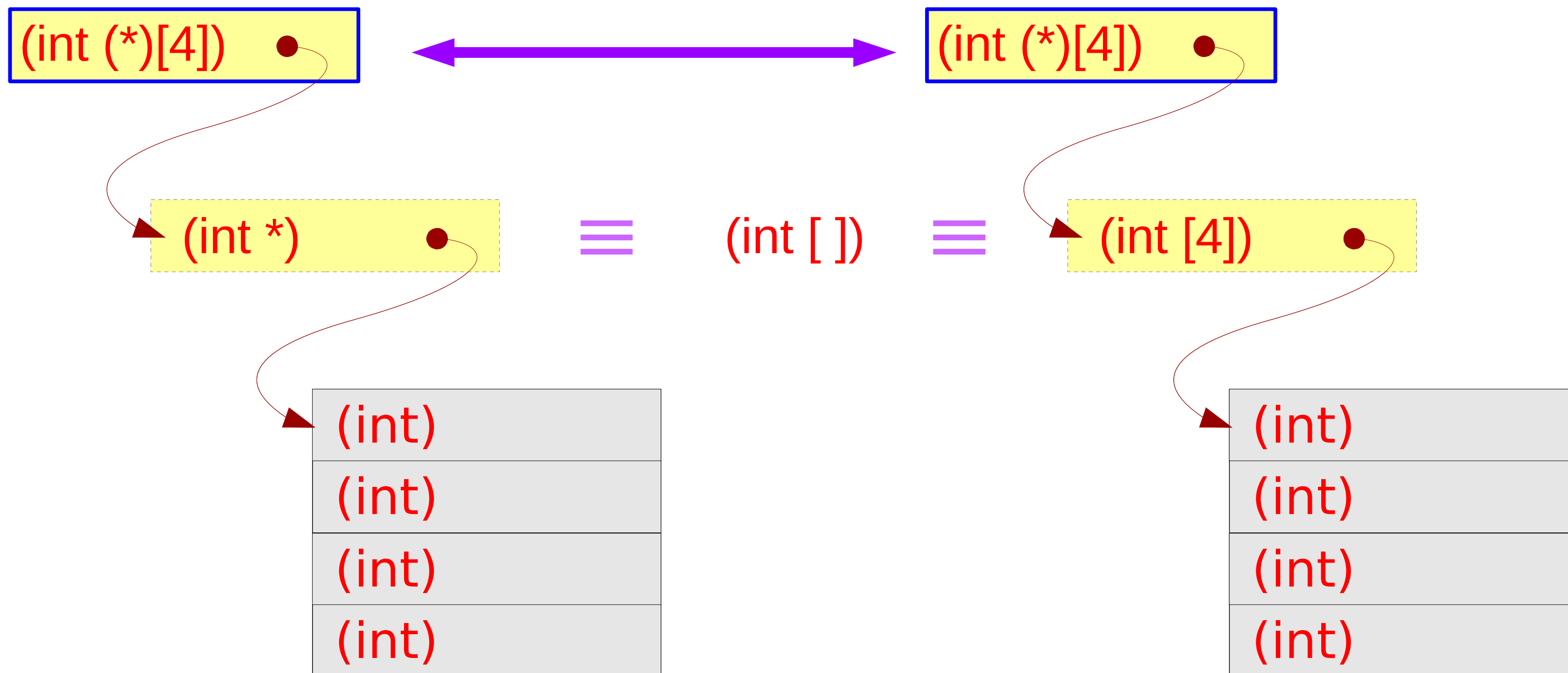
```
p  = &a;
```



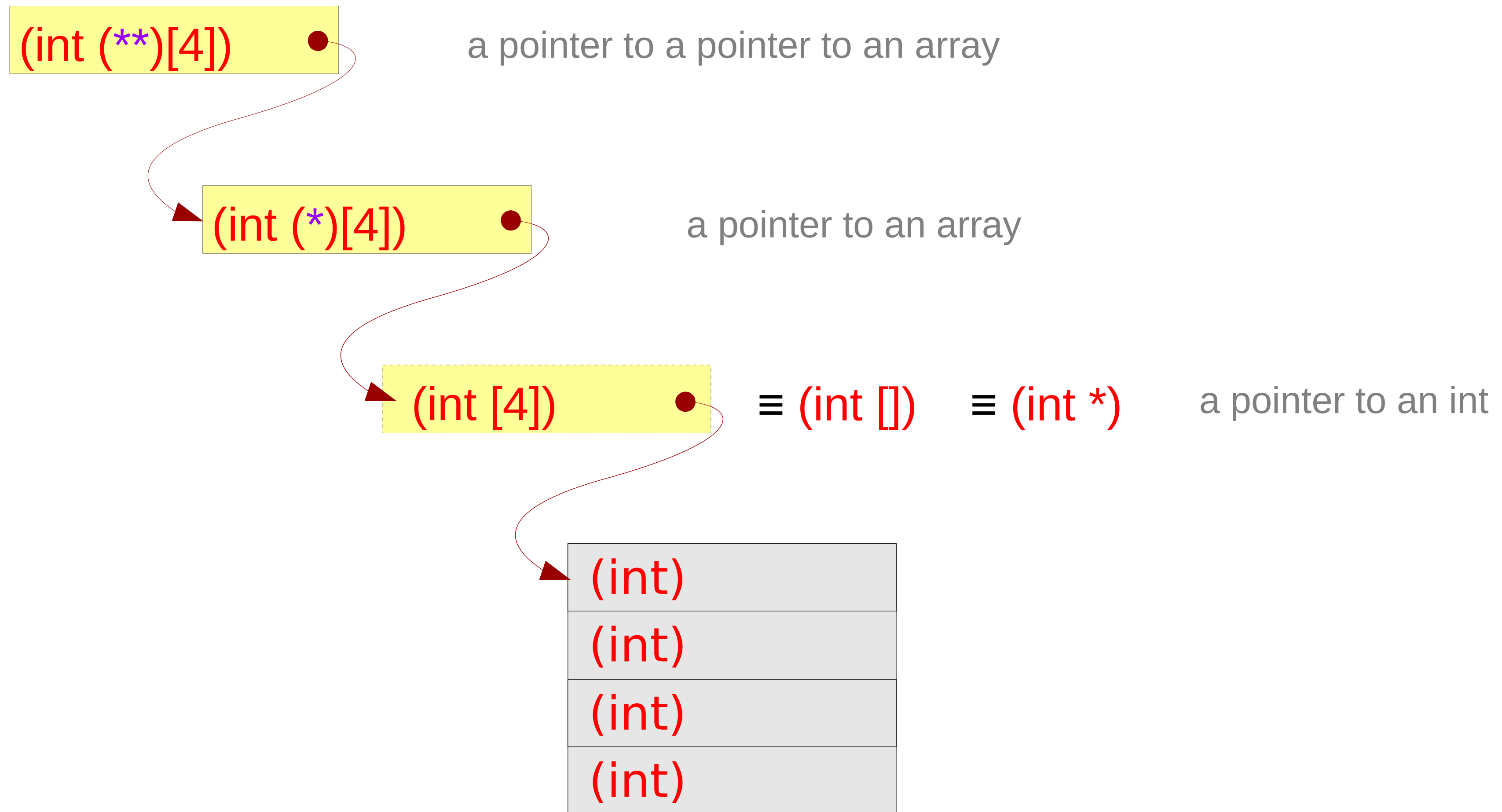
Pointer to an array – a type view

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

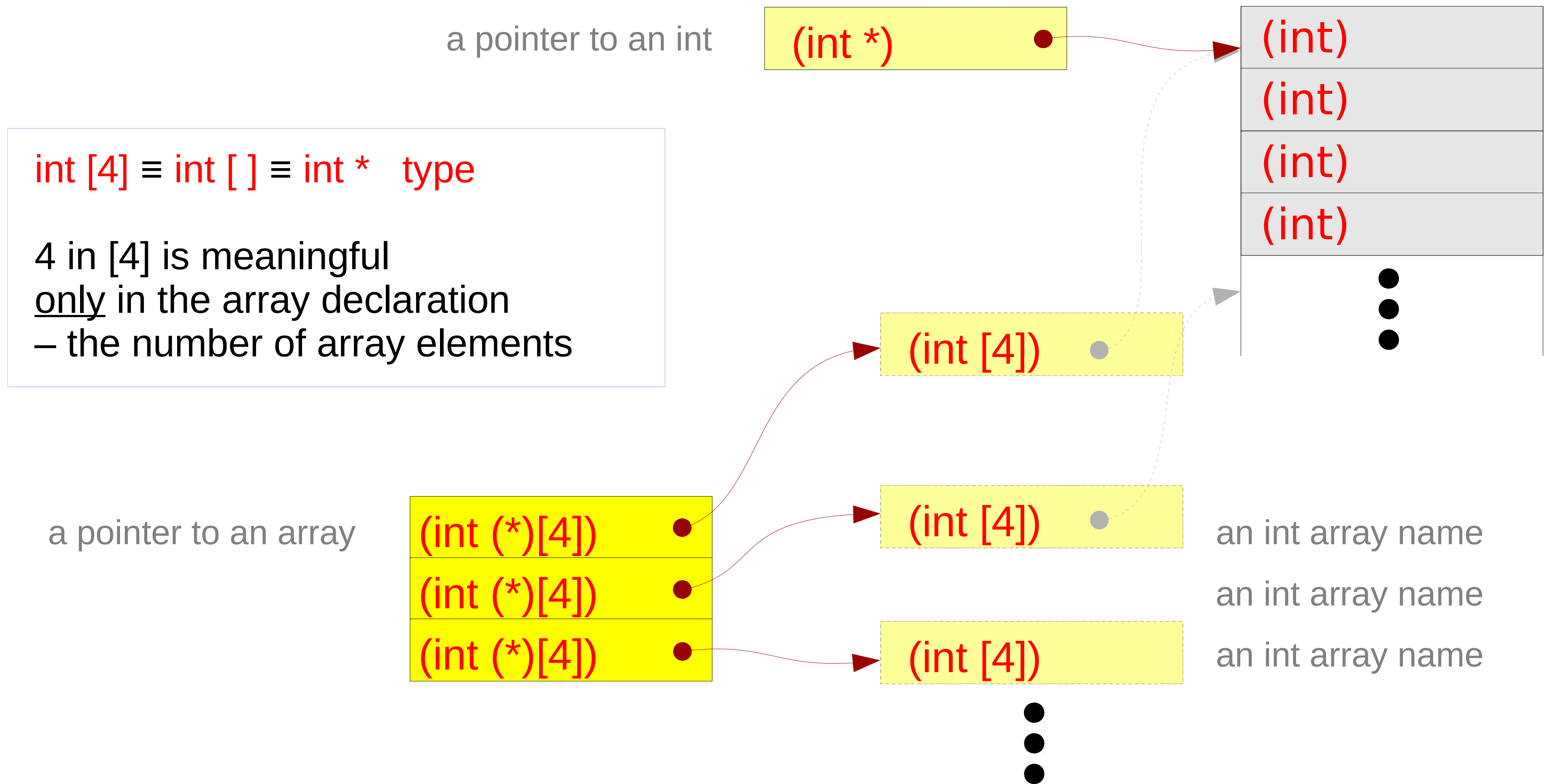
```
int  (*p) [4]  = &a;
```



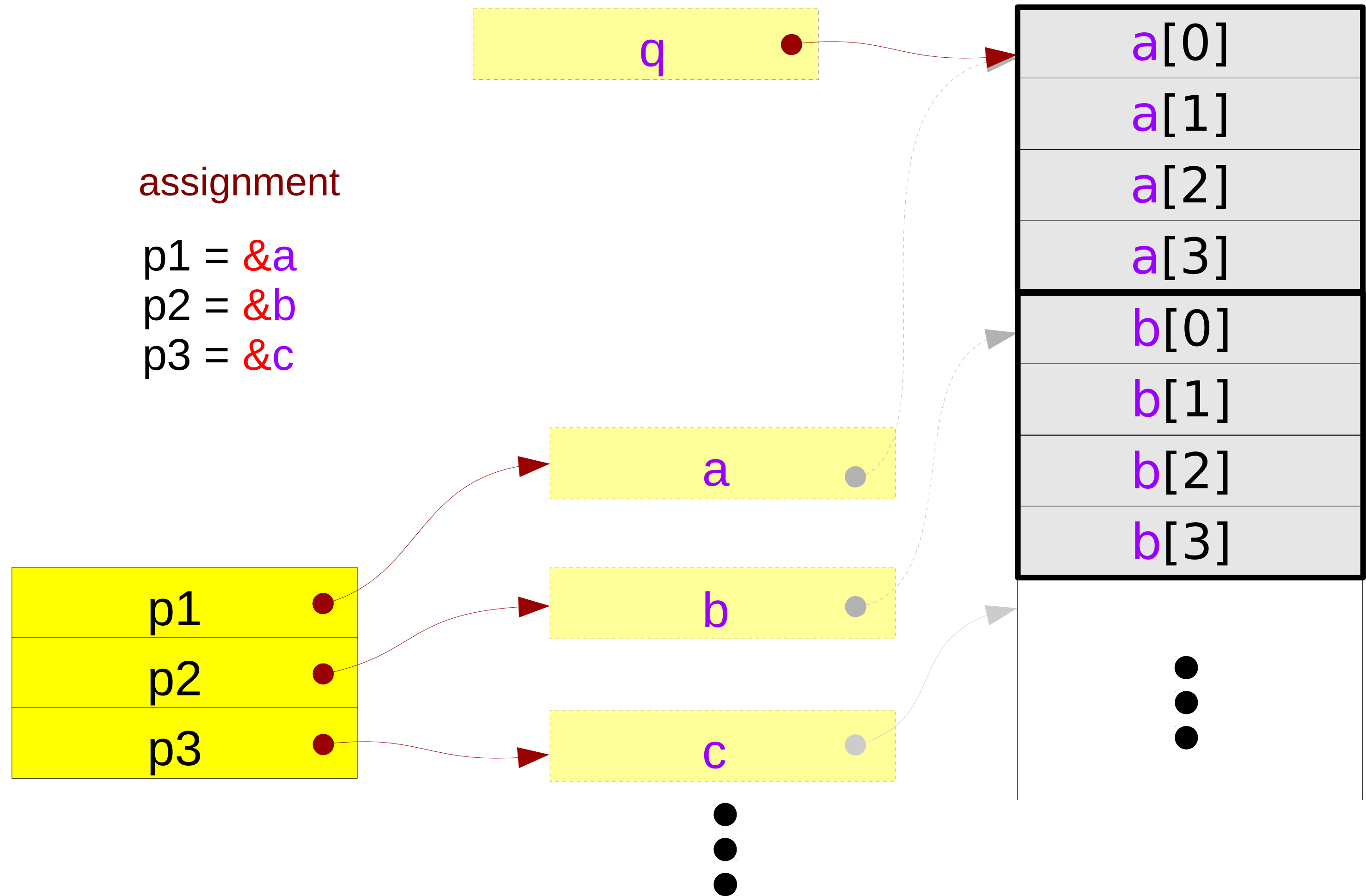
Double pointer to an array – a type view



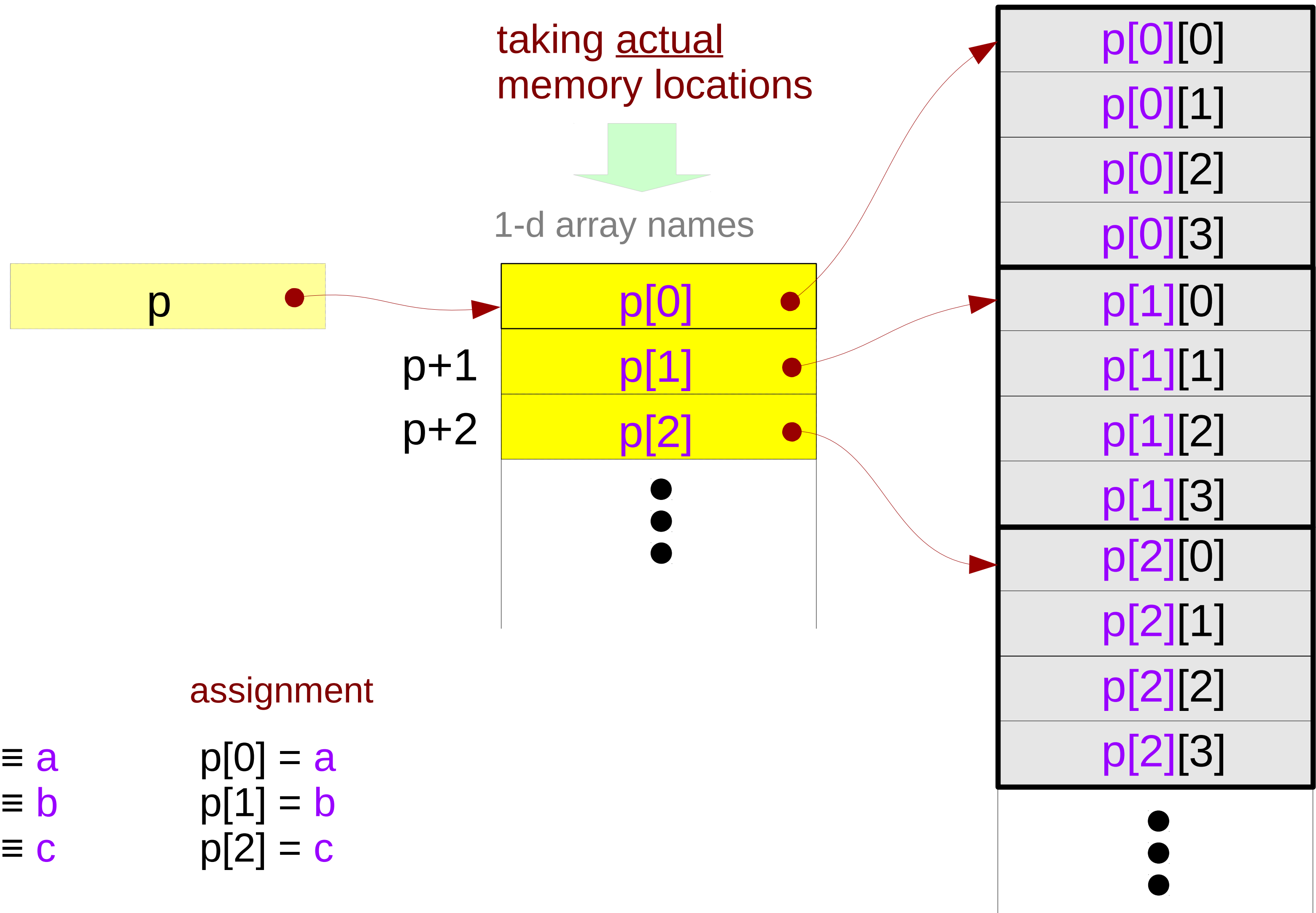
Series of array pointers – a type view



Series of array pointers – a variable view



Pointer array – a variable view



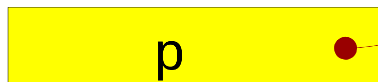
p[0] ≡ *(p+0) ≡ a
p[1] ≡ *(p+1) ≡ b
p[2] ≡ *(p+2) ≡ c

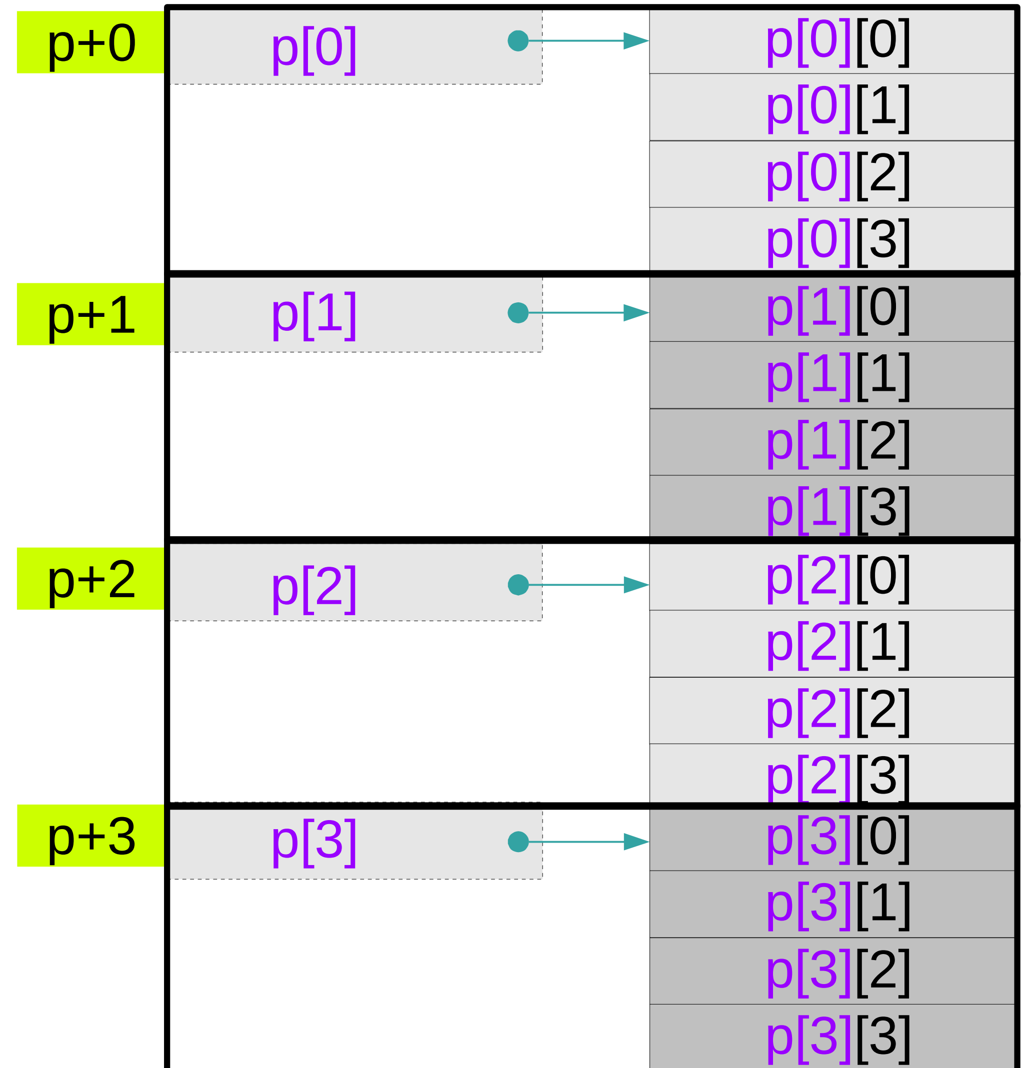
p[0] = a
p[1] = b
p[2] = c

Pointer to consecutive 1-d arrays

int (*p)[4];

a pointer to an array





equivalence

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

assignment

p = &a

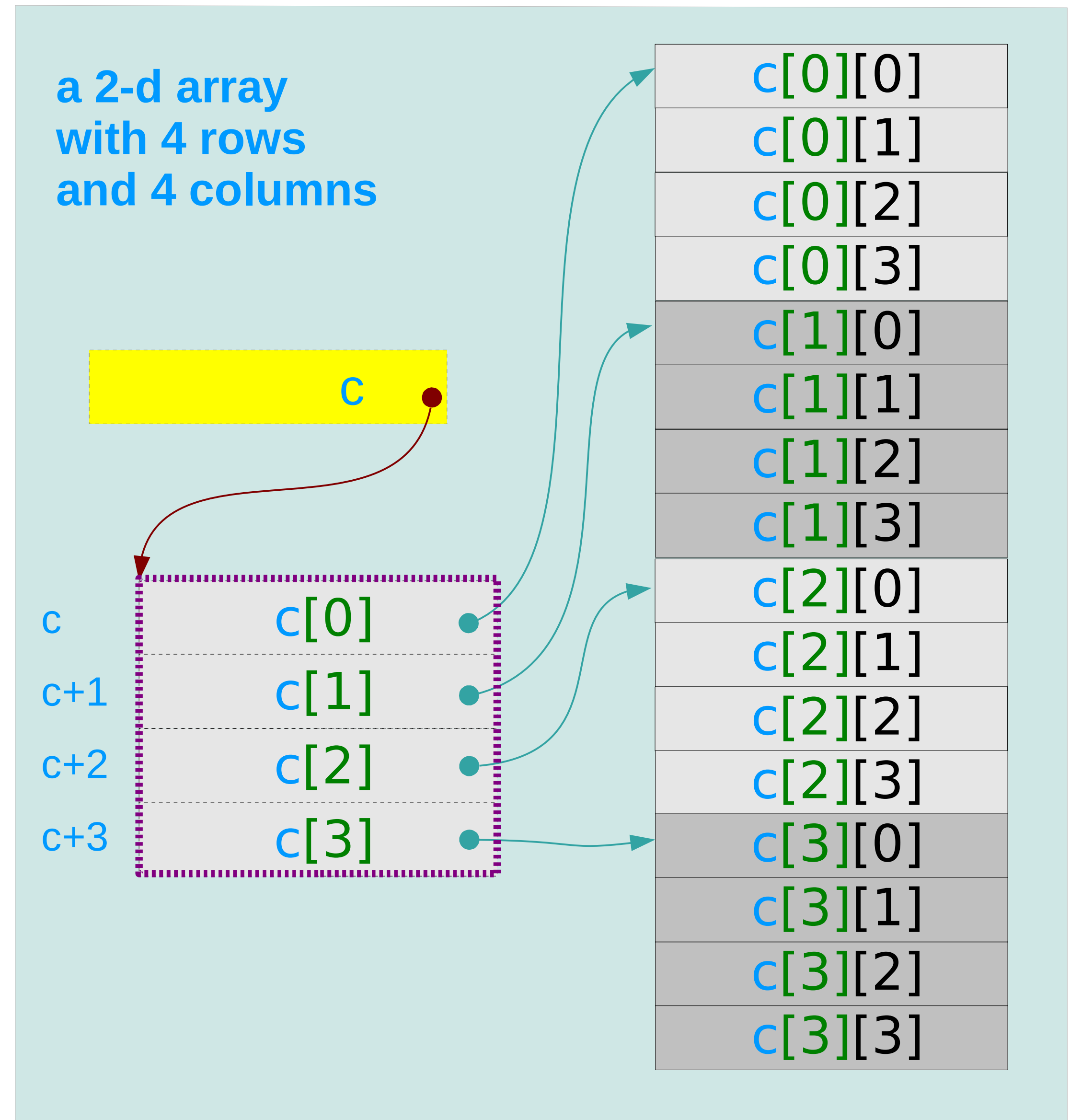
if arrays a, b, c, d
are consecutive

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

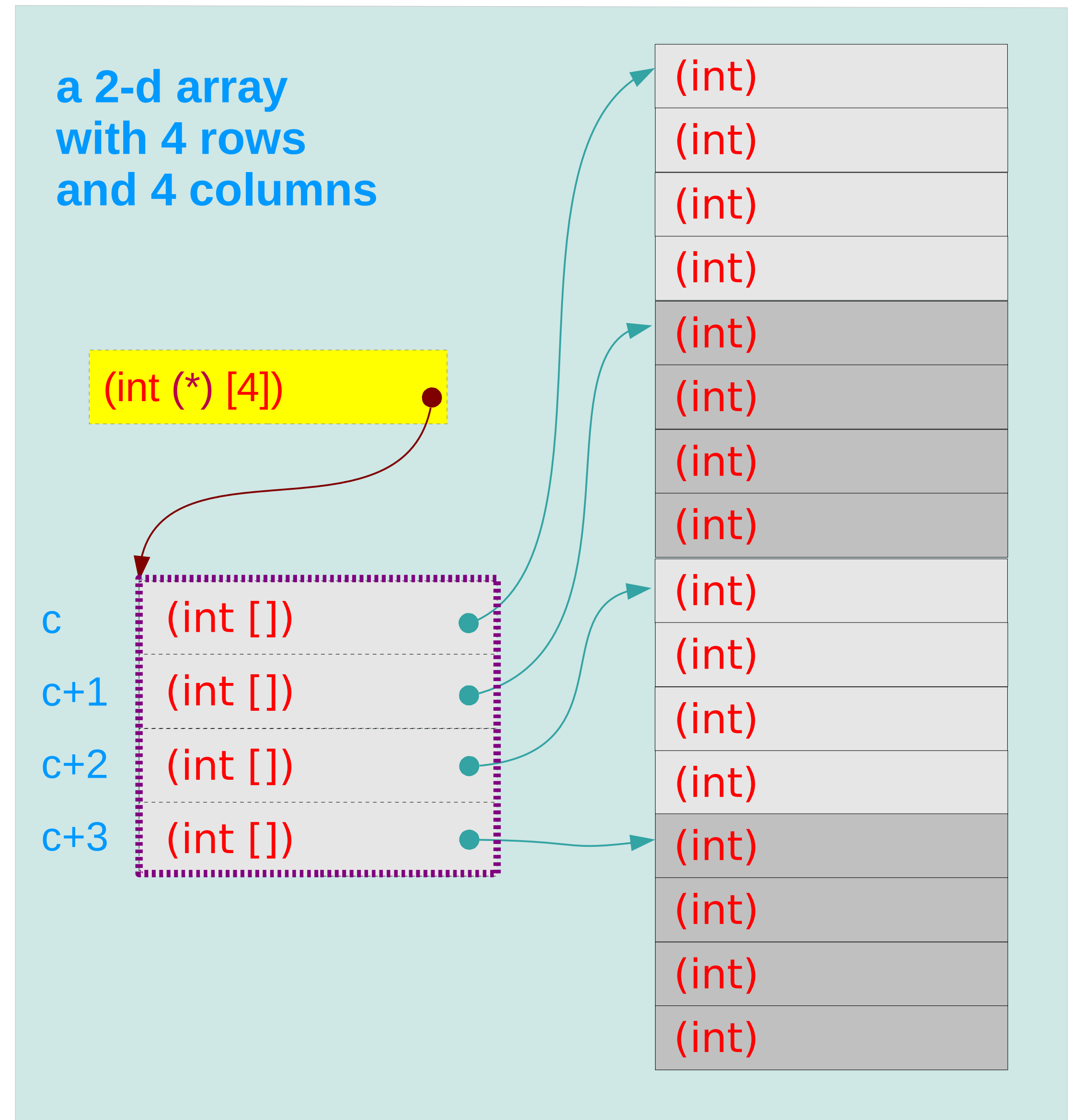
the array name **c** of a 2-d array as an array pointer which points to its 1st 1-d sub-array of 4 elements.

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

c[0], **c[1]**, **c[2]**, **c[3]** can be implemented without taking actual memory locations

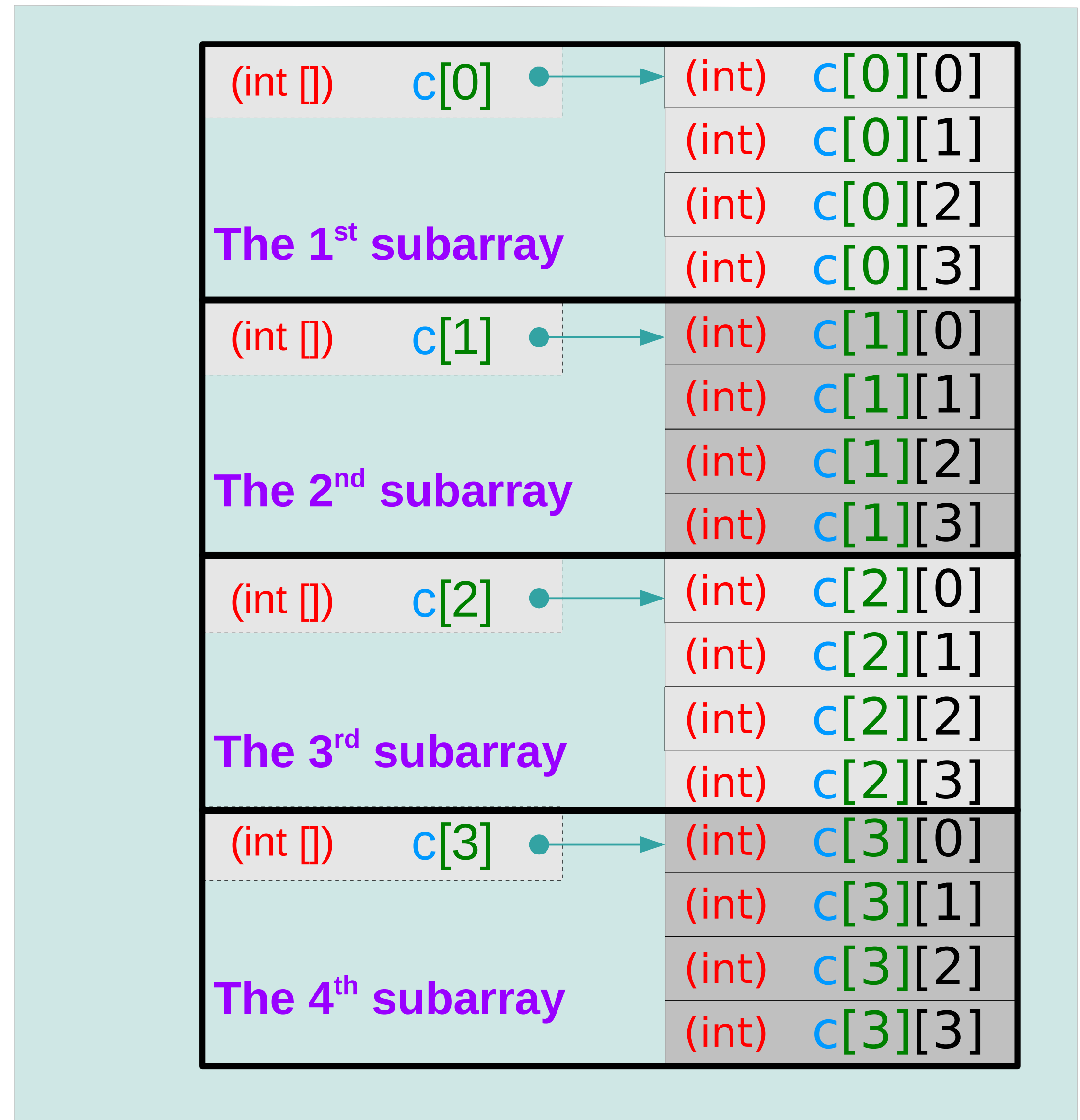


A 2-d array and its sub-arrays – a type view



1-d subarray aggregated data type

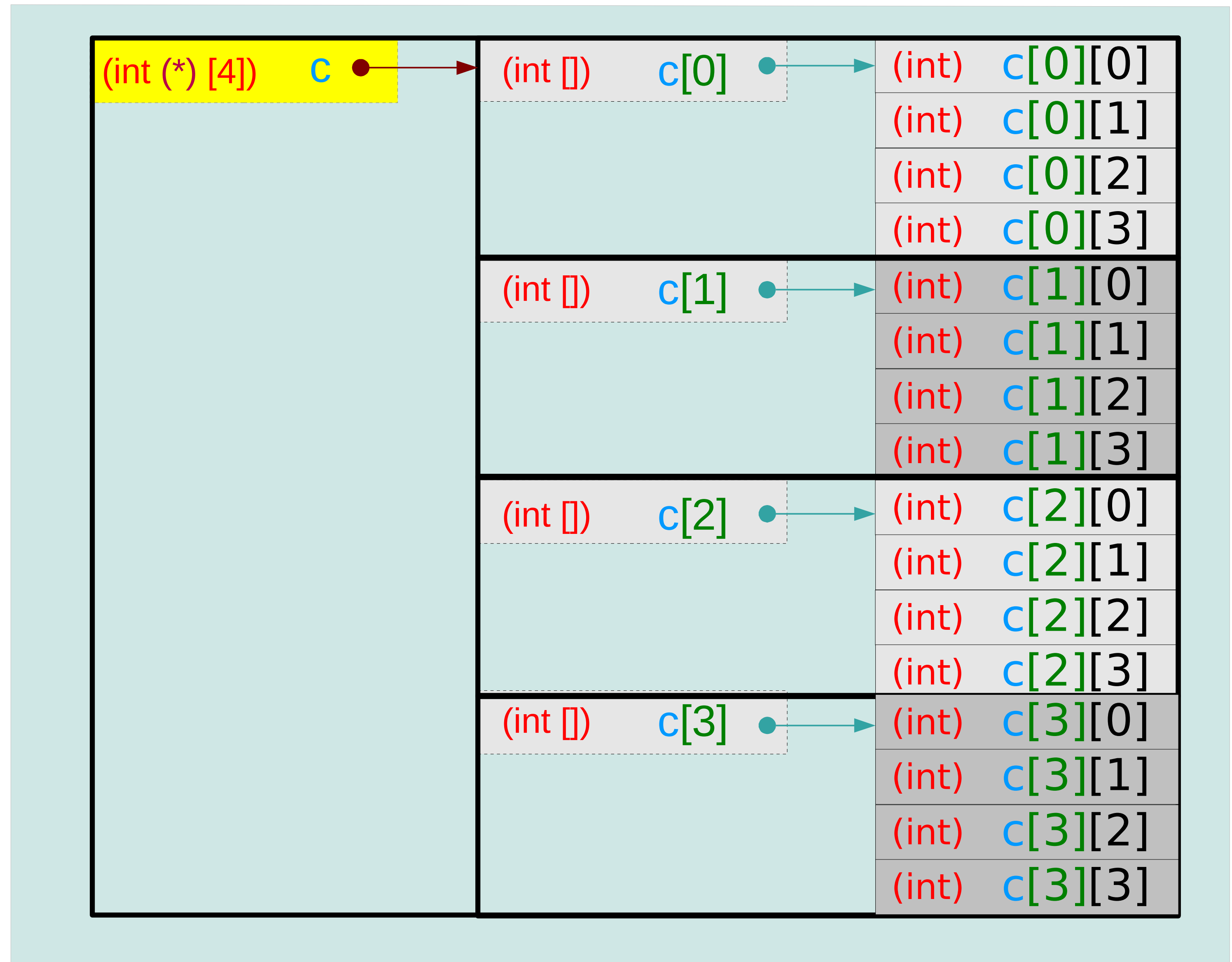
sizeof(c[0]) = 16 bytes
sizeof(c[1]) = 16 bytes
sizeof(c[2]) = 16 bytes
sizeof(c[3]) = 16 bytes



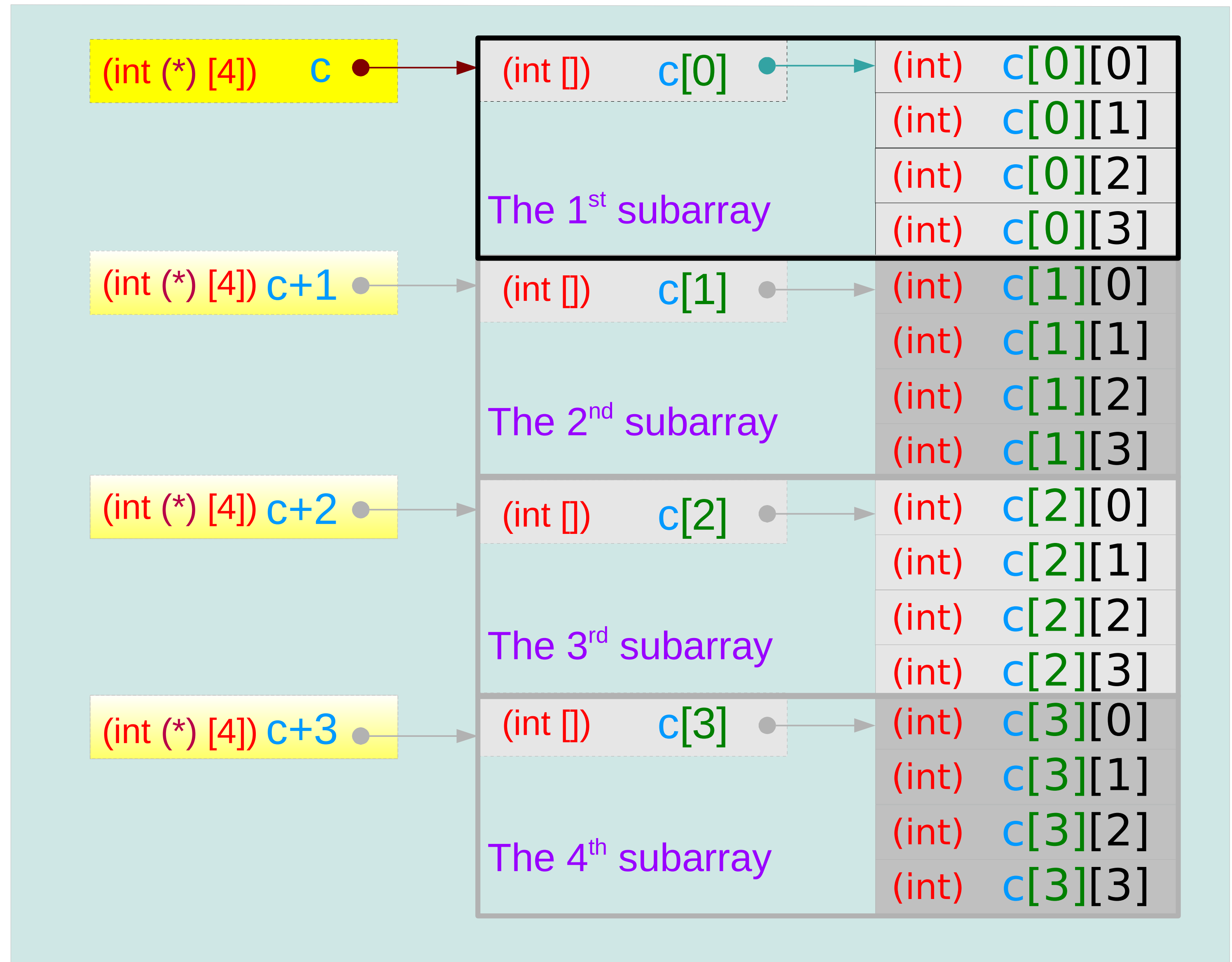
2-d subarray aggregated data type

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

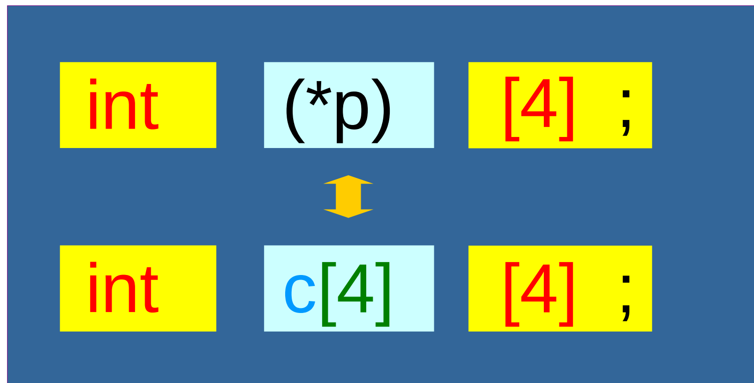
1-d sub-arrays :
sizeof(*c) = 16 bytes

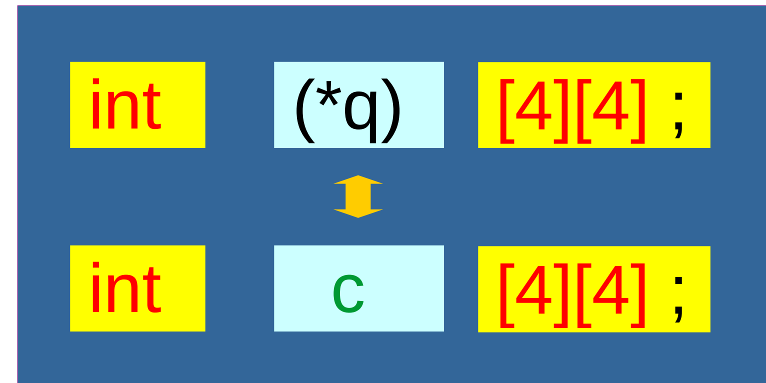


2-d array name as a pointer to a 1-d subarray



Assignment of array pointer variables





(int (*) [4])

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

```
p = c;
```

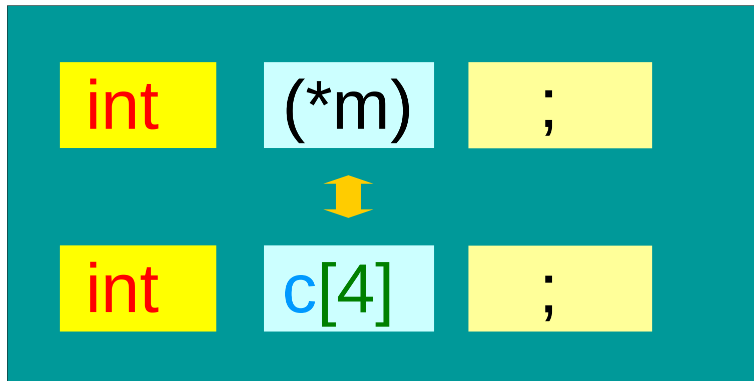
(int(*)[4][4])

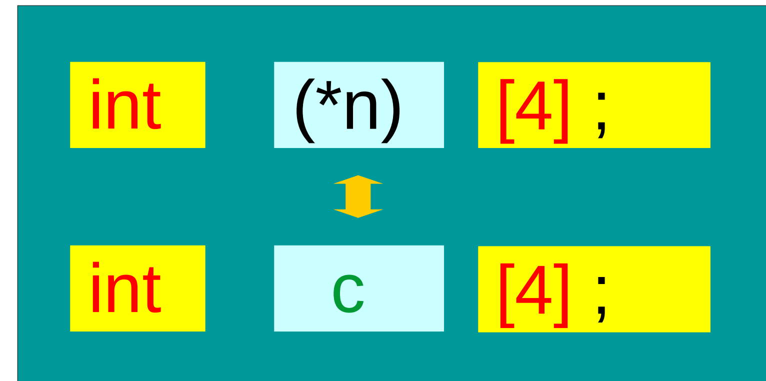
```
q = &c;
```

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

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

Assignment of array pointer variables





(int (*))

m = &c[0];

m = c;

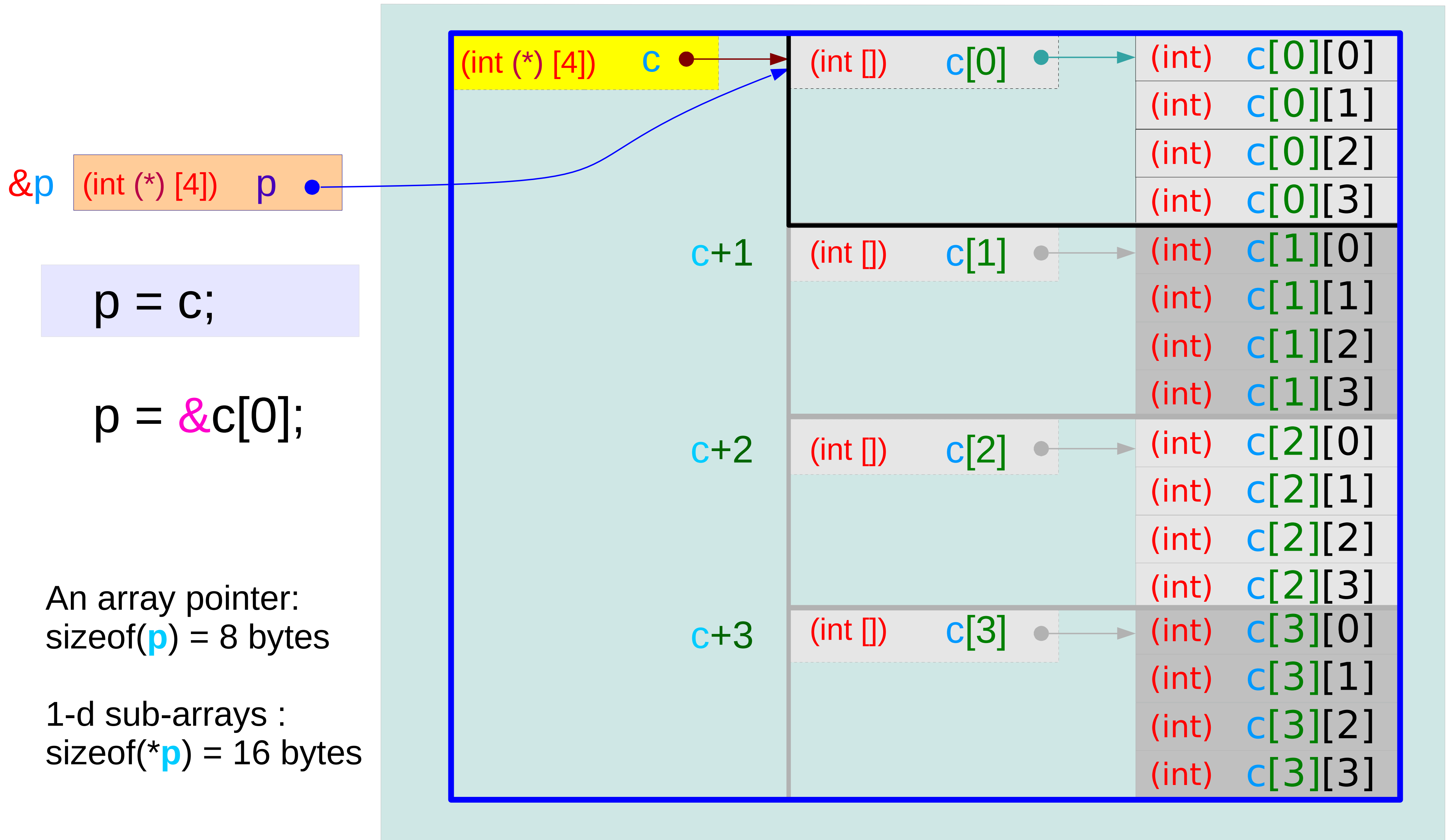
(int(*)[4])

n = &c;

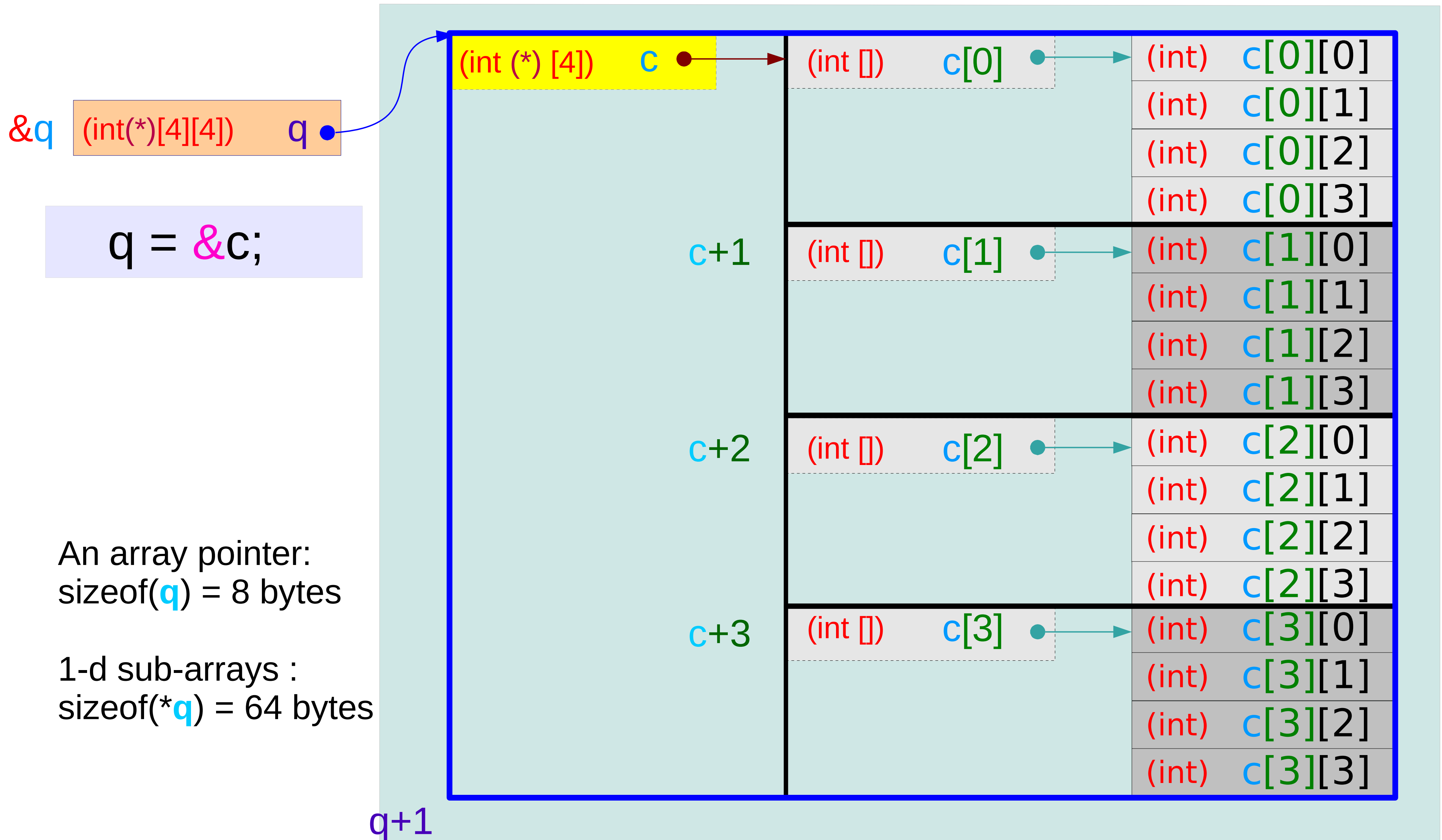
m[0] ≡ c[0]
m[1] ≡ c[1]
m[2] ≡ c[2]
m[3] ≡ c[3]

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

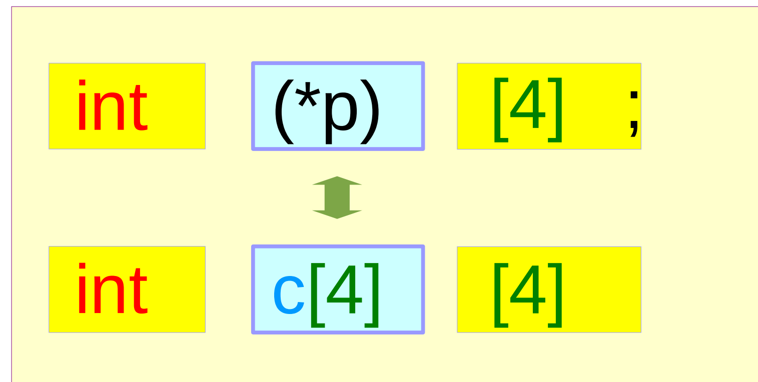
Pointer variable to a 1-d array

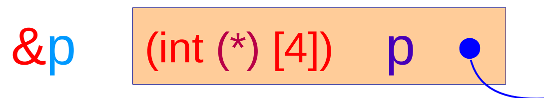


Pointer variable to a 2-d array



Using a a pointer to a 1-d array





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

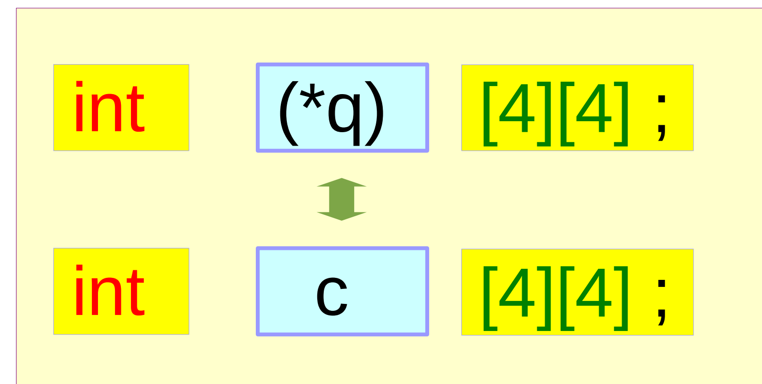
(int (*) [4]) c

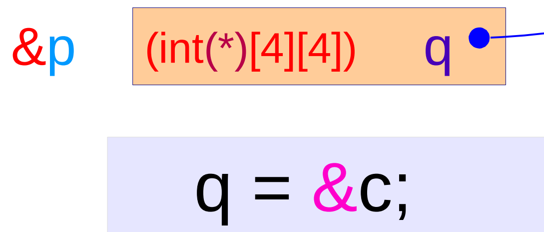
p

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 pointer to a 2-d array





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]

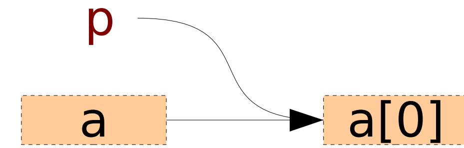
Pointer to multi-dimensional arrays (1)

Declaration	Description
int a[4] ; int (*p) ;	A pointer to a 0-d array (an integer) can be viewed as a 1-d array name
int b[4] [2]; int (*q) [2];	A pointer to a 1-d array can be viewed as a 2-d array name
int c[4] [2][3]; int (*r) [2][3];	A pointer to a 2-d array can be viewed as a 3-d array name
int d[4] [2][3][4]; int (*s) [2][3][4];	A pointer to a 3-d array can be viewed as a 4-d array name

Pointer to multi-dimensional arrays (2)

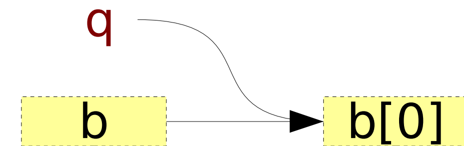
```
int a[4] ;
int (*p) ;
```

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



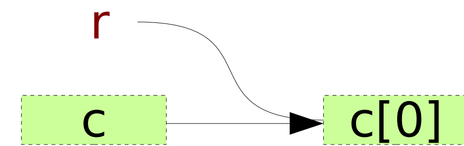

```
int b[4] [2];
int (*q) [2];
```

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



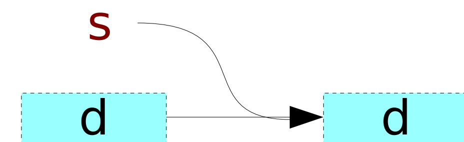

```
int c[4] [2][3];
int (*r) [2][3];
```

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

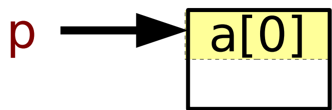


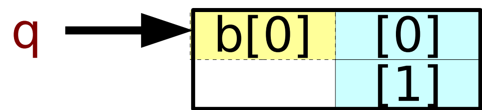

```
int d[4] [2][3][4];
int (*s)  [2][3][4];
```

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



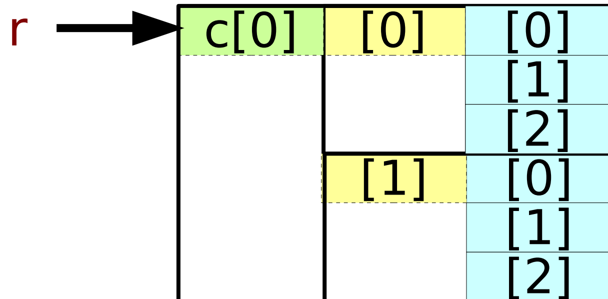
Pointer to multi-dimensional arrays (3)





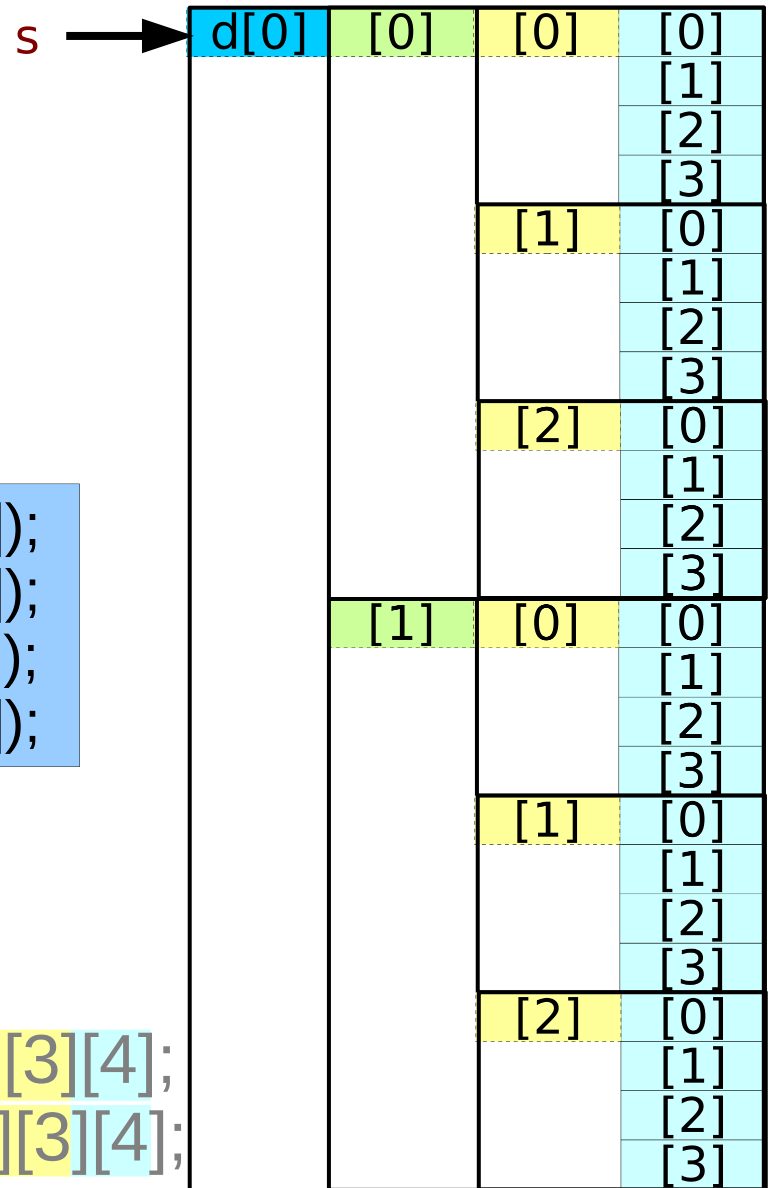
int a[4] ;
int (*p) ;

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



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

p = a; (=&a[0]);
q = b; (=&b[0]);
r = c; (=&c[0]);
s = d; (=&d[0]);



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

To pass array name

```
int a[4] ;           prototype  void func(int (*p), ...);
int (*p) ;           call       func(a, ...);

int b[4] [2];        prototype  void func(int (*q)[2], ...);
int (*q) [2];        call       func(b, ...);

int c[4] [2][3];     prototype  void func(int (*r)[2][3], ...);
int (*r)  [2][3];    call       func(c, ...);

int d[4] [2][3][4];  prototype  void func(int (*s)[2][3][4], ...);
int (*s) [2][3][4];  call       func(d, ...);
```

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