Pointers (1A)

Young Won Lim
9/23/23

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

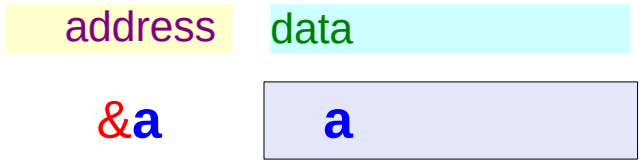
This document was produced by using LibreOffice.

Pointers

Variables

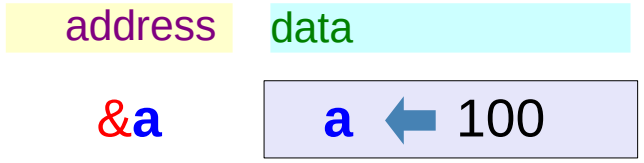
```
int  a ;
```

a can hold an integer value




```
a = 100 ;
```

a holds the integer 100



Pointer Variables

```
int * p ;
```

p can hold an address of
an integer data

type *variable*

```
int * p ;
```

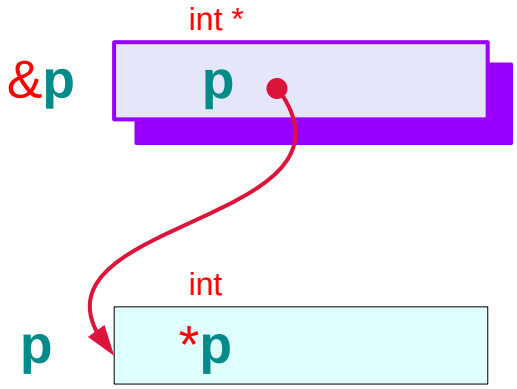
p holds the address

pointer to int

```
int * p ;
```

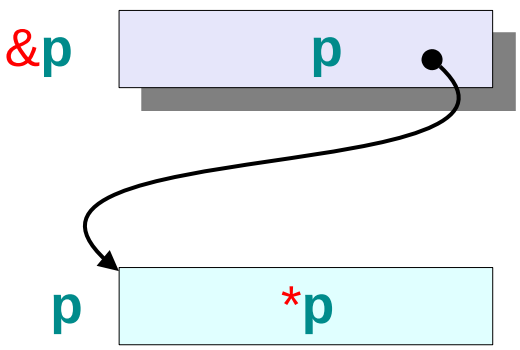
***p** holds an integer value

int



Pointer variable **p** and dereferenced variable ***p**

*&**p** : the address of pointer variable **p***



*pointer **p** points here, thus the address is **p***

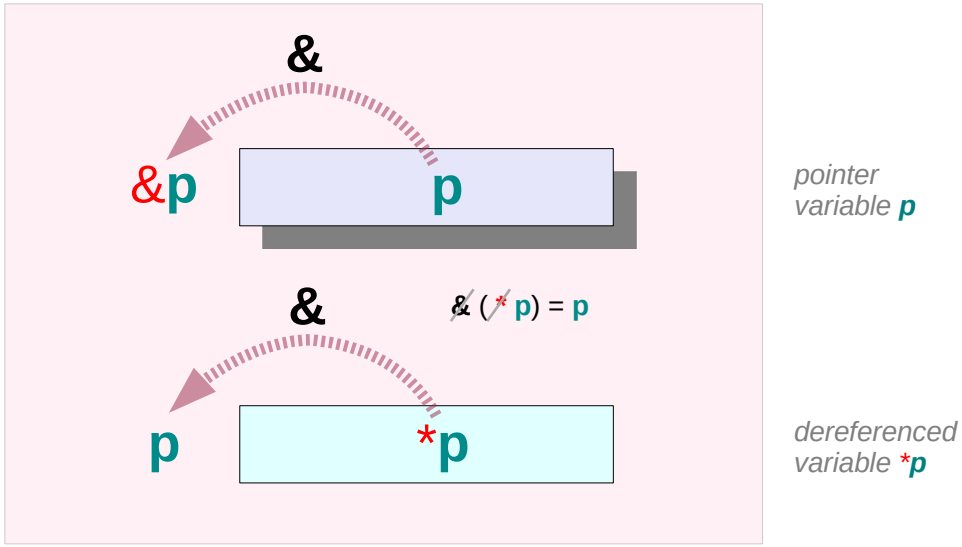
*pointer variable **p***

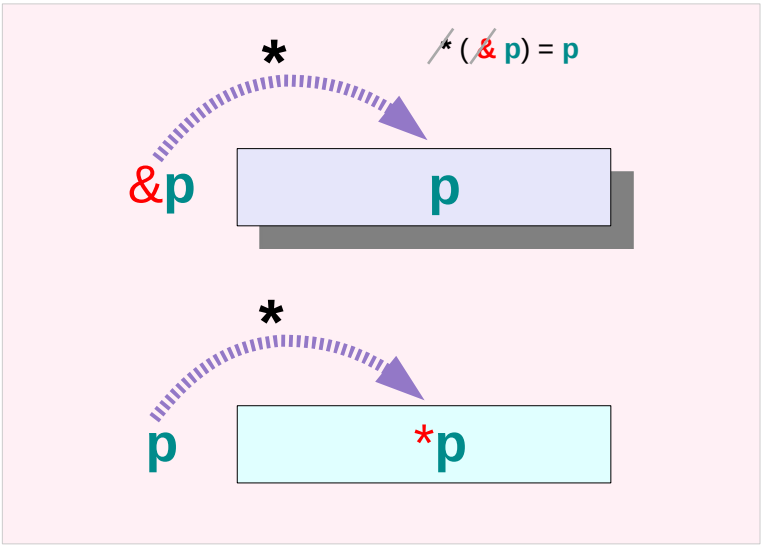
*dereferenced variable ***p***

Address-of operator and dereferencing operator (1)

the address of a variable :
address-of operator ***&***

the content of an address :
dereferencing operator *******

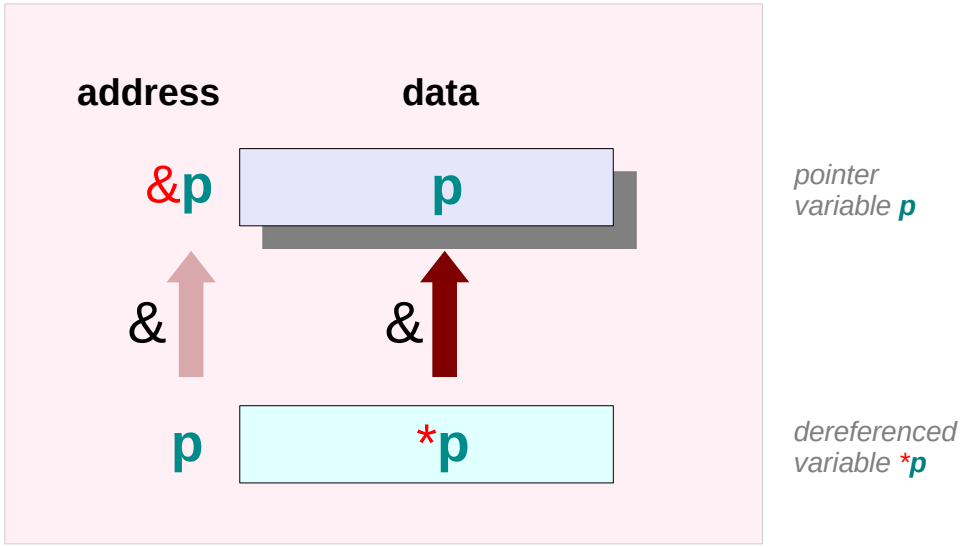


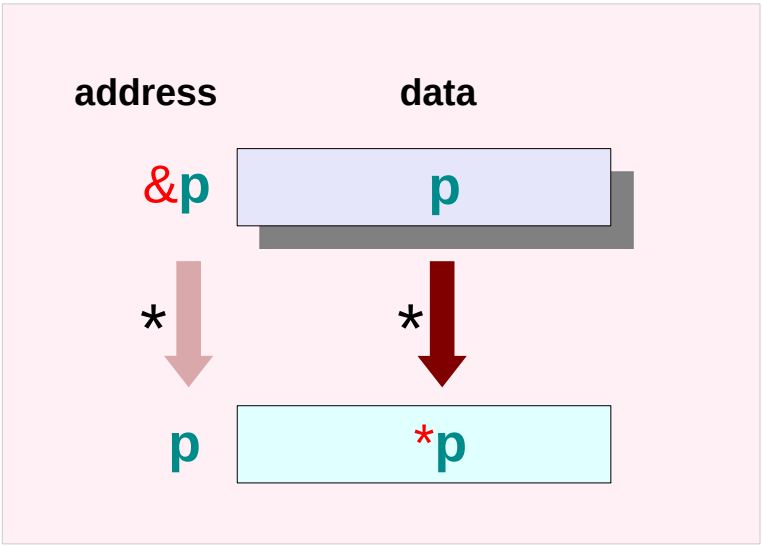


Address-of operator and dereferencing operator (2)

the address of a variable :
address-of operator ***&***

the content of an address :
dereferencing operator *******





Address-of operator and dereferencing operator (1)

the address of a variable :
address-of operator ***&***

& variable *:*
returns the address of a variable

variable *must be an lvalue*

variable *has memory locations*
whose value can be changed
by an assignment

the content of an address :
dereferencing operator *******

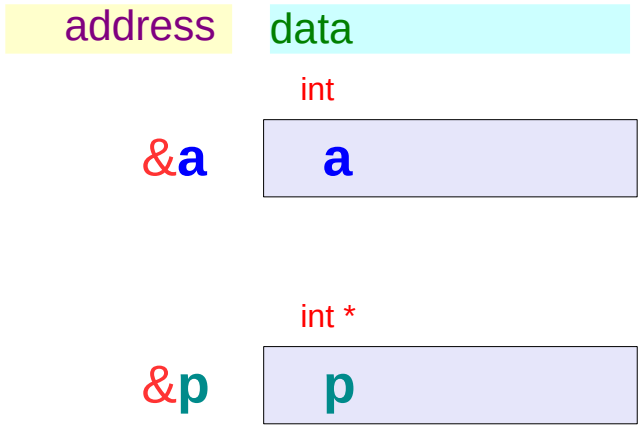
**** address*** *:*
returns the value at the address

**** address*** *is an lvalue*

**** address*** *has memory locations*
whose value can be changed
by an assignment

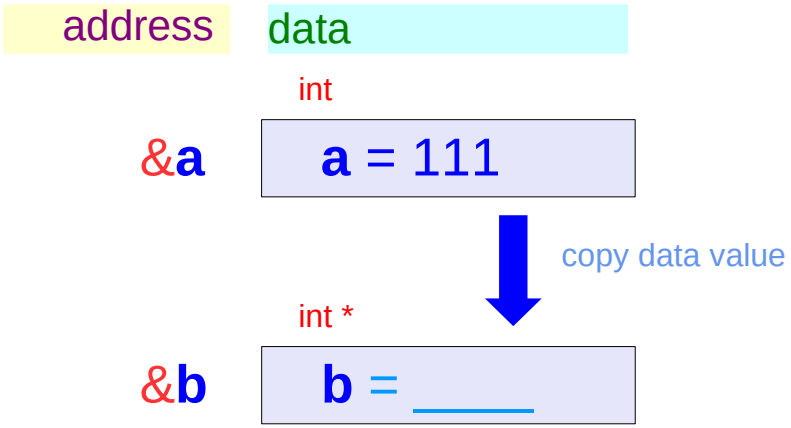
Variables and their addresses

```
int     a ;

int   * p ;
```



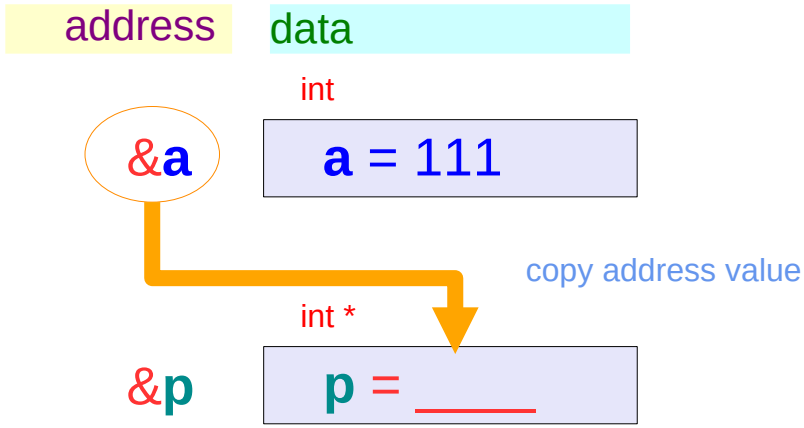
Assignment of a value

```
int     a ;

int     b ;

        b = a ;
```



Assignment of an address

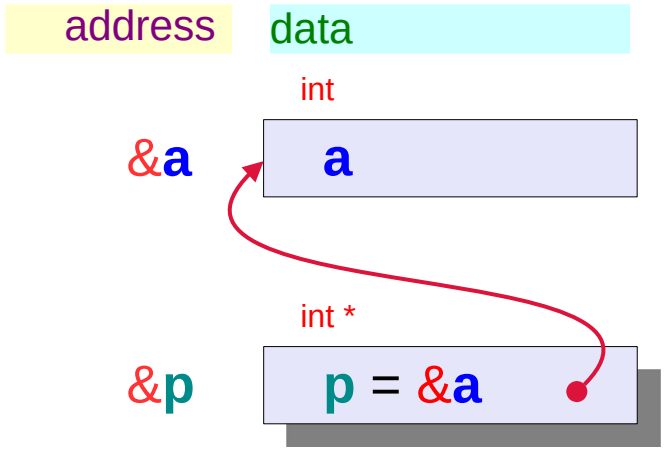
```
int     a ;

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



Variables with initializations

```
int     a ;

int  * p = &a;
```

p = &a;



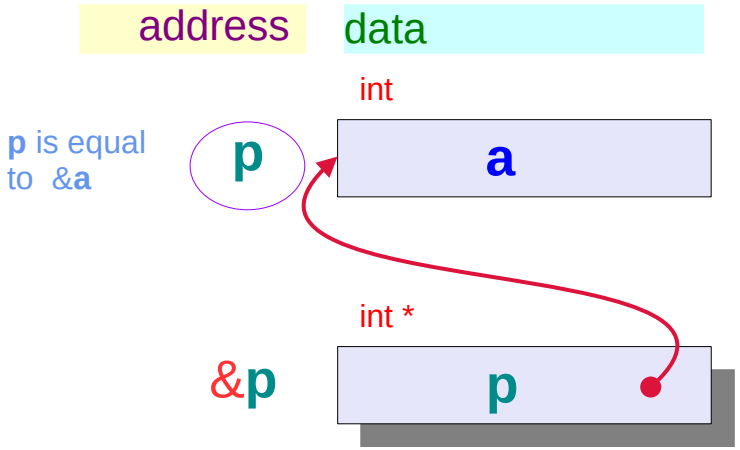
now **p** has the address of **a**

p points to where **a** is

Reference address : **p**

```
int     a;

int  *  p = &a;
```

p = &a;

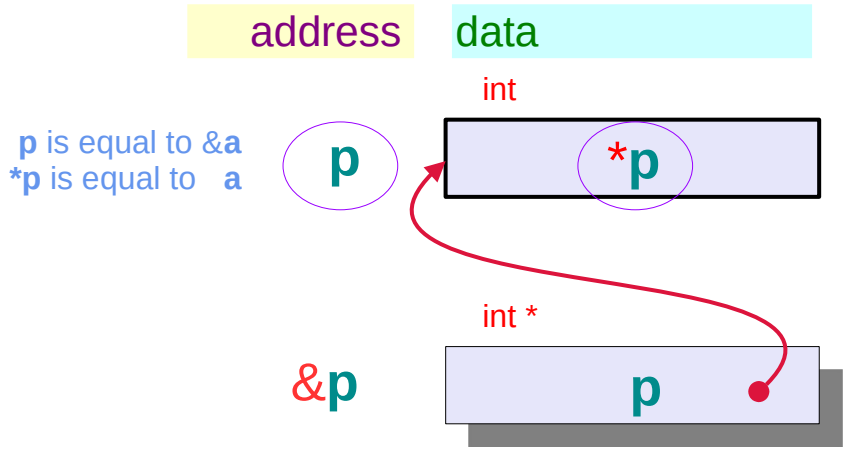


p ≡ &**a**

Dereferenced variable : *p

```
int     a;

int  *  p = &a;
```

p = &a;



equivalence

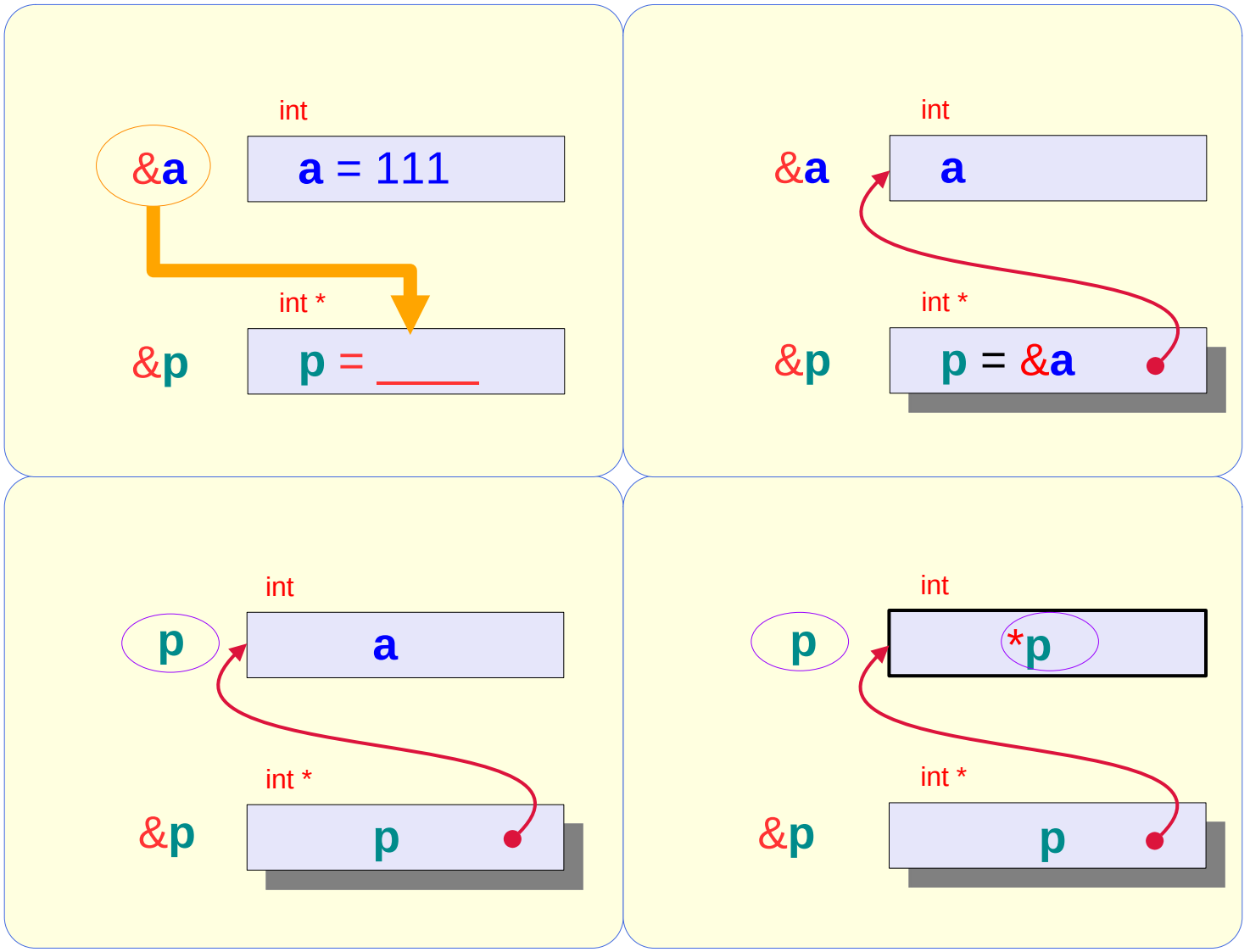
p ≡ &a

*p ≡ *&a

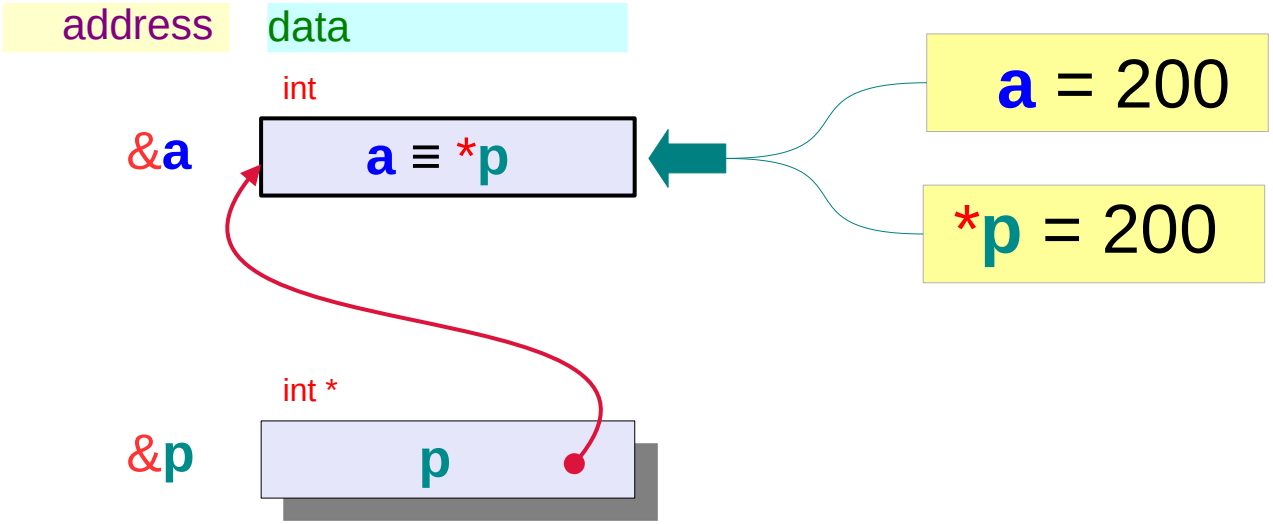
*p ≡ a

Variable **p**, ***p**

```
int   a ;

int  * p ;

p = &a;
```



Two way to access: **a** and ***p**



1) Read / Write **a**

2) Read / Write ***p**

Double Pointers

Double Pointer Variable Definition

```
int  * *    q;
```

q holds an *address*

```
int  * *    q ;
```

a pointer to
an integer pointer

q holds an ***address*** of
an integer pointer type data

```
int  *    *q ;
```

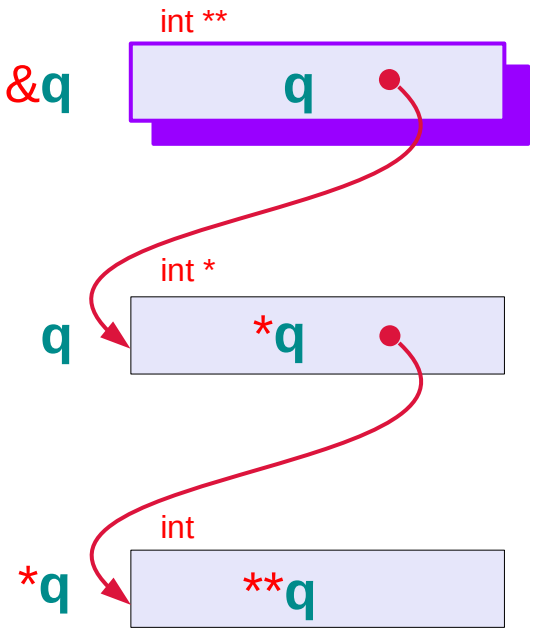
a pointer to
an integer

q** holds an ***address of
an integer type data

```
int    * *q ;
```

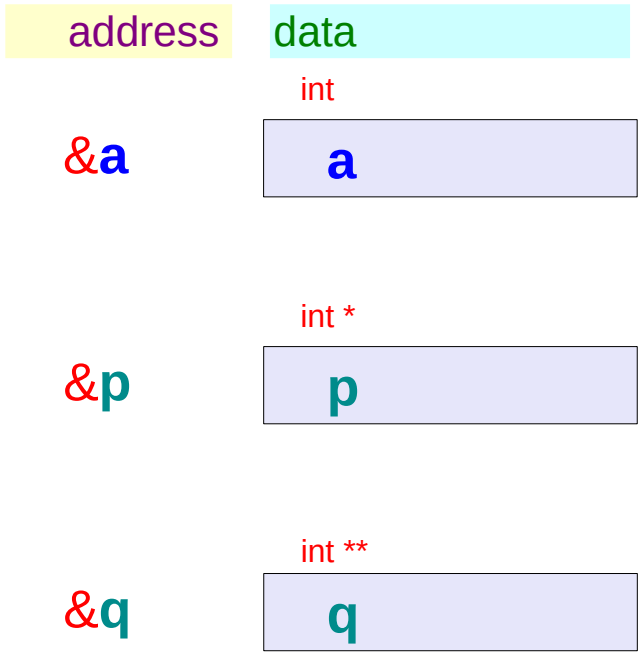
an integer

****q** holds an integer type ***data***



Variables and their addresses

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

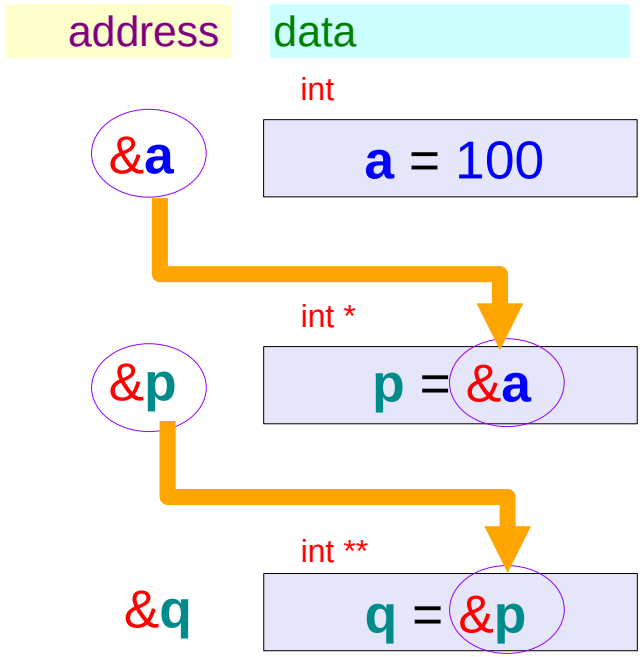


Initialization of Variables

```
int     a = 100 ;
int   * p = &a ;
int   ** q = &p ;
```

p = &a;

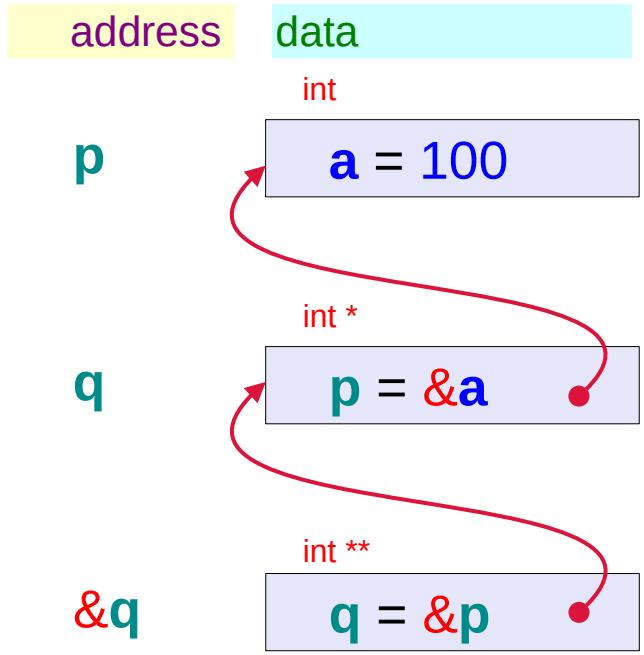
q = &p;



Pointed addresses : **p**, **q**

```
int     a = 100 ;
int   * p = &a ;
int   ** q = &p ;
```

```
p = &a;
q = &p;
```

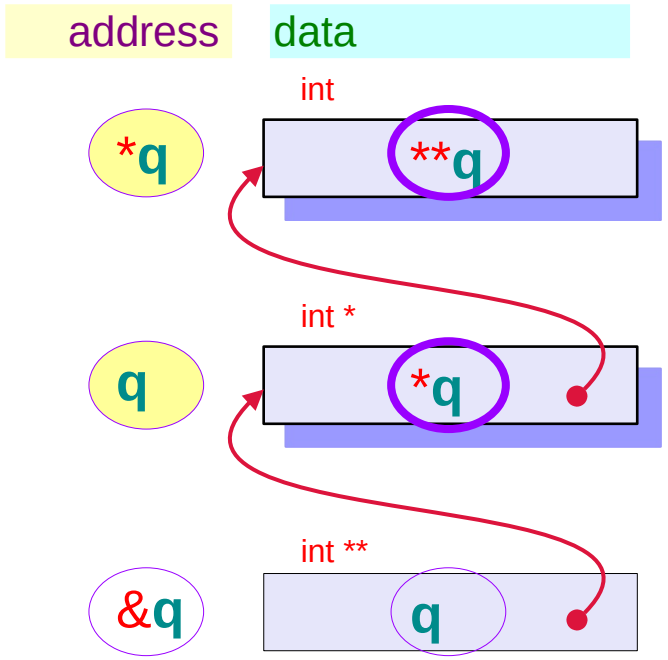


Dereferenced variables : *q, **q

```
int     a = 100 ;
int   * p = &a ;
int   ** q = &p ;
```

**q ≡ a

*q ≡ p

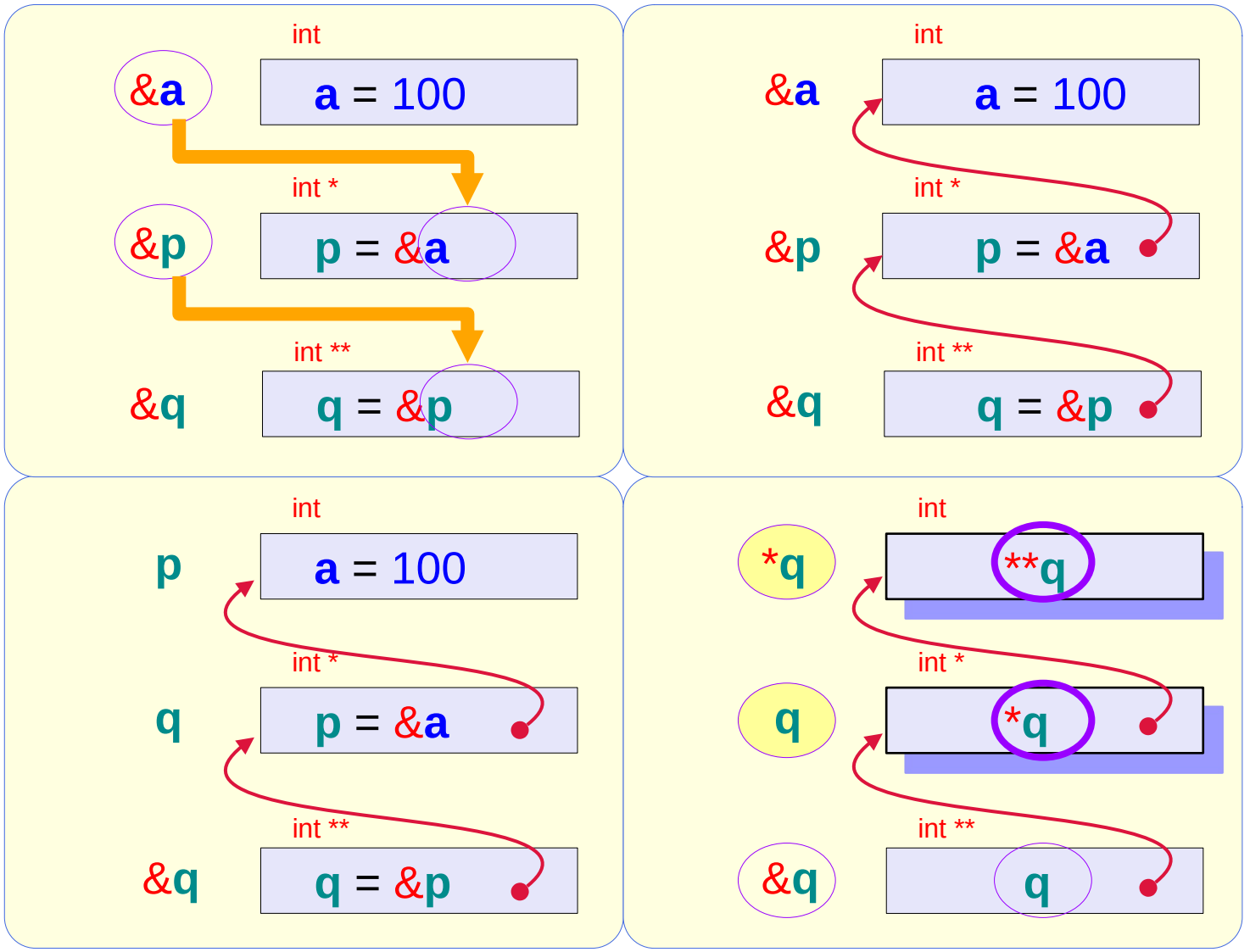


Variable **p**, ***p**, and **q**, ***q**, ****q**

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

p = &a;

q = &p;



Aliased variables : *q, **q

Address assignment | **Variable aliasing**

```
int     a = 100 ;
```

```
p = &a ;      ➡   *p ≡ a
```

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

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

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

```
q = &p ;      ➡   *q ≡ p
              ➡   **q ≡ a
```

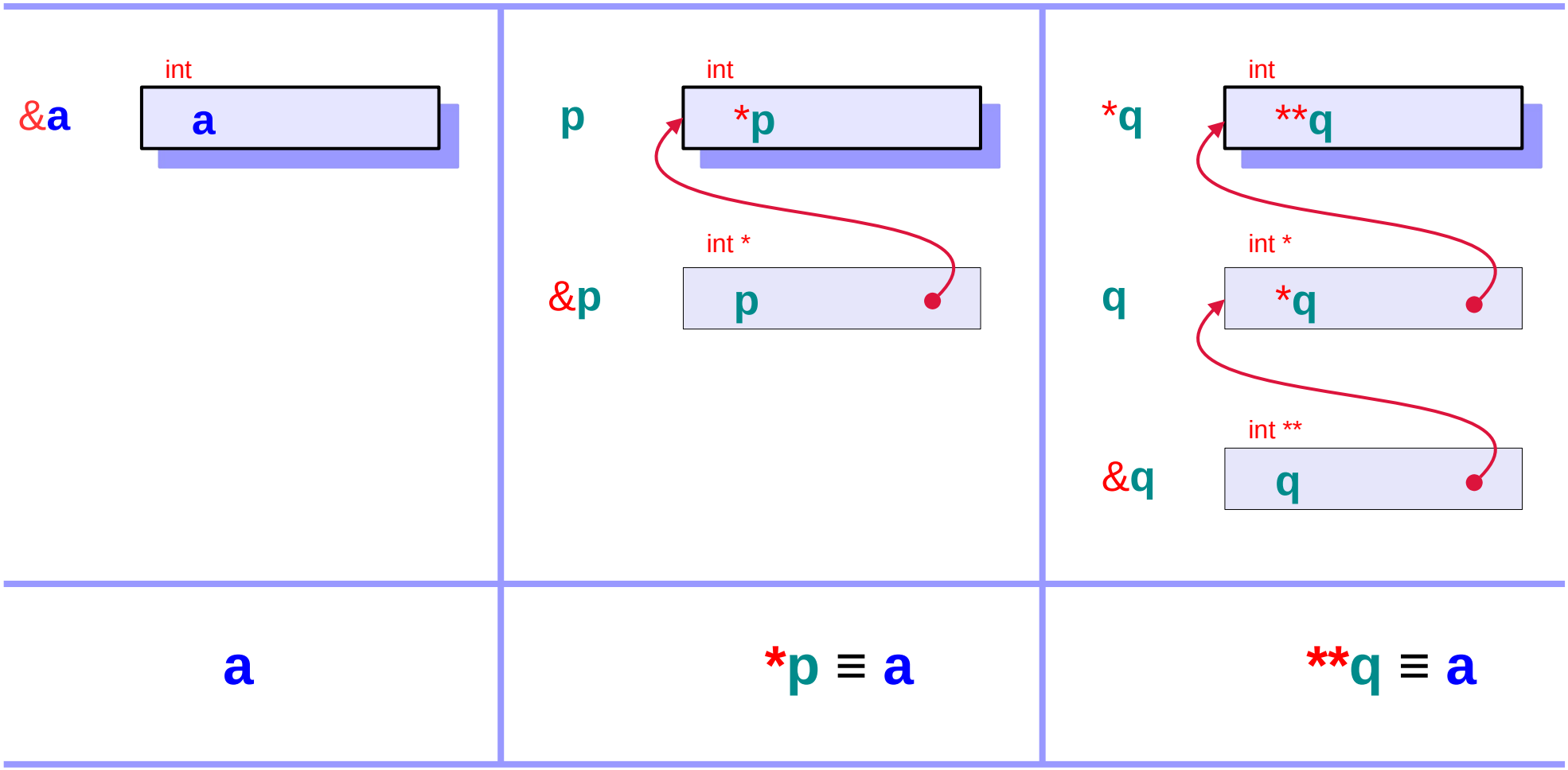
```
p = &a;
```

```
q = &p;
```

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

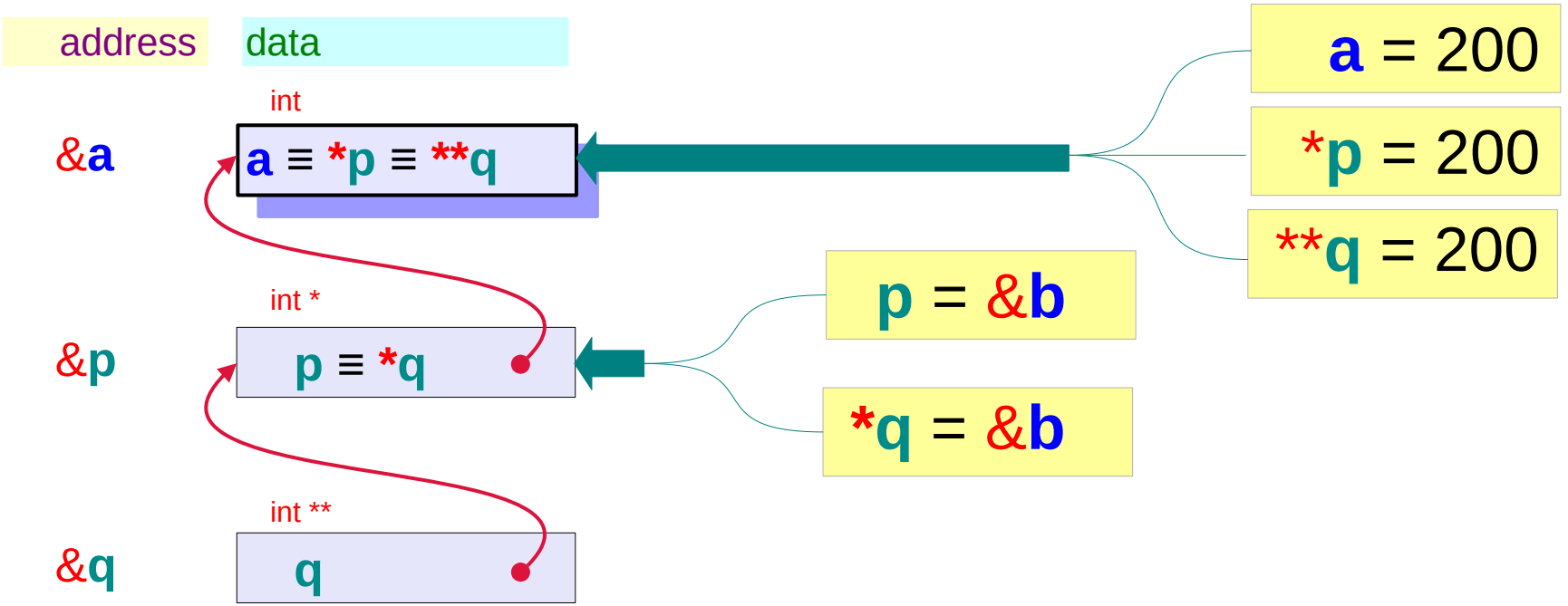
equivalent relations after address assignment

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



a	***p ≡ a**	****q ≡ a**

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



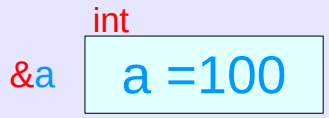
1) Read / Write **p**
2) Read / Write ***q**

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

Single and Double Pointers

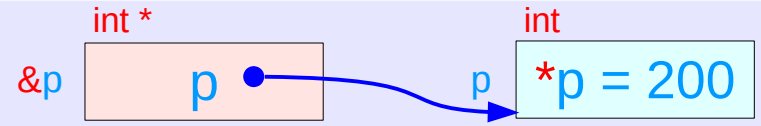
Pointed Addresses and Data

```
int a ;
```



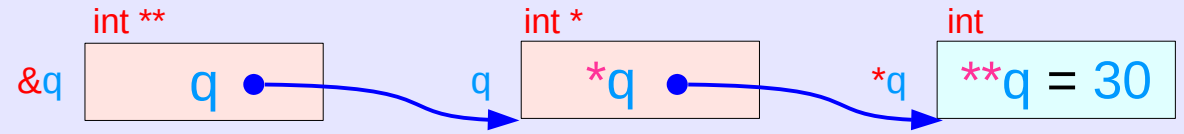
The variable a holds an integer data

```
int * p ;
```



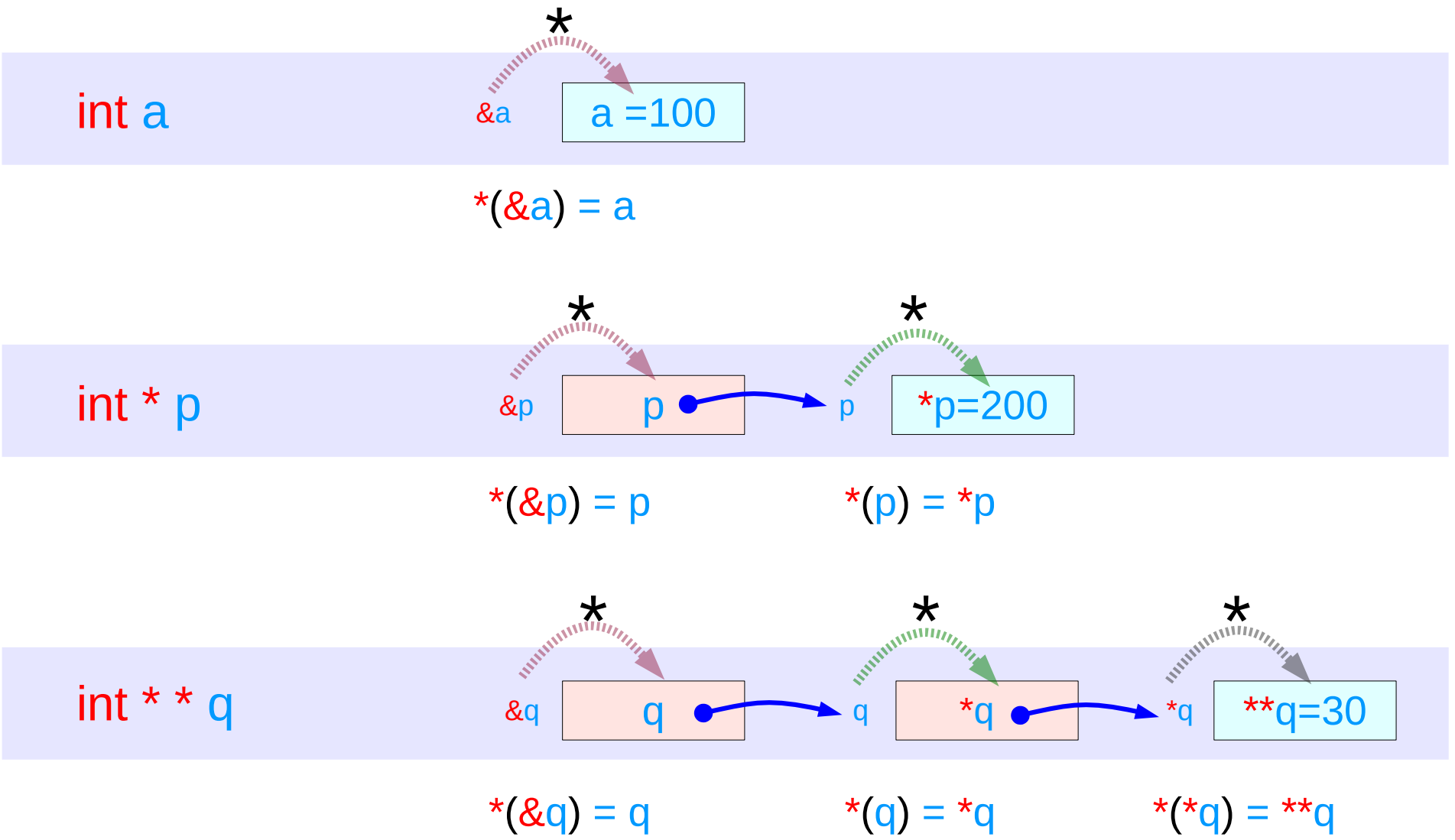
The **pointer** variable p holds an address,
at this address, an integer data is stored

```
int * * q ;
```

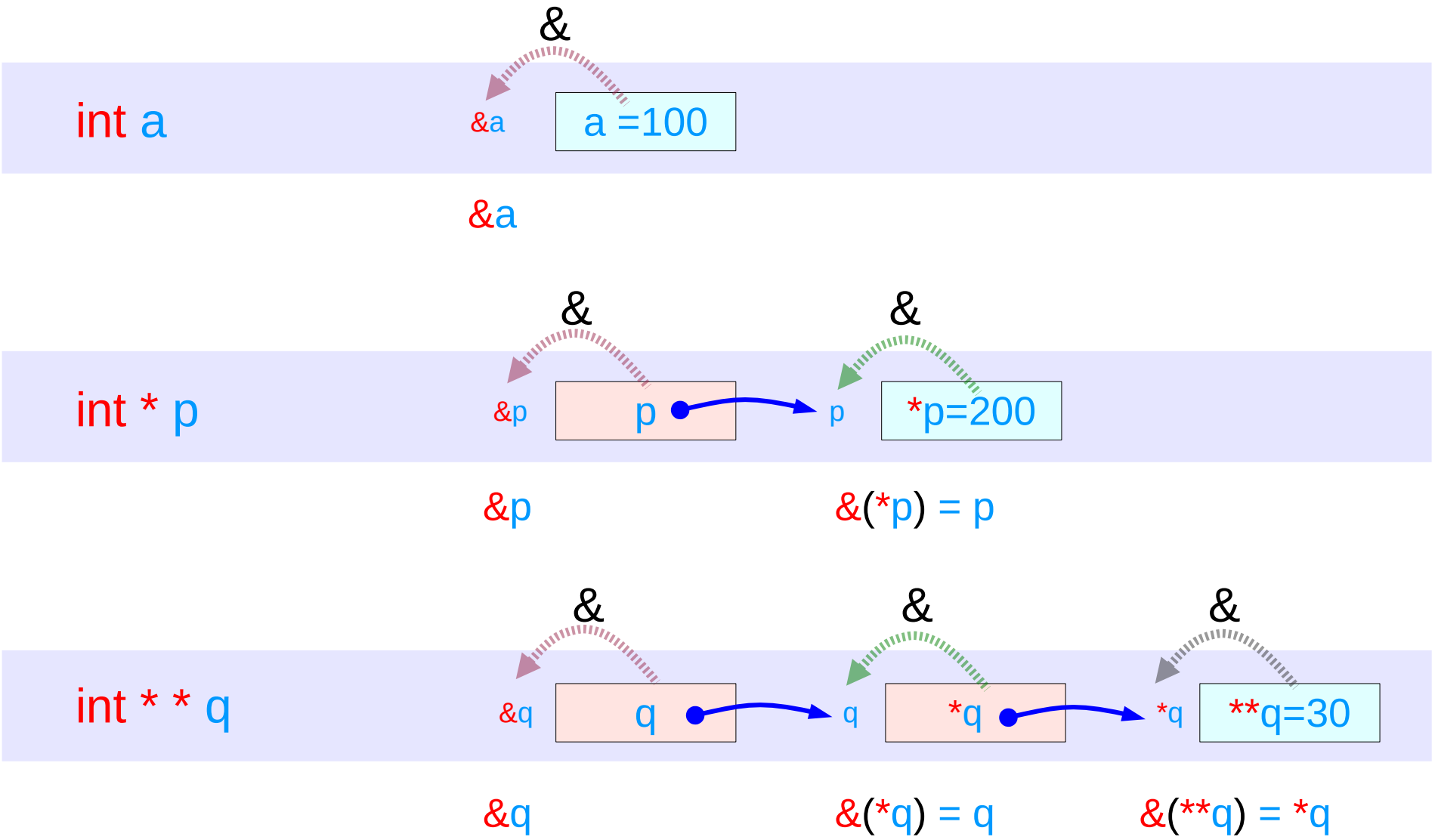


The **pointer** variable q holds an address,
at the address q, another address *q is stored,
at the address *q, an integer data **q is stored

Dereferencing Operator *

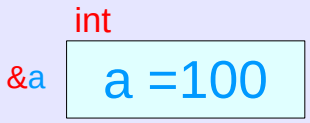


Address-of Operator &

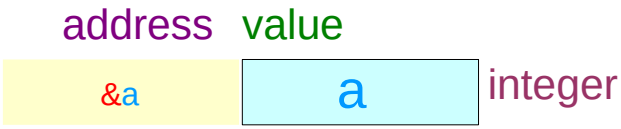


Direct access to an integer **a**

```
int a ;
```



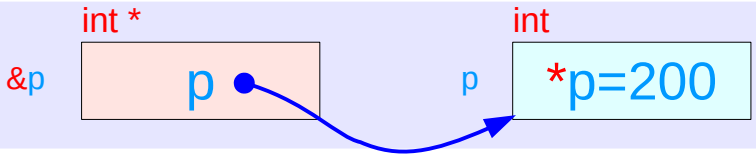
Direct Access



1 memory access

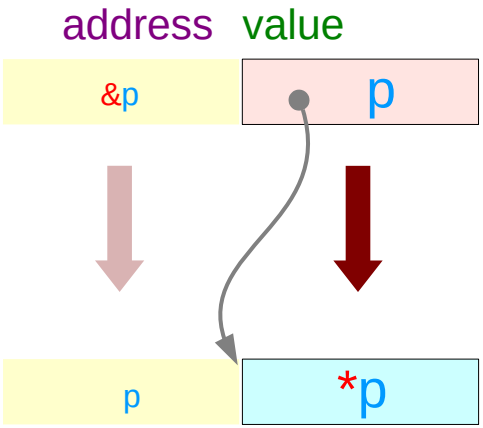
Indirect access *p to an integer a

```
int * p ;
```



Indirect Access

2 memory accesses

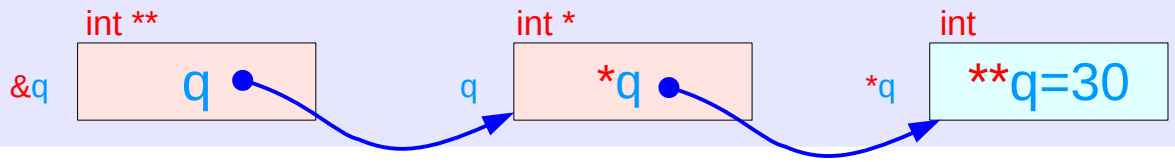


Dereference Operator *

the content of the pointed location

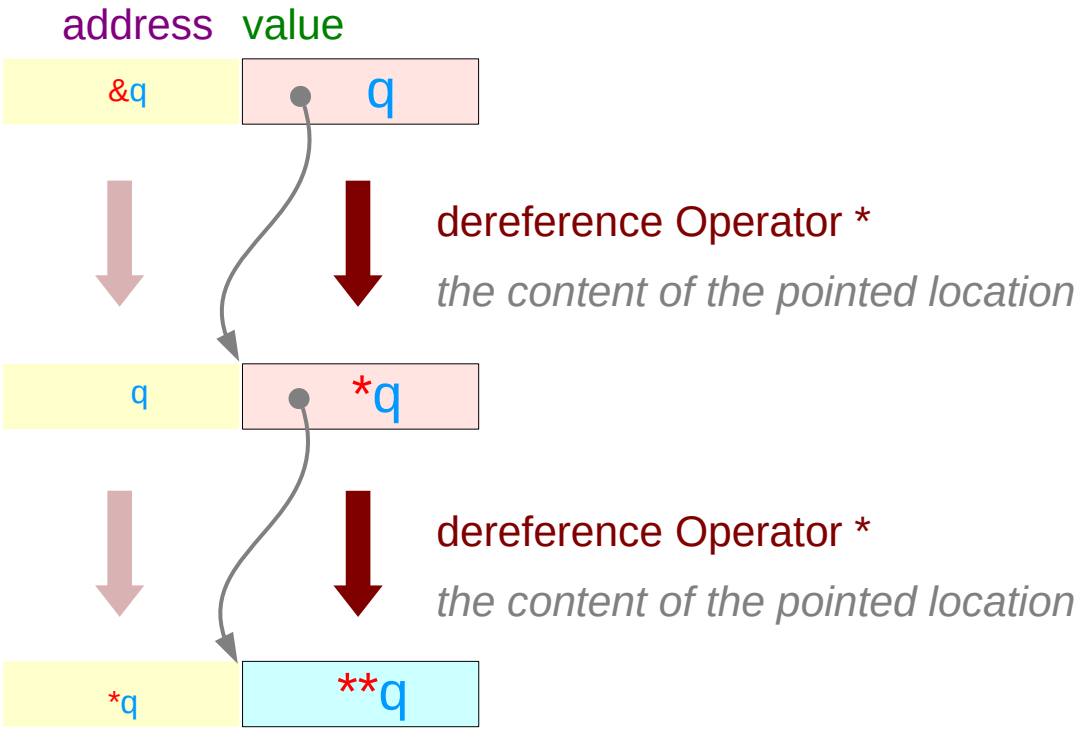
Double indirect access **q to an integer a

```
int * * q ;
```

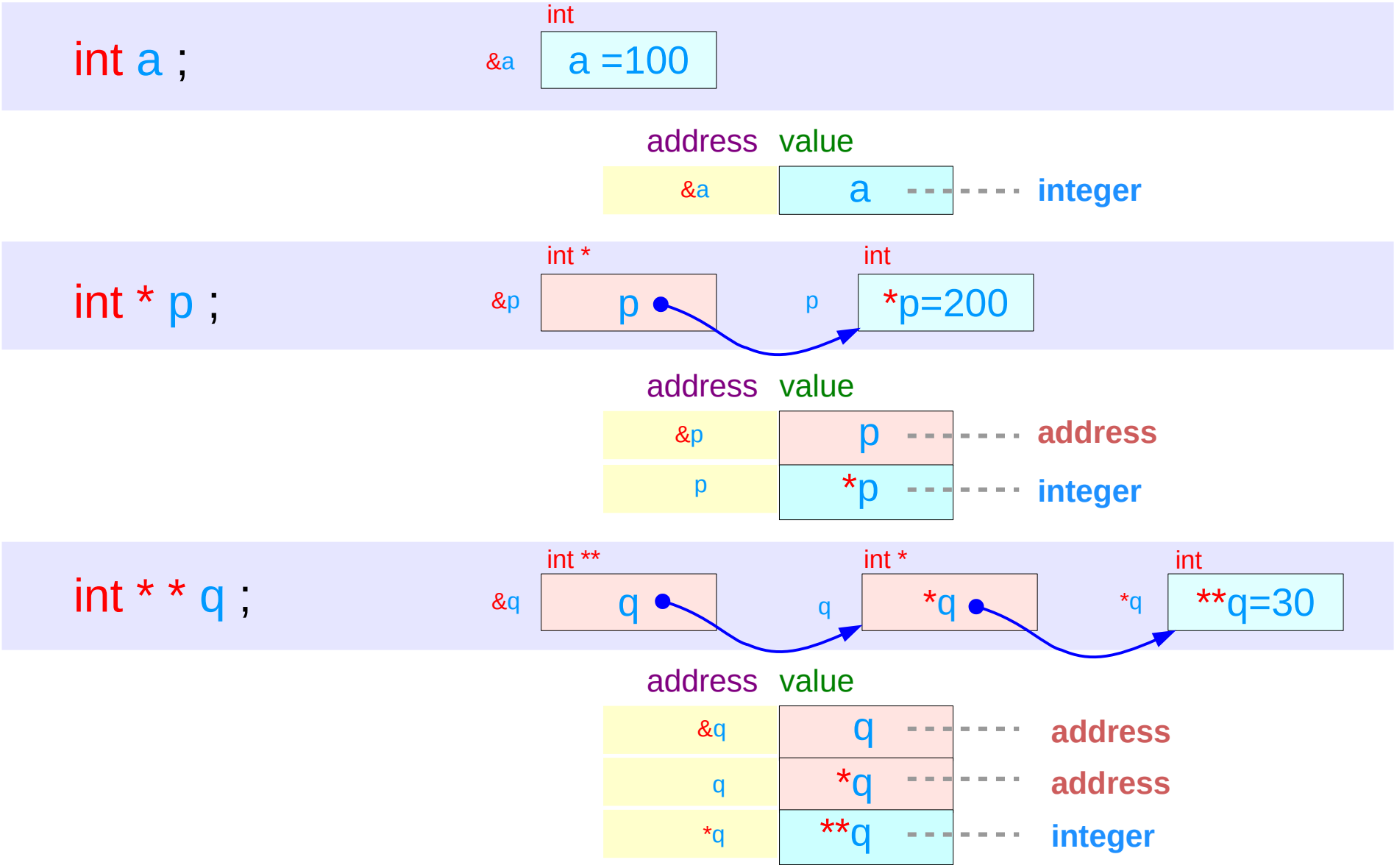


Double Indirect Access

3 memory accesses

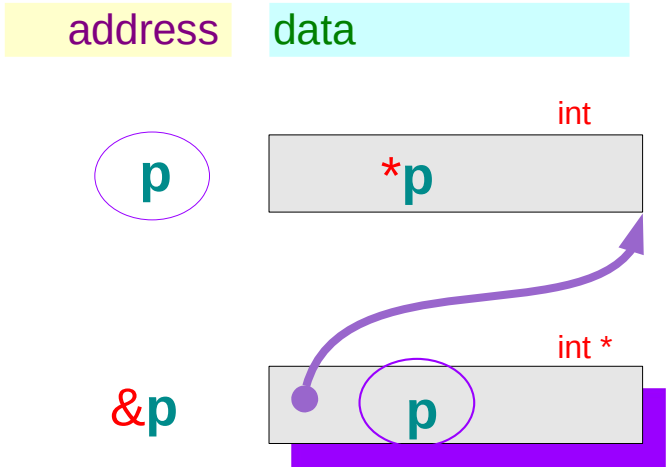


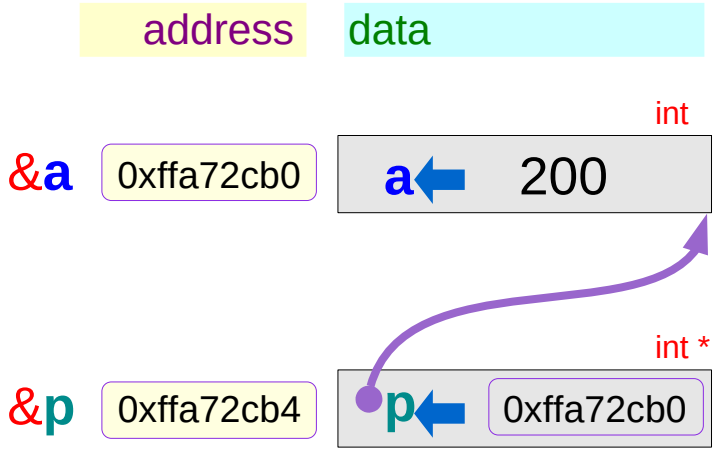
Values of variables



Pointer Variable Example

Pointer variable **p** of the type **int ***

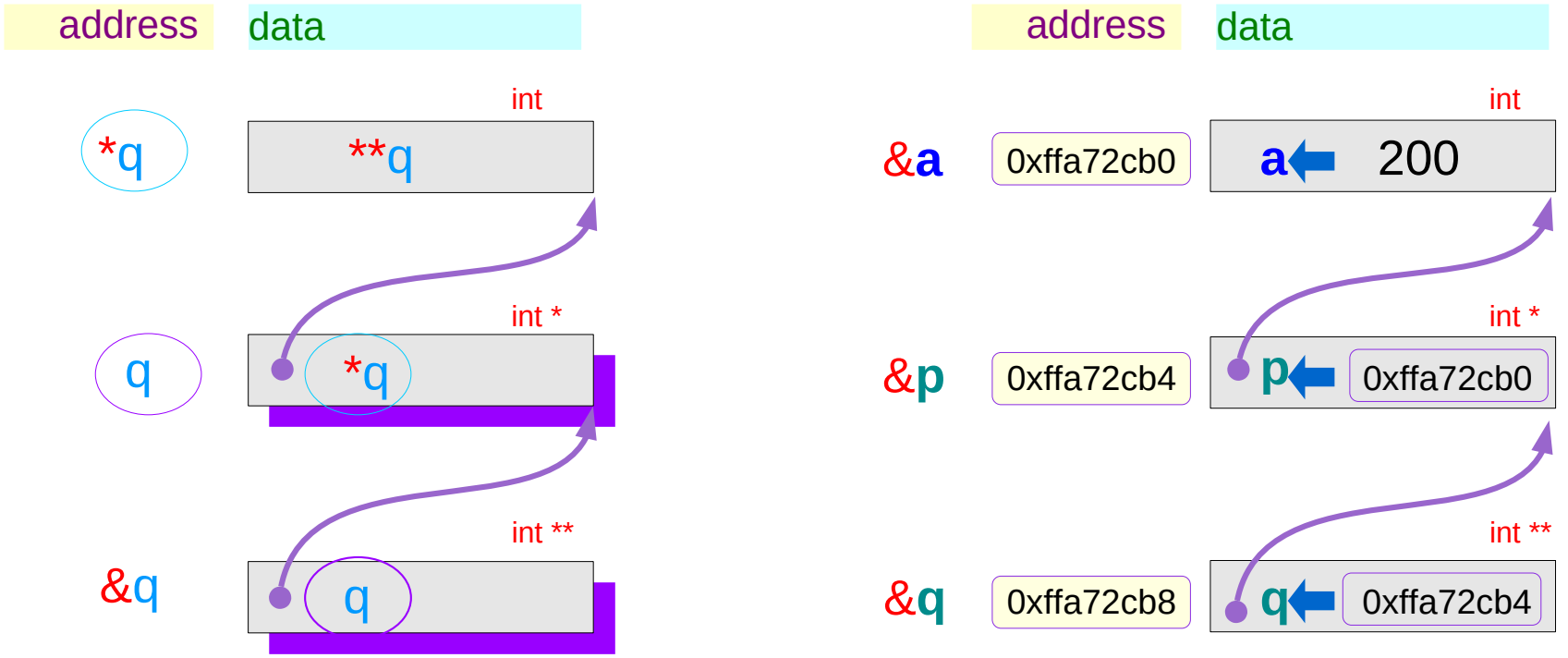





```
int     a = 200;
int *   p = &a;
```

&p ➡ 0xffa72cb4
p ➡ 0xffa72cb0
*p ➡ 200

Pointer variable **q** of the type **int ****




```
int      a = 200;
int *    p = &a;
int ** q = &p;
```

&q ← 0xffa72cb8
q ← 0xffa72cb4
*q ← 0xffa72cb0
**q ← 200

Pointer variable example codes

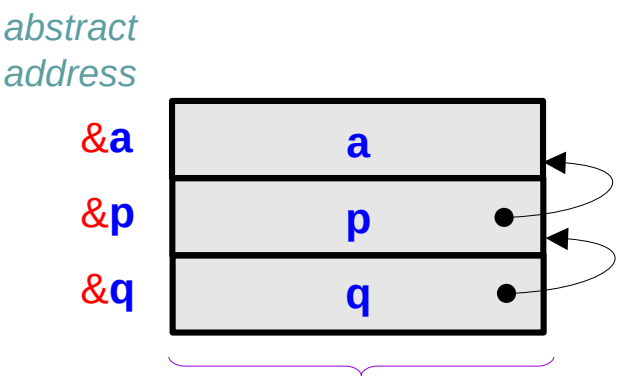
```
// t.c
#include <stdio.h>

int main(void) {
  int   a = 200;
  int  *p = &a;
  int **q = &p;

  printf("&a=%p a=%d \n", &a, a);
  printf("&p=%p p=%p \n", &p, p);
  printf("&q=%p q=%p \n", &q, q);

}
```

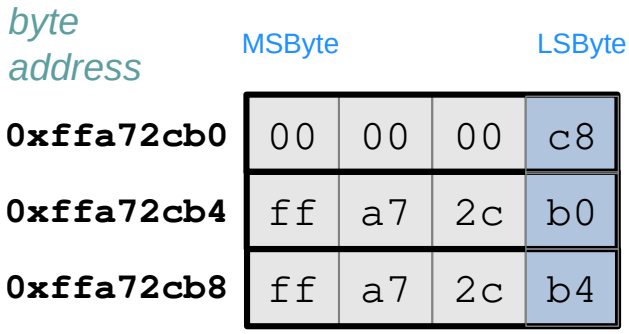
```
  printf("  &p=%12p \n",   &p);
  printf("   p=%12p \n",    p);
  printf("  *p=%12d \n",   *p);
  printf("\n");

  printf("  &q=%12p \n",   &q);
  printf("   q=%12p \n",    q);
  printf("  *q=%12p \n",   *q);
  printf(" **q=%12d \n",  **q);
```

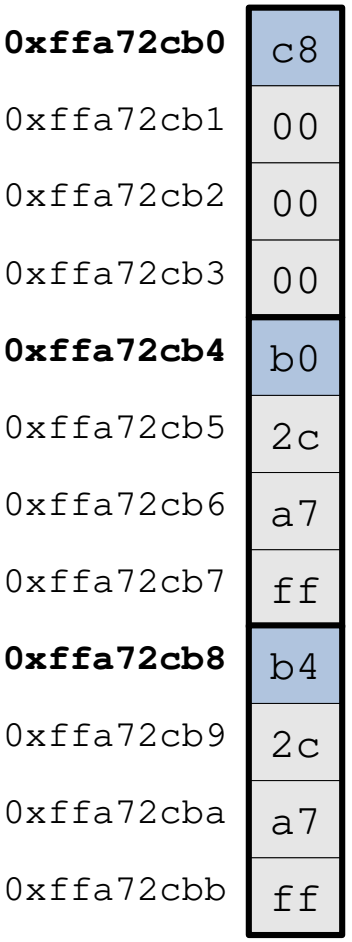
```
gcc -Wall -m32 t.c
./a.out

&a=  0xffa72cb0 a=         200
&p=  0xffa72cb4 p=  0xffa72cb0
&q=  0xffa72cb8 q=  0xffa72cb4
```

```
&p=  0xffa72cb4
 p=  0xffa72cb0
*p=         200

&q=  0xffa72cb8
 q=  0xffa72cb4
*q=  0xffa72cb0
**q=        200
```

Byte addresses of variables







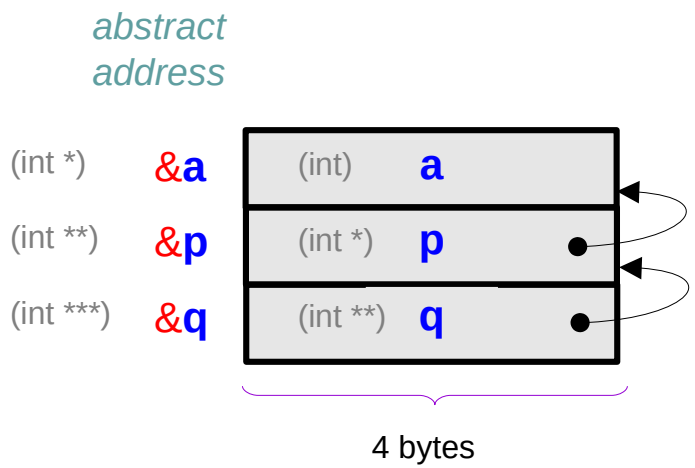

```
int   a = 200 ;
int  *p = &a ;
int **q = &p ;

&a=   0xffa72cb0 ➡ value(&a)
&p=   0xffa72cb4 ➡ value(&p)
&q=   0xffa72cb8 ➡ value(&q)
```

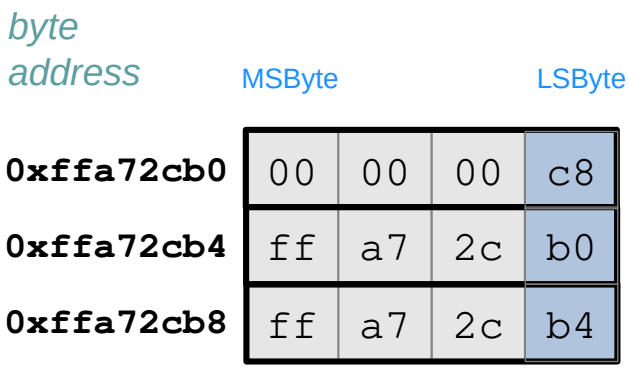
here, the values of &a, &p, &q
are displayed

address value

byte address of
the least significant byte
in a little endian system

Abstract vs. byte addresses





Abstract Address	Type	Size	Value
&a	int *	4 bytes	0xffa72cb0
&p	int **	4 bytes	0xffa72cb4
&q	int ***	4 bytes	0xffa72cb8

contains information of type, size, value

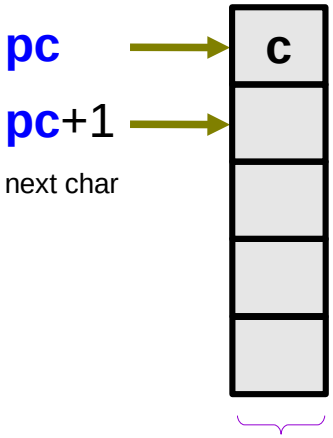
Byte Address	Value
value(&a)	0xffa72cb0
value(&p)	0xffa72cb4
value(&q)	0xffa72cb8

contains only value information

Endianness and memory alignment

Pointers and abstract addresses

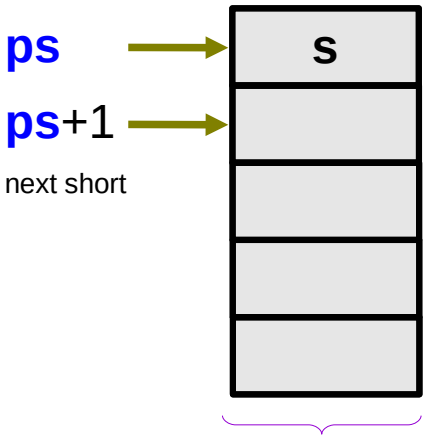
```
char c ;
char *pc = &c ;
```



sizeof(*pc) =
sizeof(c) =
sizeof(int) = 1 byte

sizeof(pc) = 4 or 8 bytes
type(pc) = char *
value(pc) = byte address

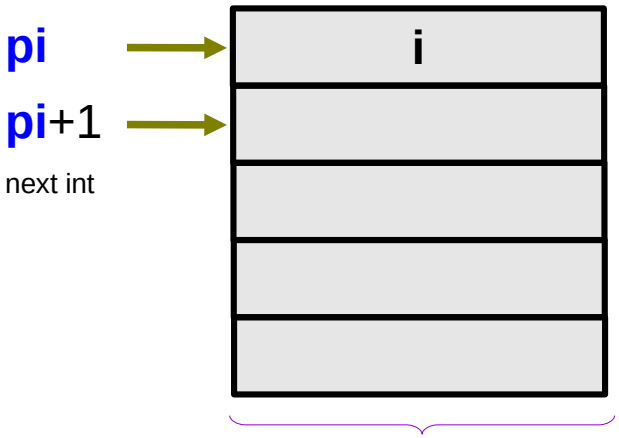
```
short s ;
short *ps = &s ;
```



sizeof(*ps) =
sizeof(s) =
sizeof(short) = 2 bytes

sizeof(ps) = 4 or 8 bytes
type(ps) = short *
value(ps) = byte address

```
int i ;
int *pi = &i ;
```

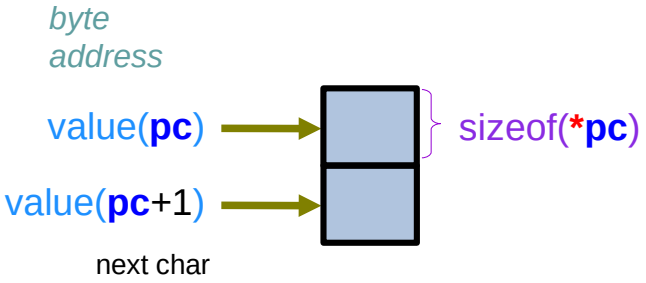


sizeof(*pi) =
sizeof(i) =
sizeof(int) = 4 bytes

sizeof(pi) = 4 or 8 bytes
type(pi) = int *
value(pi) = byte address

Pointer values and byte addresses

char ***pc** ;

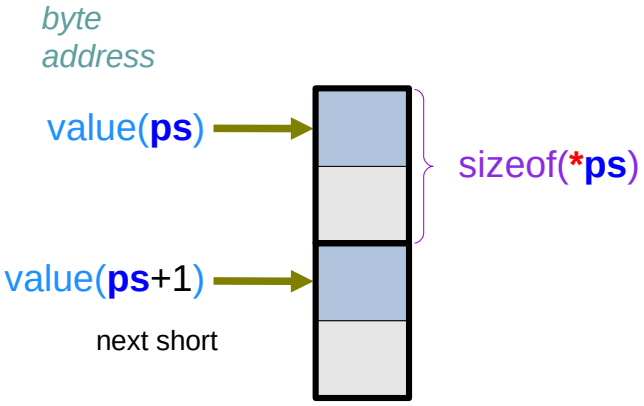


The value of a *pointer* (address value) is the byte address of the least significant byte in a little endian system

pc points to the least significant byte, i.e., the **1** byte of **char**

value(**pc**+1)
= value(**pc**) + sizeof(***pc**)

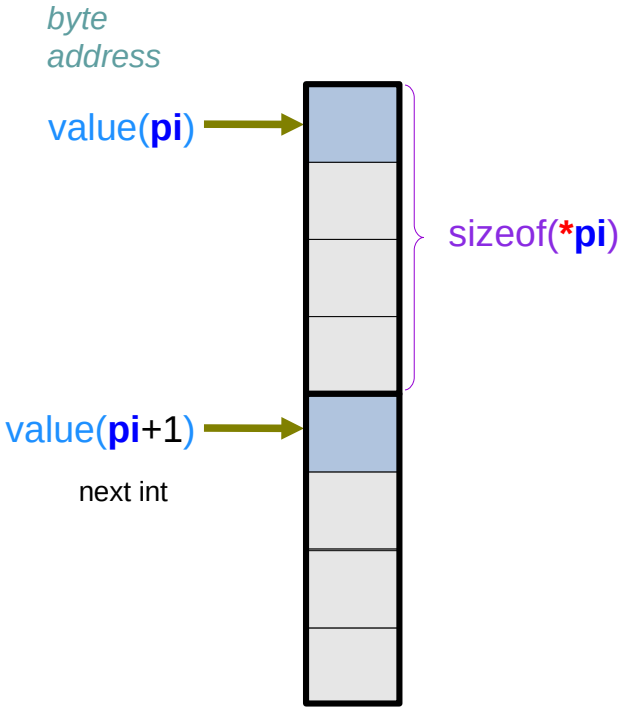
short ***ps** ;



ps points to the least significant byte among the **2** bytes of **short**

value(**ps**+1)
= value(**ps**) + sizeof(***ps**)

int ***pi** ;



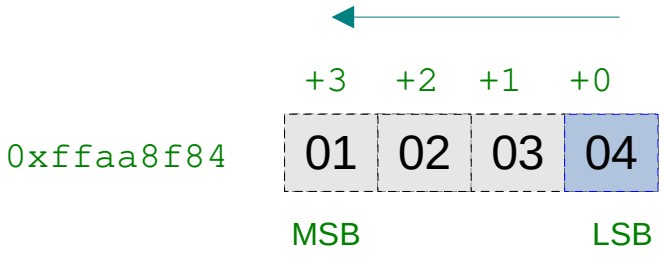
pc points to the least significant byte among the **4** bytes of an **int**

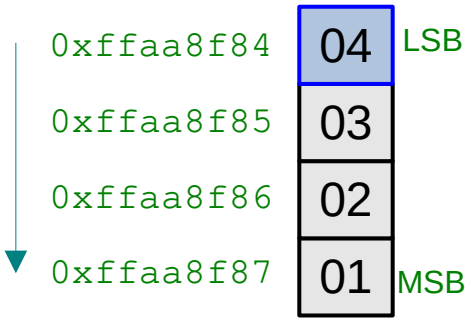
value(**pi**+1)
= value(**pi**) + sizeof(***pi**)

Little Endian and Big Endian

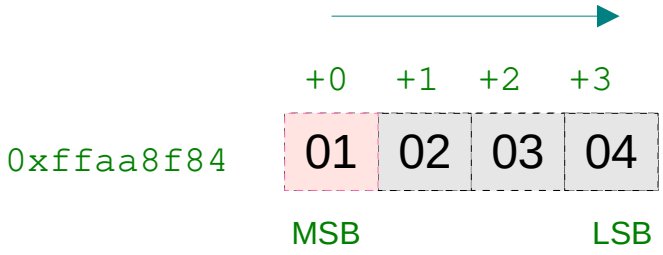
MSB LSB

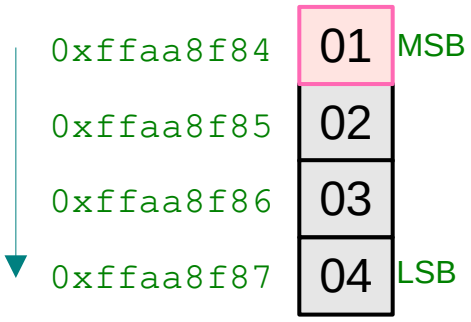
```
int i = 0x01020304;
```





Little Endian
stores LSByte first

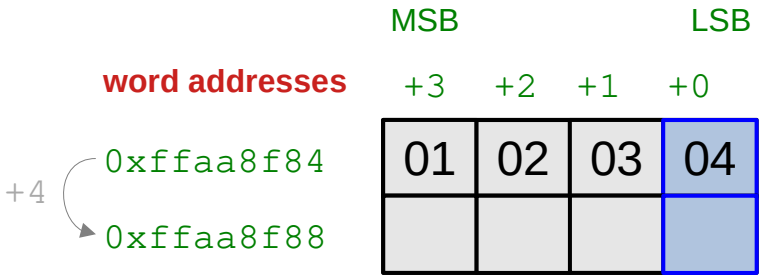


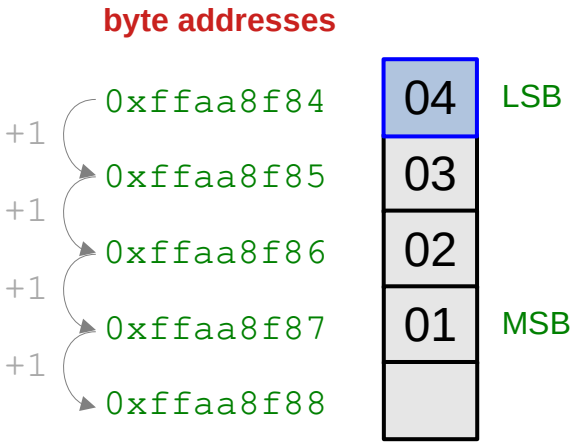


Big Endian
stpres MSByte first

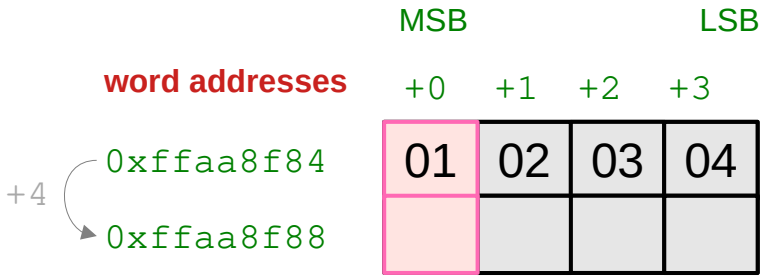
Word addresses and byte addresses

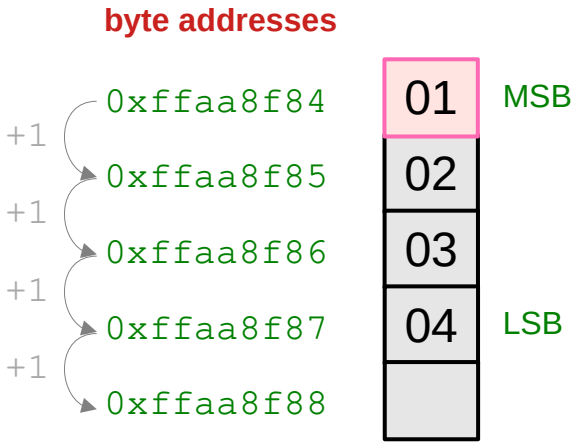
Little Endian store LSByte first





Big Endian store MSByte first

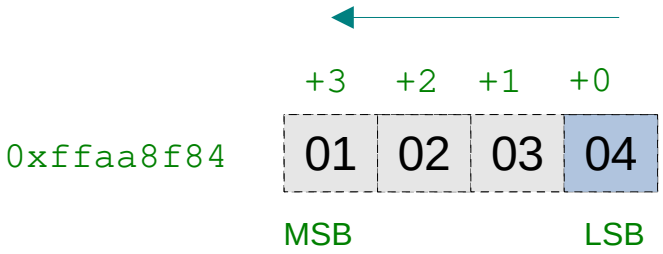




Little Endian Examples

MSB LSB

```
int i = 0x01020304;
```



Little Endian

store LSB first

Least Significant Byte (**LSB**)

```
                MSB     LSB
int i    = 0x01020304;
char *p = (char *)&i;

printf("%p \n", &i);

printf("%p : %x \n", p+0, p[0]);
printf("%p : %x \n", p+1, p[1]);
printf("%p : %x \n", p+2, p[2]);
printf("%p : %x \n", p+3, p[3]);
```

```
                0xffaa8f84
LSB address     0xffaa8f84 : 4    LSB data
                0xffaa8f85 : 3
                0xffaa8f86 : 2
MSB address     0xffaa8f87 : 1    MSB data
```

assume 32-bit CPU
32-bit address
32-bit data (1 word, 4 bytes)

Storing values in a little endian system

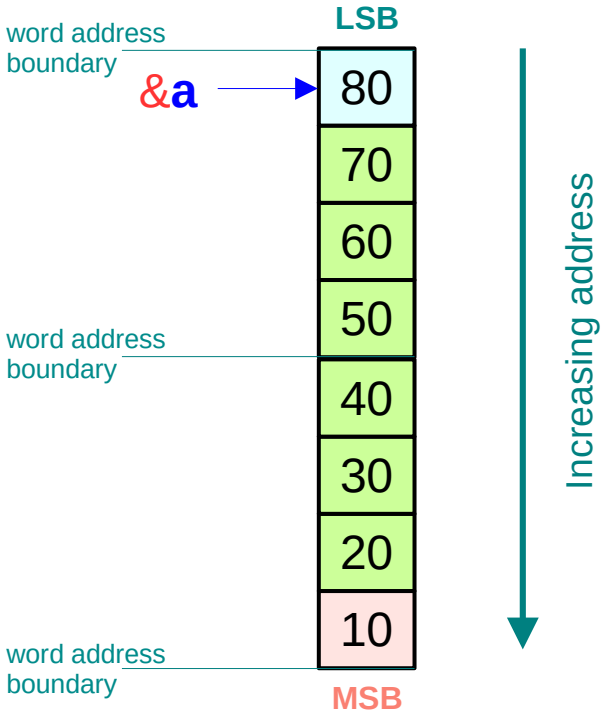
long **a** = 0x1020304050607080 ;

MSB LSB

MSB (Most Significant Byte)
LSB (Least Significant Byte)

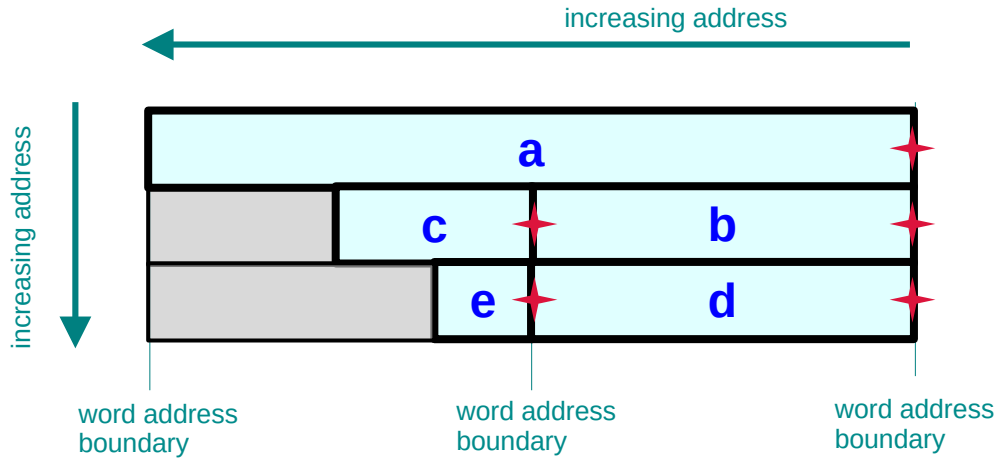
Little Endian System

store the LSB first

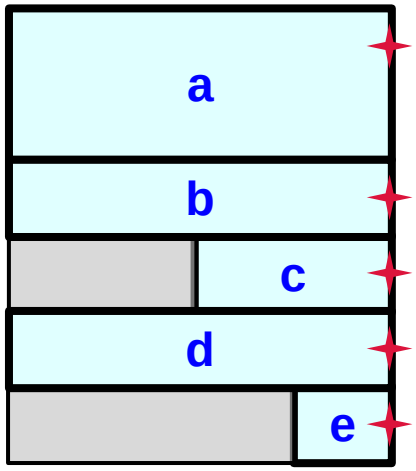


Memory alignment in a little endian system

10	20	30	40	50	60	70	80
		12	34	11	22	33	44
			99	01	02	03	04

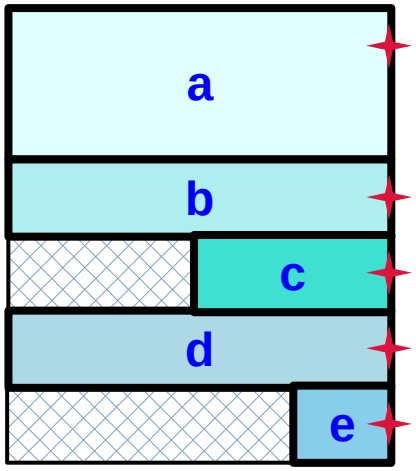



```
long    a = 0x10203040506070 80 ;
int     b = 0x1122334 4 ;
short   c = 0x12 34 ;
int     d = 0x0102030 4 ;
char    e = 0x 99 ;
```

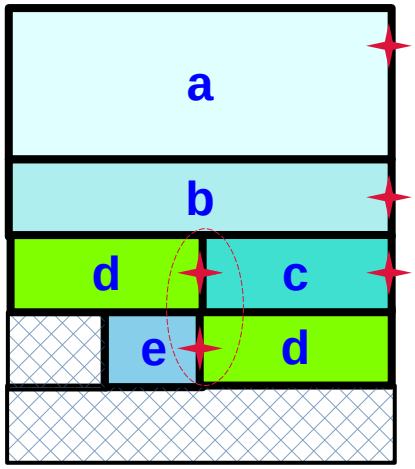


Memory alignment – performance

Memory alignment



No memory alignment



saves memory

loses performance :

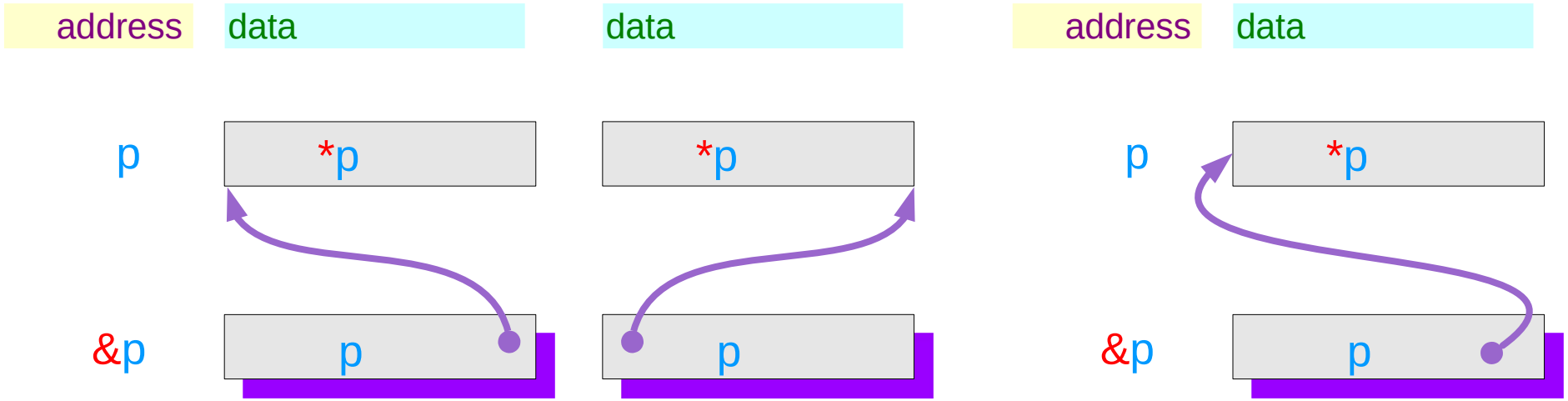
accessing **d** may require
two memory cycles
instead of one

memory hardware constraints

```
long    a = 0x1020304050607080 ;
int     b =         0x11223344 ;
short   c =               0x1234 ;
int     d =         0x01020304 ;
char    e =               0x99 ;
```

My Arrow Notations

My arrow notations (1)



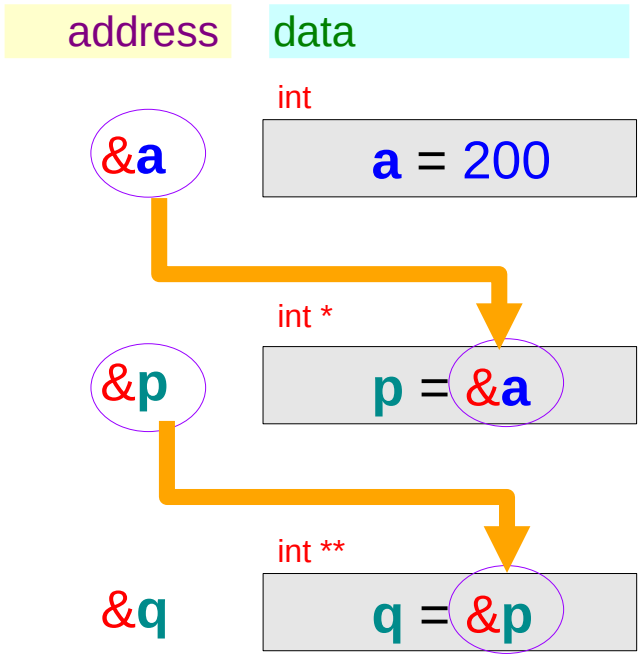
Word address =
MSByte address

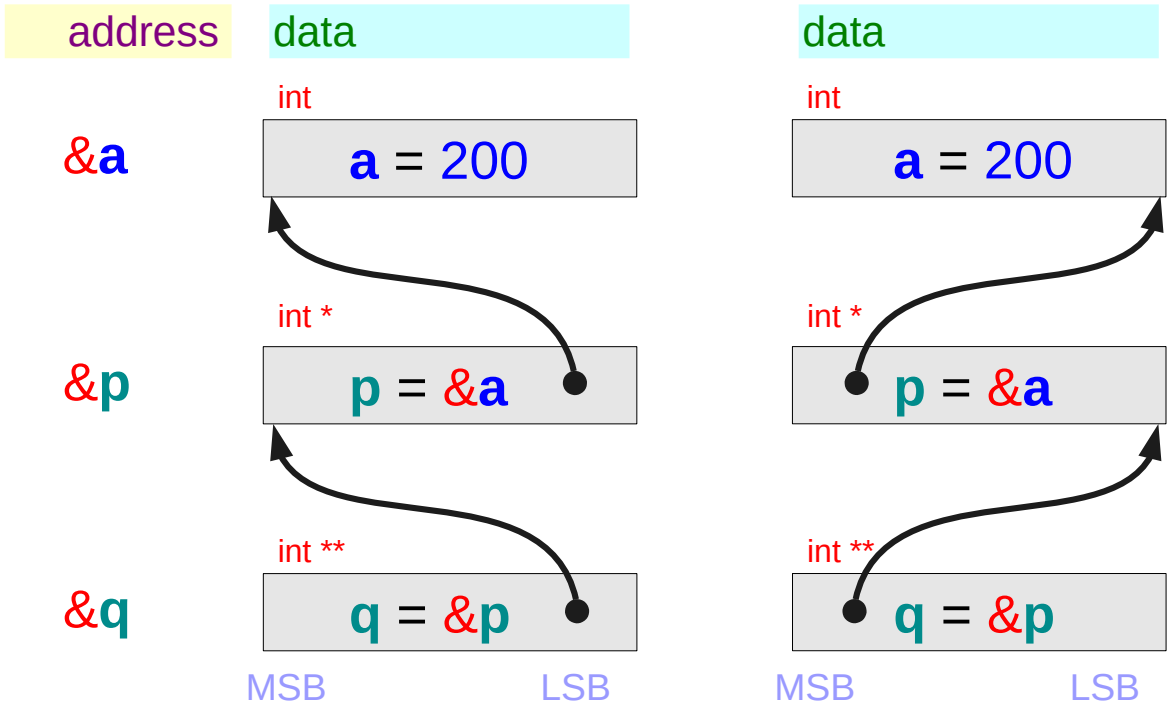
Most Significant Byte

Word address =
LSByte address

Least Significant Byte

My arrow notations (2)






```
int     a = 200;
int *   p = &a;
int ** q = &p;
```

Big Endian

Little Endian

Arrays

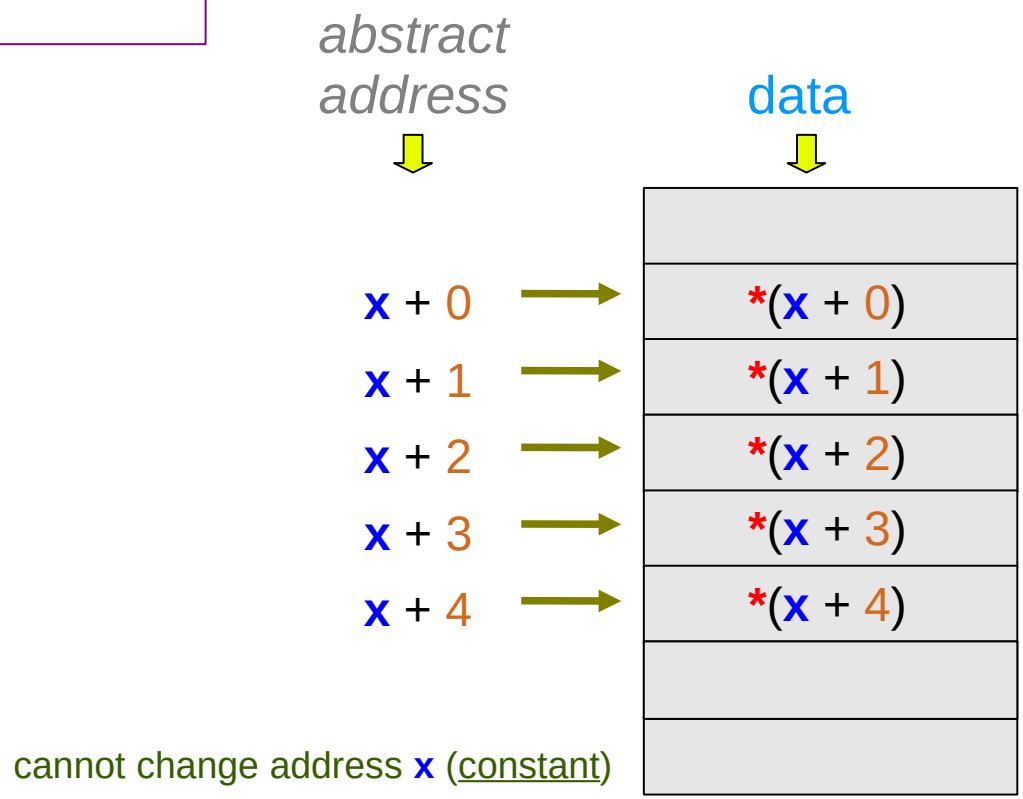
Accessing array elements – using abstract addresses

```
int     x[5] ;
```

x holds the *starting address*
*of **5** consecutive **int** variables*

5 int variables

index	data
0	x[0]
1	x[1]
2	x[2]
3	x[3]
4	x[4]



Accessing array elements – using byte addresses

int x[5] ;

x holds the *starting address*
*of **5** consecutive **int** variables*

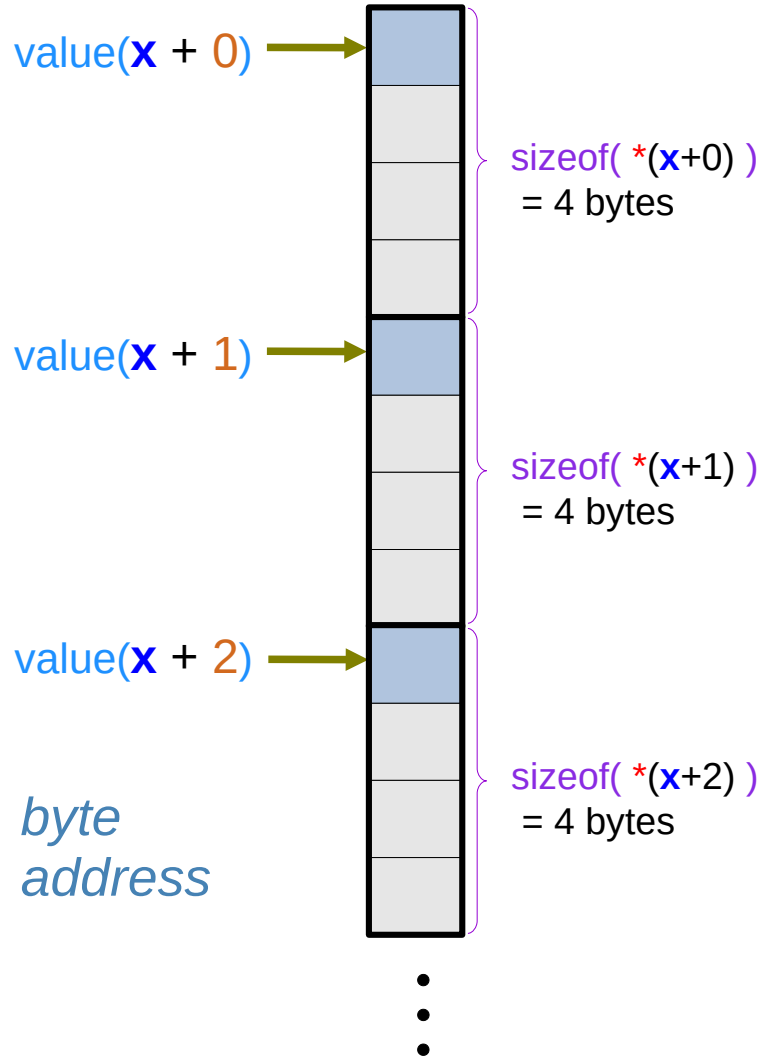
5 int variables

value(**x** + 0) = value(**x**) + 0 * sizeof(***x**)

value(**x** + 1) = value(**x**) + 1 * sizeof(***x**)

value(**x** + 2) = value(**x**) + 2 * sizeof(***x**)

byte address byte address sizeof(int) = 4 bytes



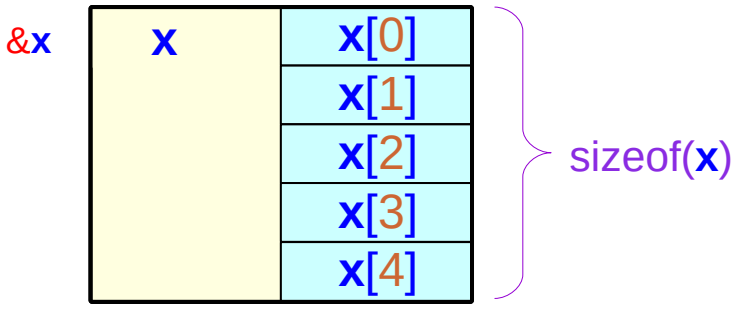
Value and size of an array variable

```
int x [5] ;
```

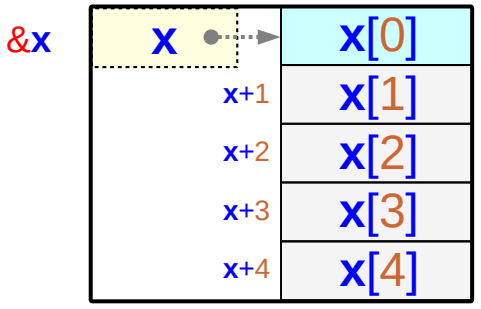
x : an array variable name (constant)

sizeof(**x**) : the *total size* of
5 consecutive **int** variables

value(**x**) : the *starting address* of
5 consecutive **int** variables



sizeof(**x**) = 5 * sizeof(int)



value(**x**) = value(&**x**[0])

x : can be viewed as a pointer

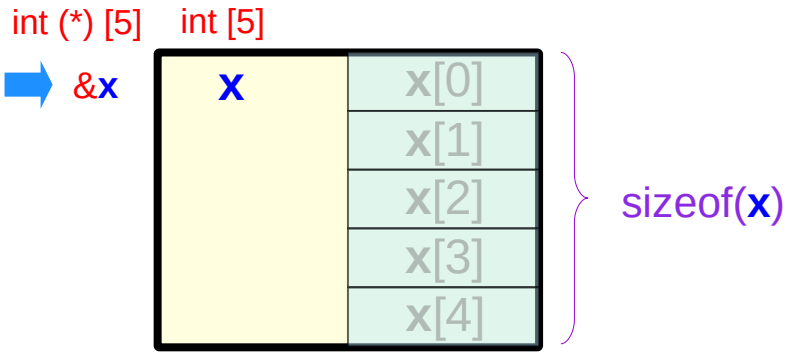
x holds the address of
the 1st array element **x[0]**
x = &x[0], *x ≡ x[0]

Types, sizes, and addresses of **x** and ***x**

```
int x [5] ;
```

x : an array variable name (constant)

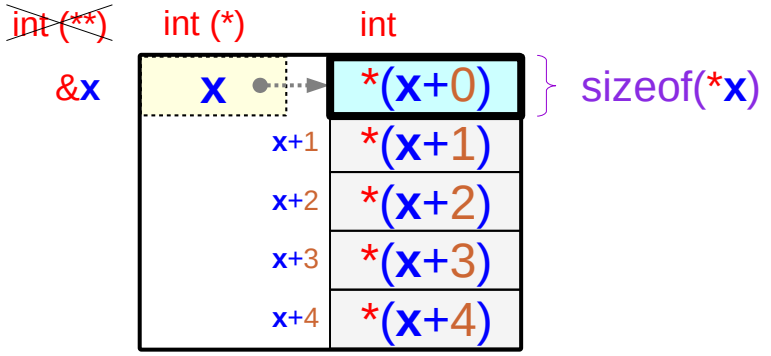
outside of an array x



Abstract Data x

Address	value(&**x**) = value(**x**)
Size	sizeof(**x**) = 5 * sizeof(int)
Type	int [5]

inside of an array x

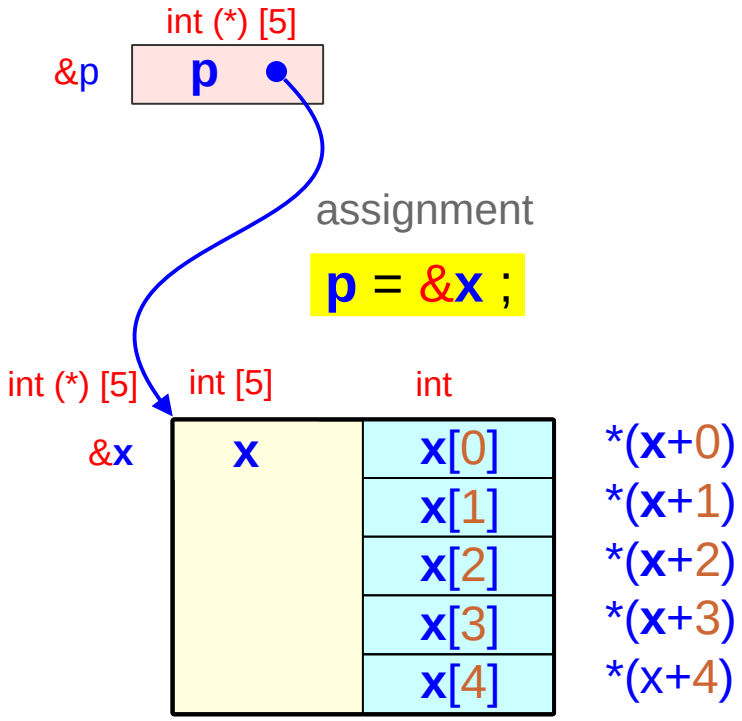


Primitive Data *x = x[0]

Address	value(&**x**[0]) = value(**x**)
Size	sizeof(**x**[0]) = sizeof(int)
Type	int

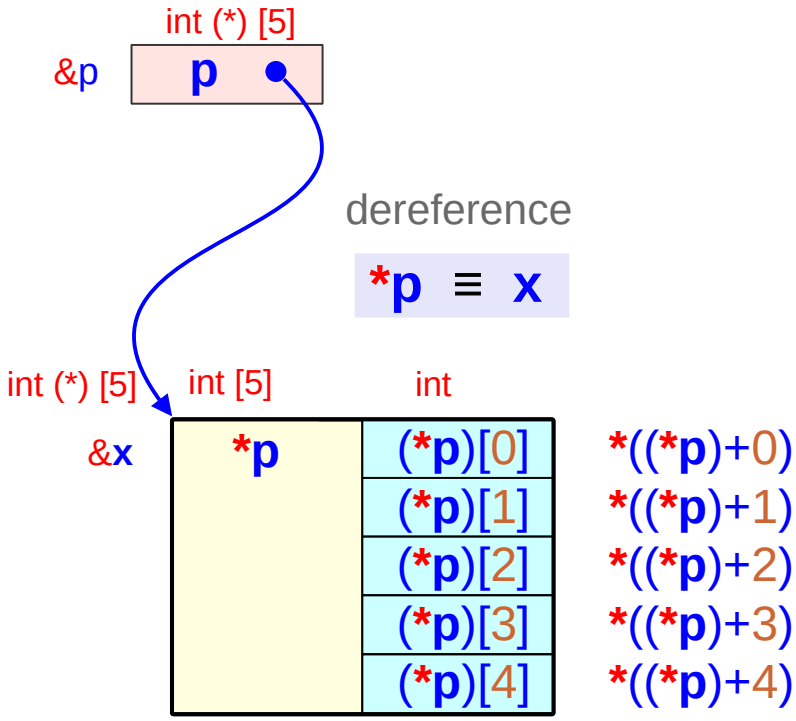
Using a **1-d** array pointer **p**

```
int x [5] ;
```



*x ≡ x[0]
*(x+i) ≡ x[i]

```
int  (*p) [5] ;
```

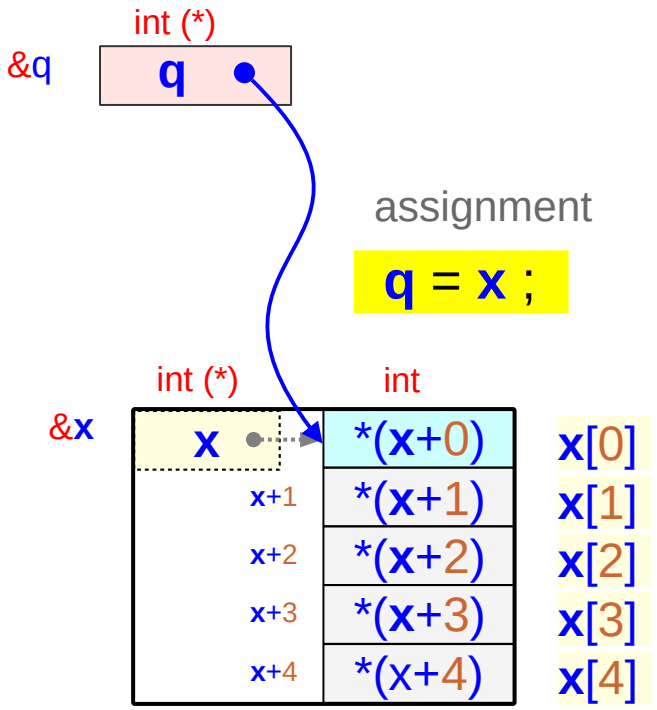


*(*p) ≡ (*p)[0]
*((*p)+i) ≡ (*p)[i]

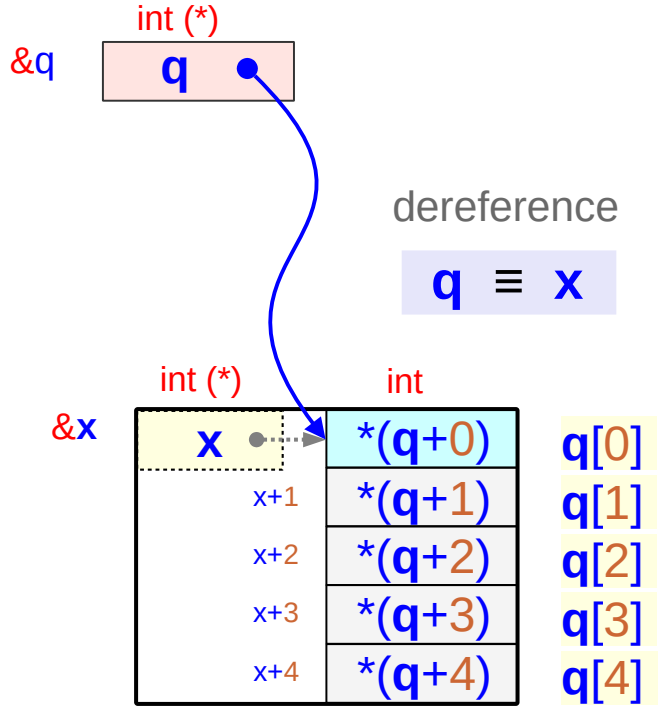
Using a **0-d** array pointer **q**

```
int x [5] ;
```

```
int * q = x ;
```



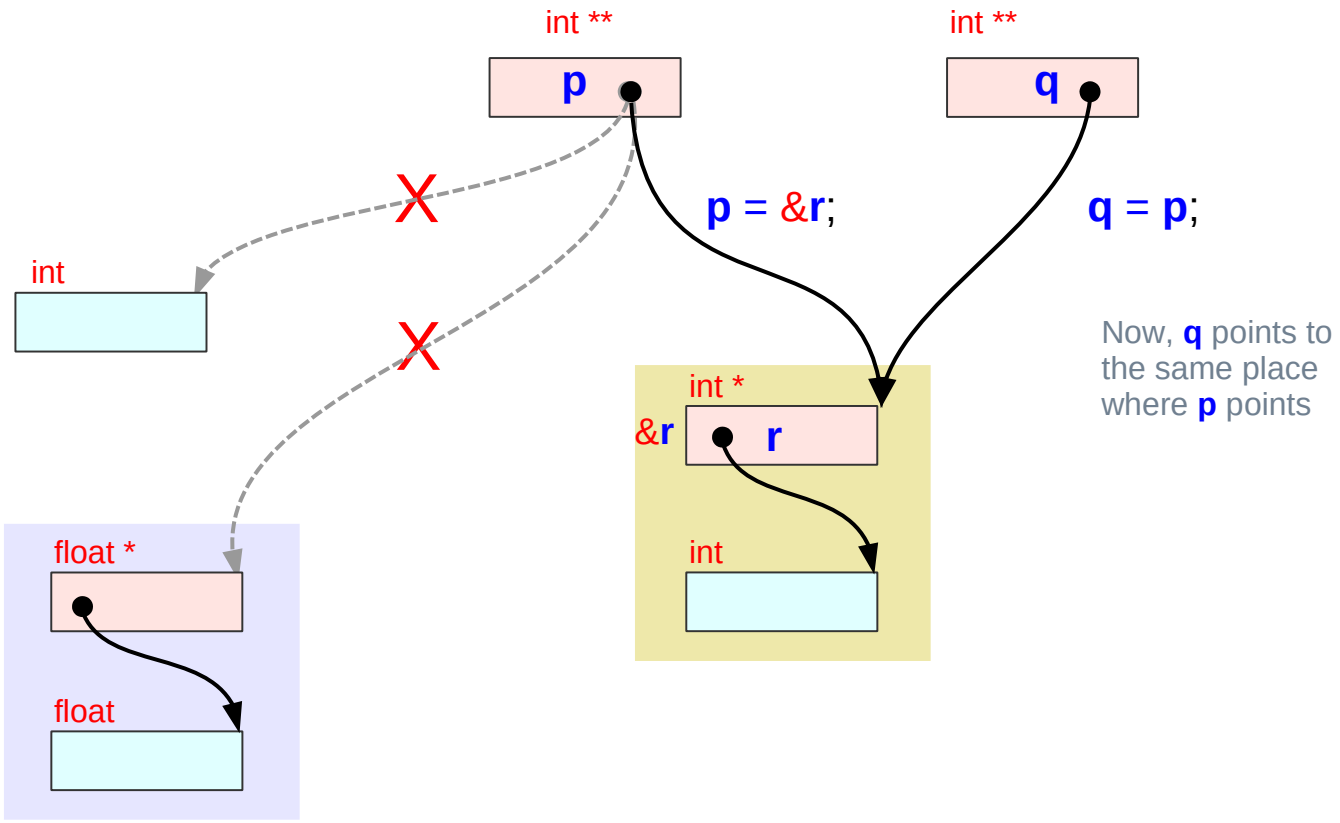
*x ≡ x[0]
*(x+i) ≡ x[i]



*q ≡ q[0]
*(q+i) ≡ q[i]

Double pointer variable assignments

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

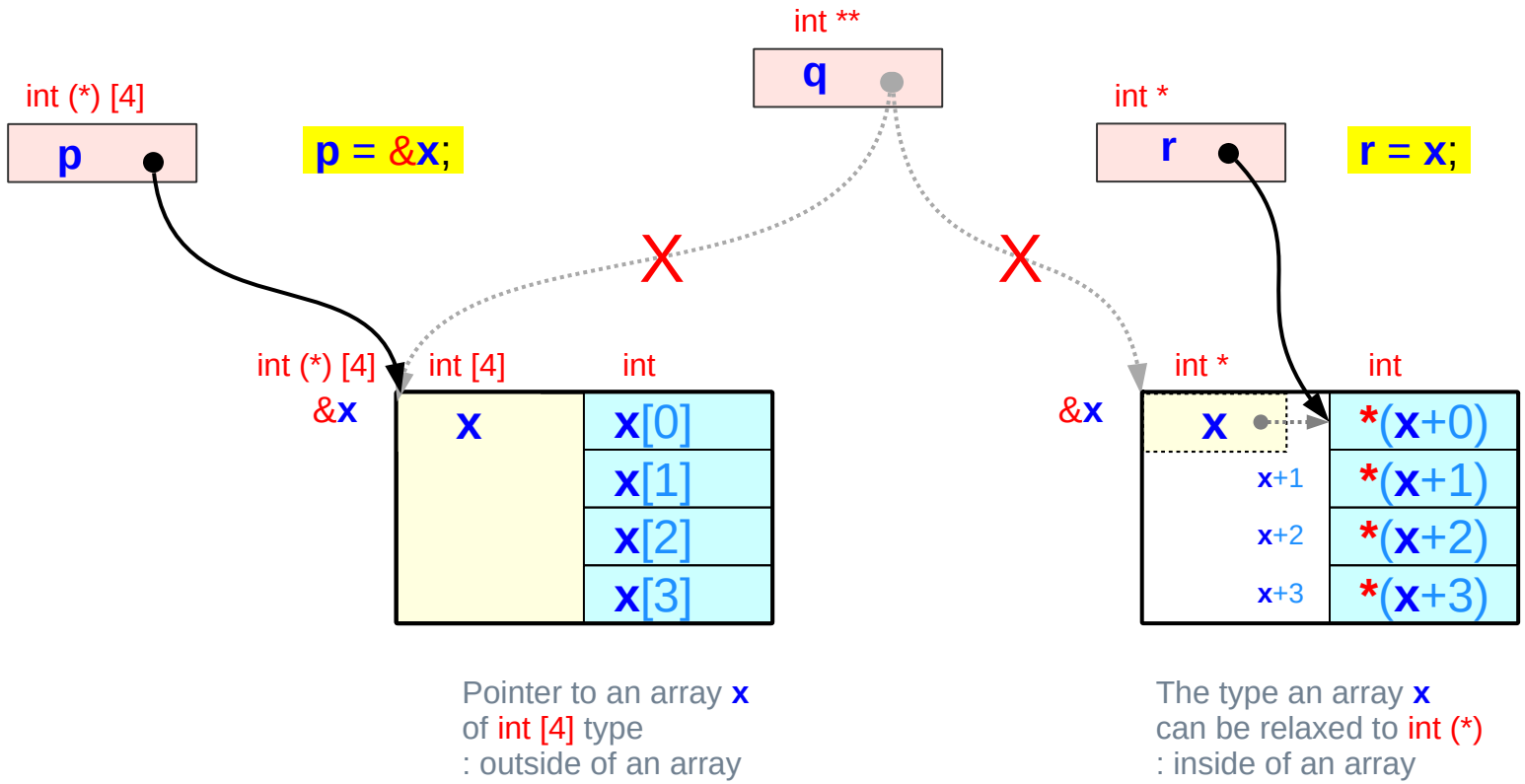


Double pointer variable assignments

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

```
int  ** q ;
```

```
int  *r ;
```



Pointers, arrays, and operator precedence

Operator Precedence of * and [ ]

*x[m] ⟺ *(x[m])

x[m][n] ⟺ (x[m])[n]

**x ⟺ *(*x)

[ ] has a higher priority than *

Left-to-right associativity

Right-to-left associativity

(*x)[m][n] ⟺ ((*x)[m])[n]

(*x[m])[n] ⟺ (*(x[m]))[n]

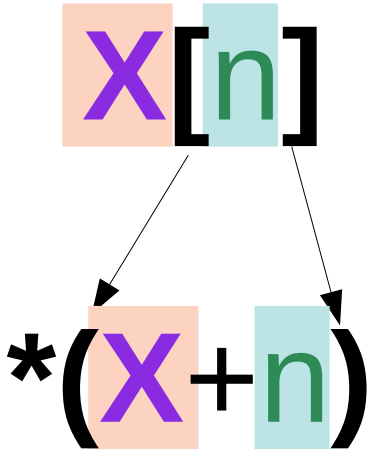
( ) must be used

( ) can be removed

Left-to-right associative [ ]

p[i]	⟺	p[i]	⟺	*(p+i)
p[i][j]	⟺	(p[i])[j]	⟺	*(*(p+i)+j)
p[i][j][k]	⟺	((p[i])[j])[k]	⟺	*(*(*(p+i)+j)+k)

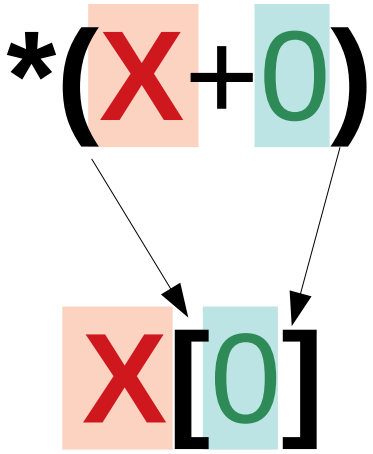
*(X+n) ≡ X[n]



Right-to-left associative *, with **i = 0**

***p**	⟺	***(p)**	⟺	**p[0]**
****p**	⟺	***(*(p))**	⟺	**(p[0])[0]**
*****p**	⟺	***(*(*(p)))**	⟺	**((p[0])[0])[0]**

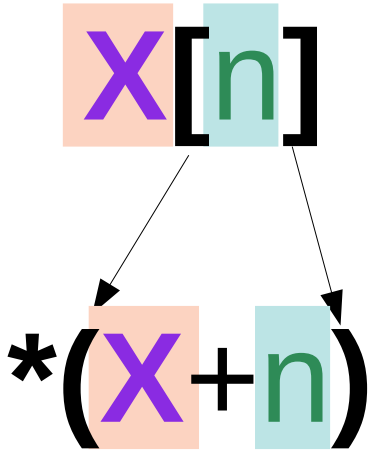
***(X+n)** ≡ **X[n]**

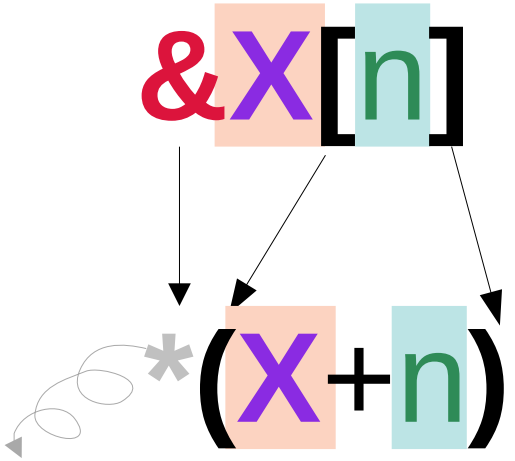


Equivalences

p[i]	⟺	*(p+i)
p[i][j]	⟺	*(*(p+i)+j)
p[i][j][k]	⟺	*(*(*(p+i)+j)+k)

&p[i]	⟺	(p+i)
&p[i][j]	⟺	(*(p+i)+j)
&p[i][j][k]	⟺	(*(*(p+i)+j)+k)

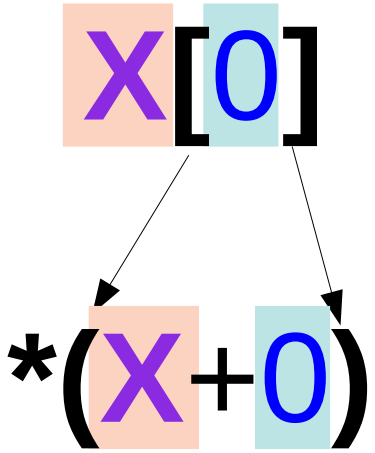


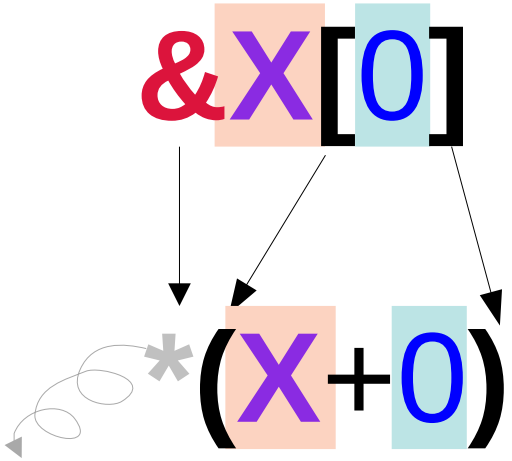


Equivalences with **i = 0**

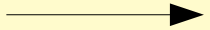
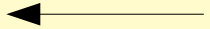
p[0]	⟺	*p
p[i][0]	⟺	*p[i]
p[i][j][0]	⟺	*p[i][j]

&p[0]	⟺	p
&p[i][0]	⟺	p[i]
&p[i][j][0]	⟺	p[i][j]





Operator Precedence

Precedence	Operator	Description	Associativity
1	++ -- () [] . -> (type){list}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access member access through pointer Compound literal(C99)	Left-to-right (((x[m])[n])[p]) 
2	++ -- + - ! ~ (type) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left *(*(*(*x))) 

https://en.cppreference.com/w/c/language/operator_precedence

Pass by Reference

Variable Scopes

```
int func1 (int a, int b)
{
   int i, int j;
   ...
   ...


   ...
   ...
}
```

i and j's
variable scope

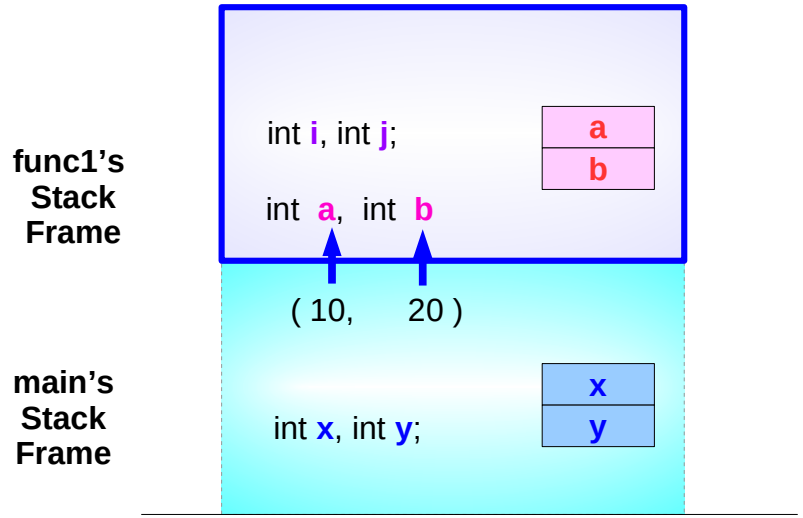
cannot access
each other

```
int main ()
{
   int x, int y;
   ...
   ...

   func1 ( 10, 20 );

   ...
   ...
}
```

x and y's
variable scope

Only **top** stack frame is active
and its variable can be accessed

Communications are performed
only through the **parameter** variables



Pass by Reference

```
int func1 (int* a, int* b)
{
   int i, int j;
   ...
   ...


   ...
   ...
}
```

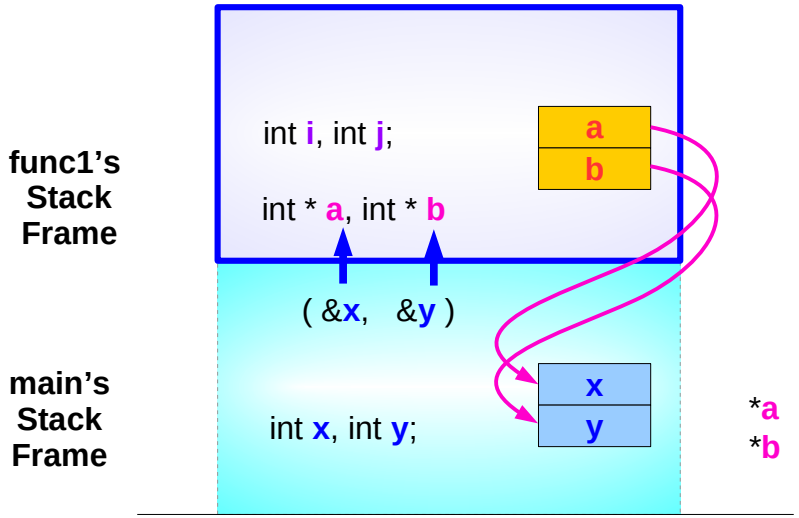
```
int main ()
{
   int x, int y;
   ...
   ...

   func1 ( &x, &y );

   ...
   ...
}
```

x and y are made known to **func1**
func1 can read / write x and y
through their addresses

*a
*b

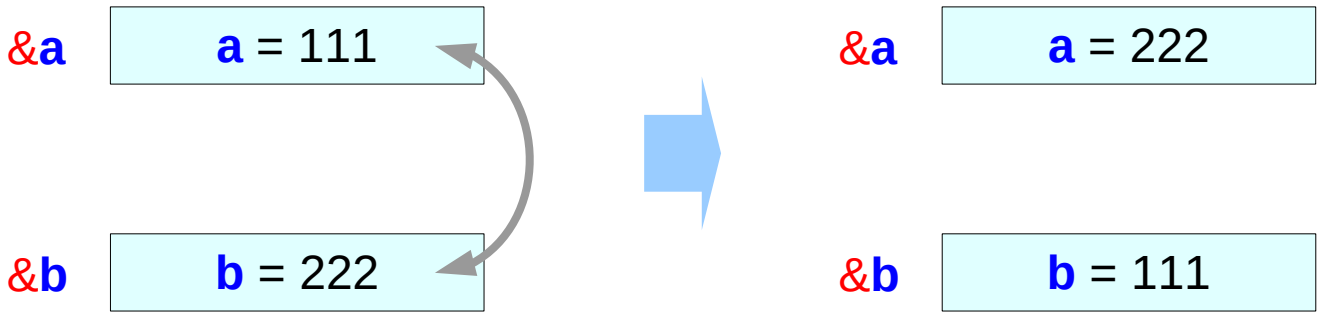
a = &x
b = &y

x and y's
variable scope



Example :
swapping integers
swapping pointers

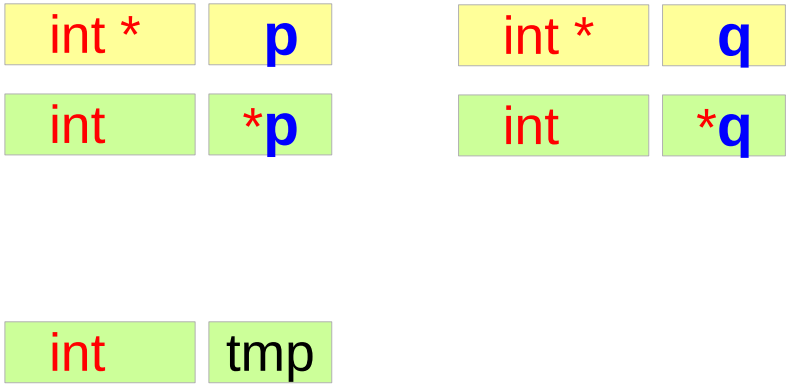
Swapping integers




```
int a, b;

swap( &a, &b );        function call

swap( int *, int * );  function prototype
```

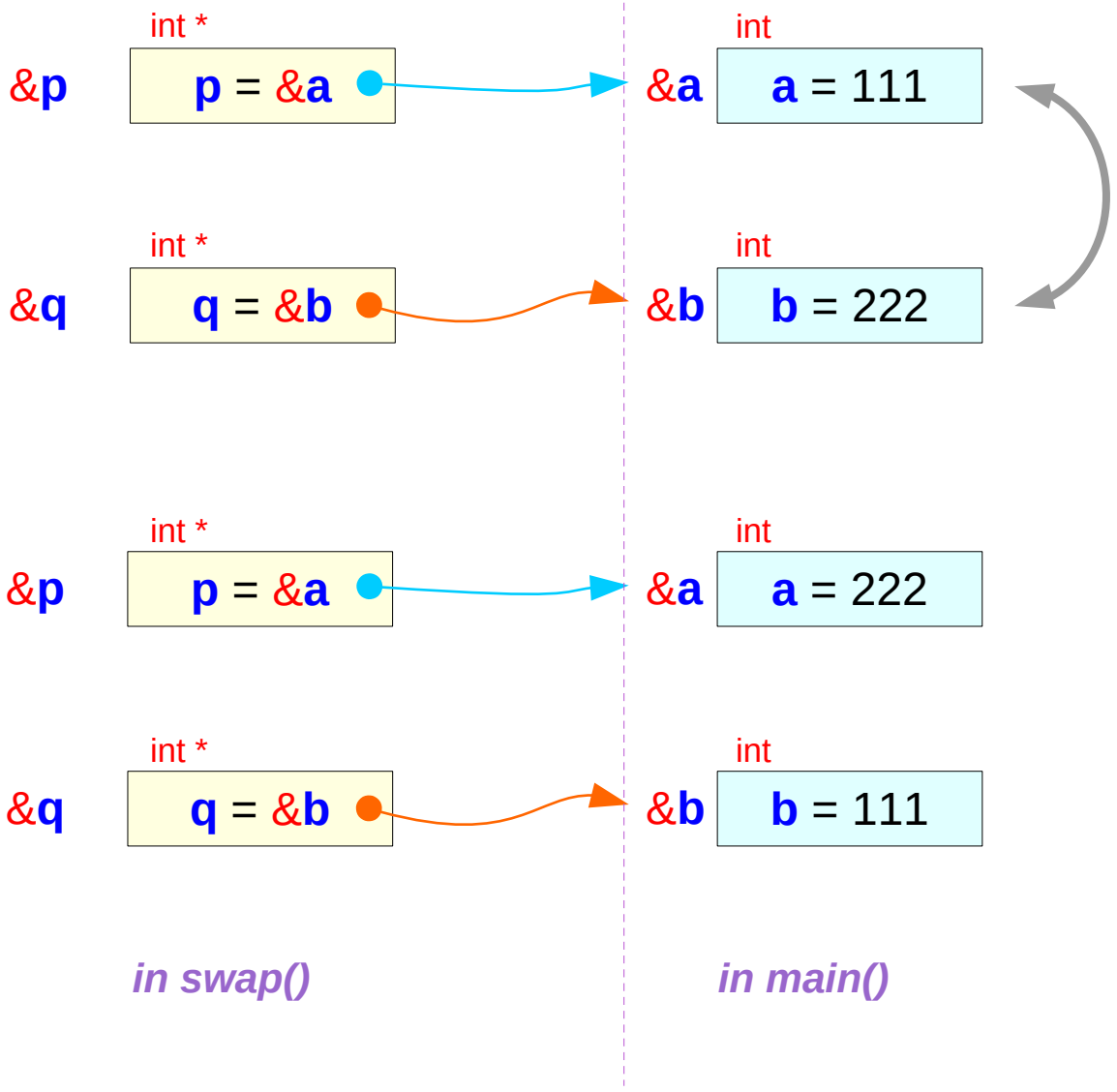
Pass by integer reference

```
void swap(int *p, int *q) {
    int tmp;

    tmp = *p;
    *p  = *q;
    *q  = tmp;
}
```




```
int a = 111, b = 222;

    …
swap( &a, &b );
```

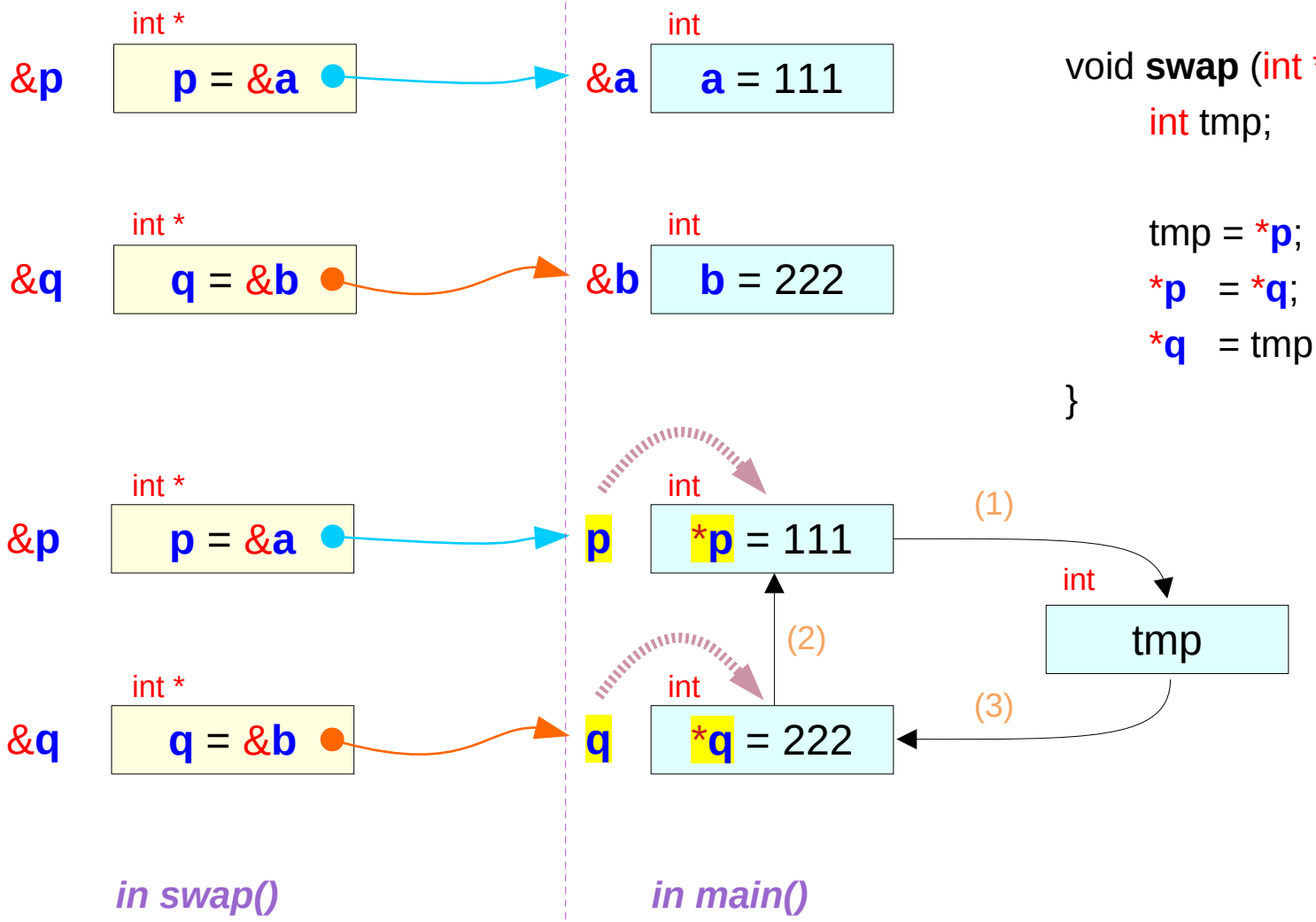
Swapping integers



before

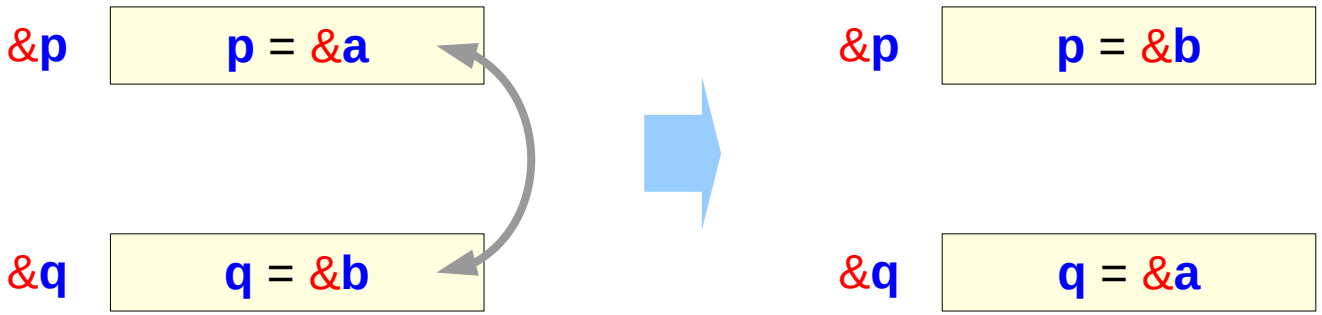
after

Swapping integers via integer pointers




```
void swap (int *p, int *q) {
        int tmp;

        tmp = *p;       (1)
        *p  = *q;       (2)
        *q  = tmp;      (3)
}
```

Swapping integer pointers




```
int    *p,    *q  ;

pswap (   &p,    &q );          function call

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

Pass by integer pointer reference

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

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

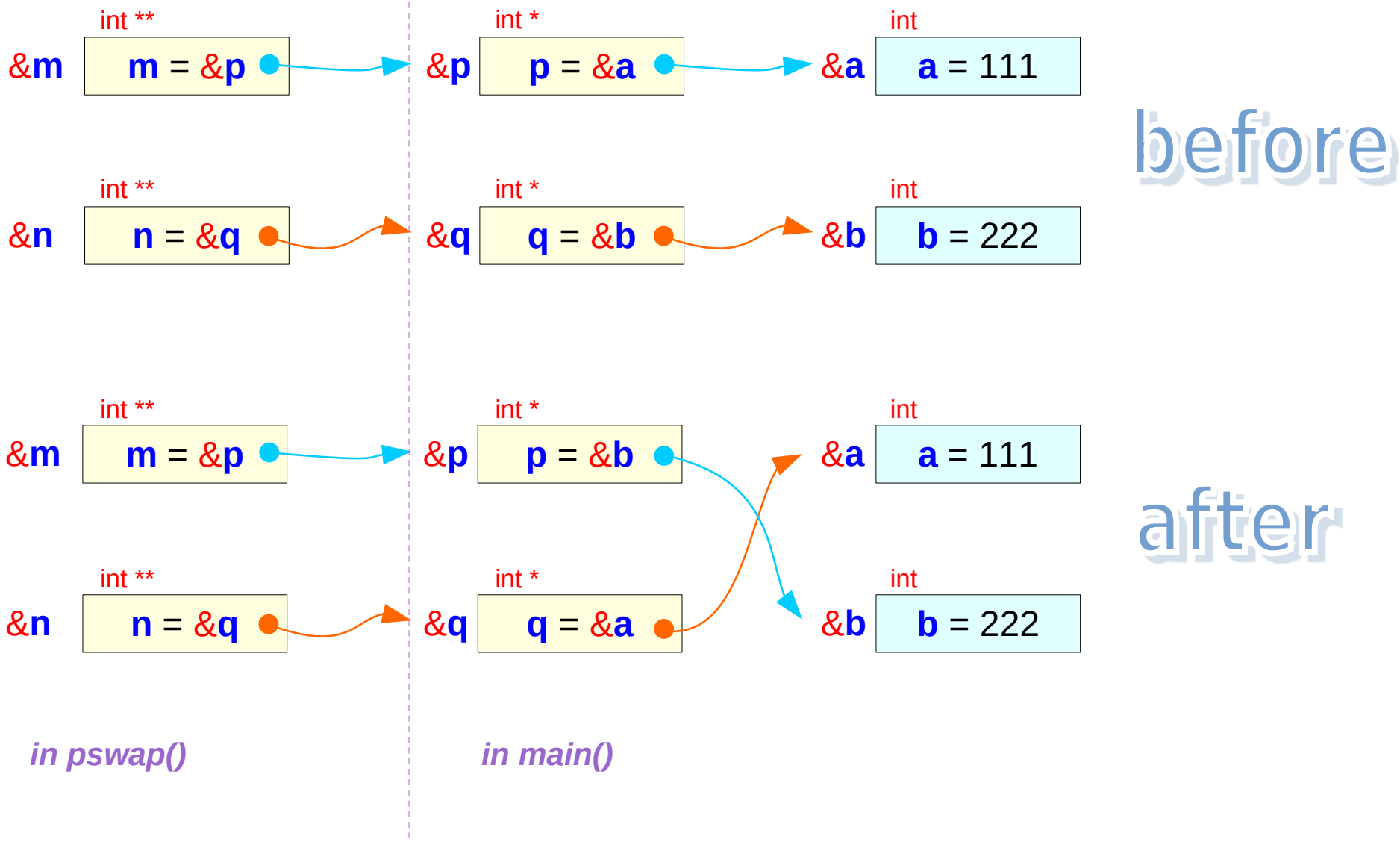
int **	m	int **	n
int *	*m	int *	*n
int	**m	int	**n

int *	tmp

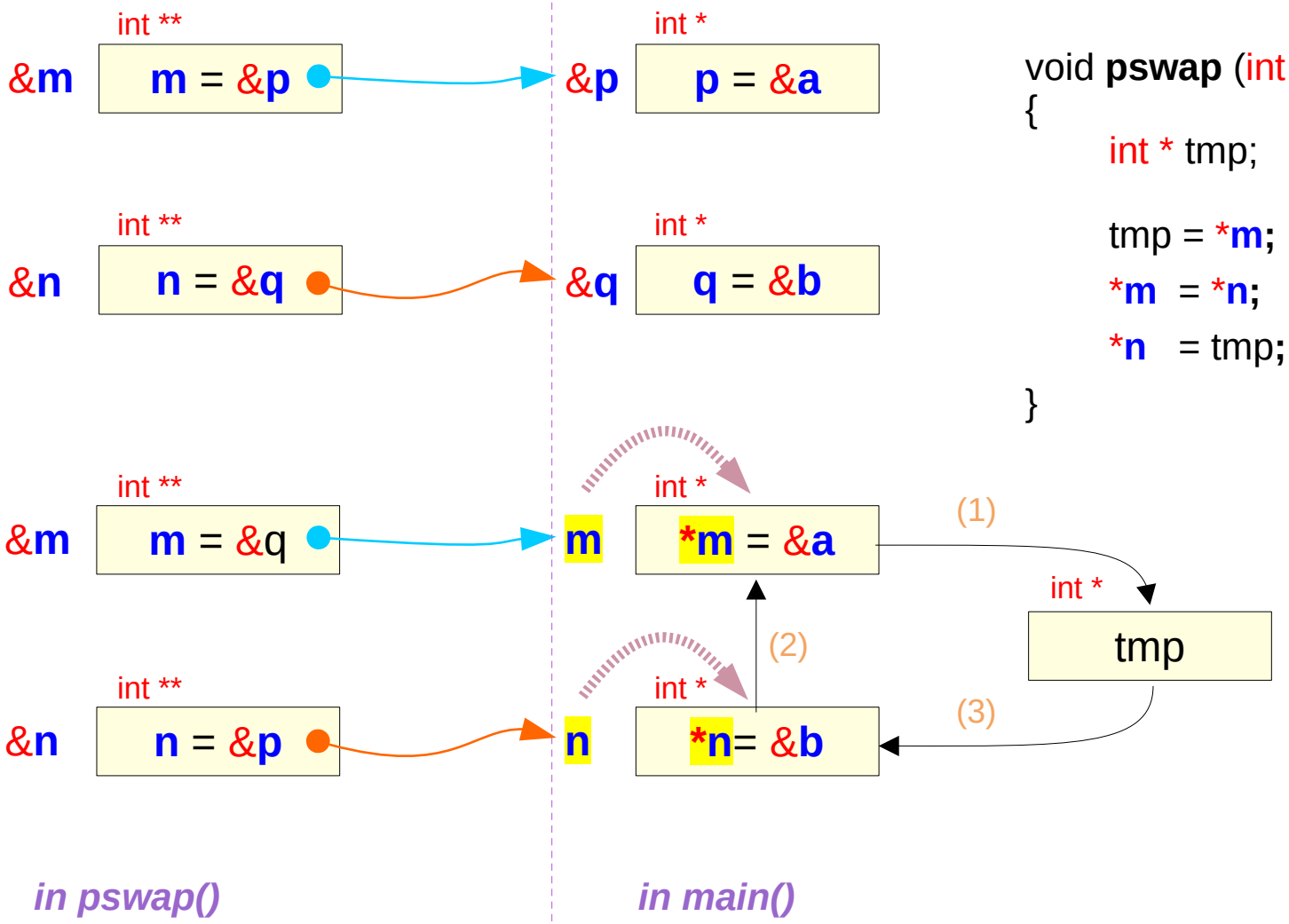
```
int    a = 111,  b = 222;
int  *p = &a, *q = &b;
       …
pswap ( &p, &q );
```

int **	m
int *	*m
int	**m

Swapping integer pointers



Swapping integer pointers via double pointers




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

        tmp = *m;       (1)
        *m  = *n;       (2)
        *n   = tmp;     (3)
}
```

Pointer Type Cast

Assigning pointer variables

long a ; &a address of a long value

address of an int value address of a long value

int * p ; p &a

address of a short value address of a long value

short * q ; q &a

address of a char value address of a long value

char * r ; r &a

Type cast pointers

```
long    a;
```

address of a long value `&a`

```
int     *   p ;
```

address of an int value `p`

int address conversion `(int *) &a`

```
short   *   q ;
```

address of a short value `q`

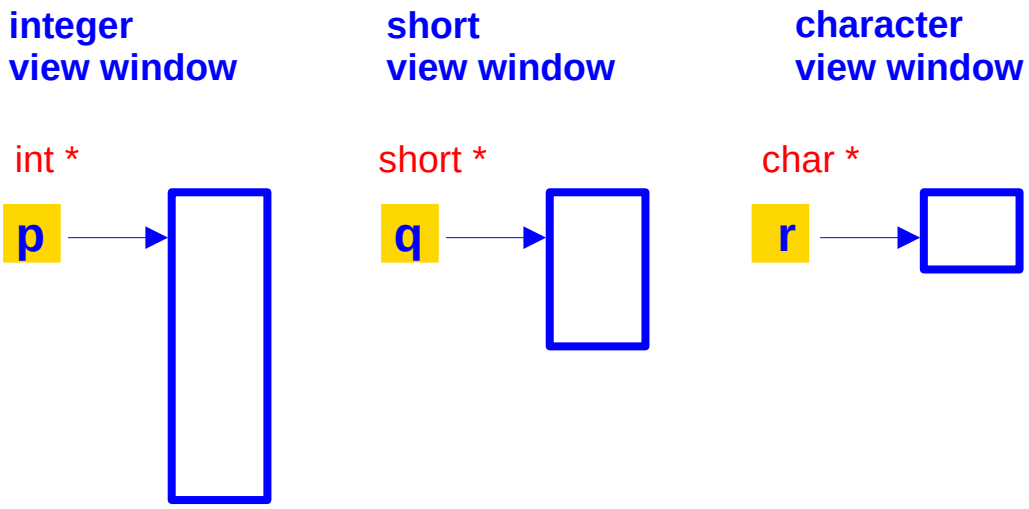
short address conversion `(short *) &a`

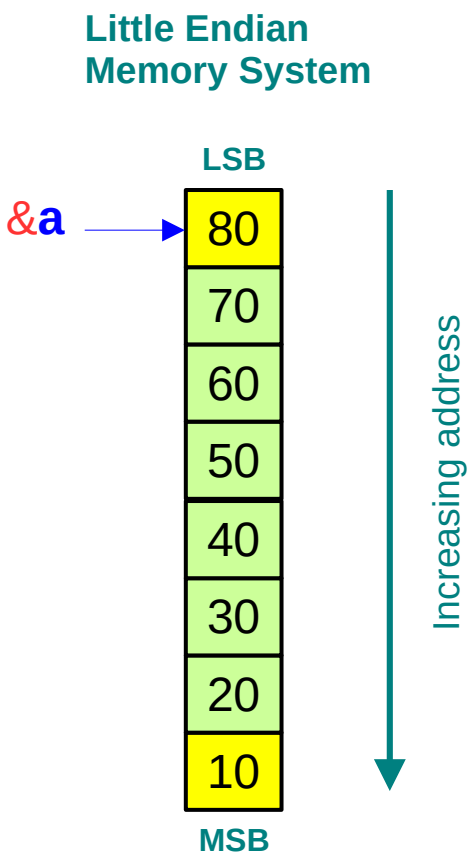
```
char    *   r ;
```

address of a char value `r`

char address conversion `(char *) &a`

View windows – type cast pointer

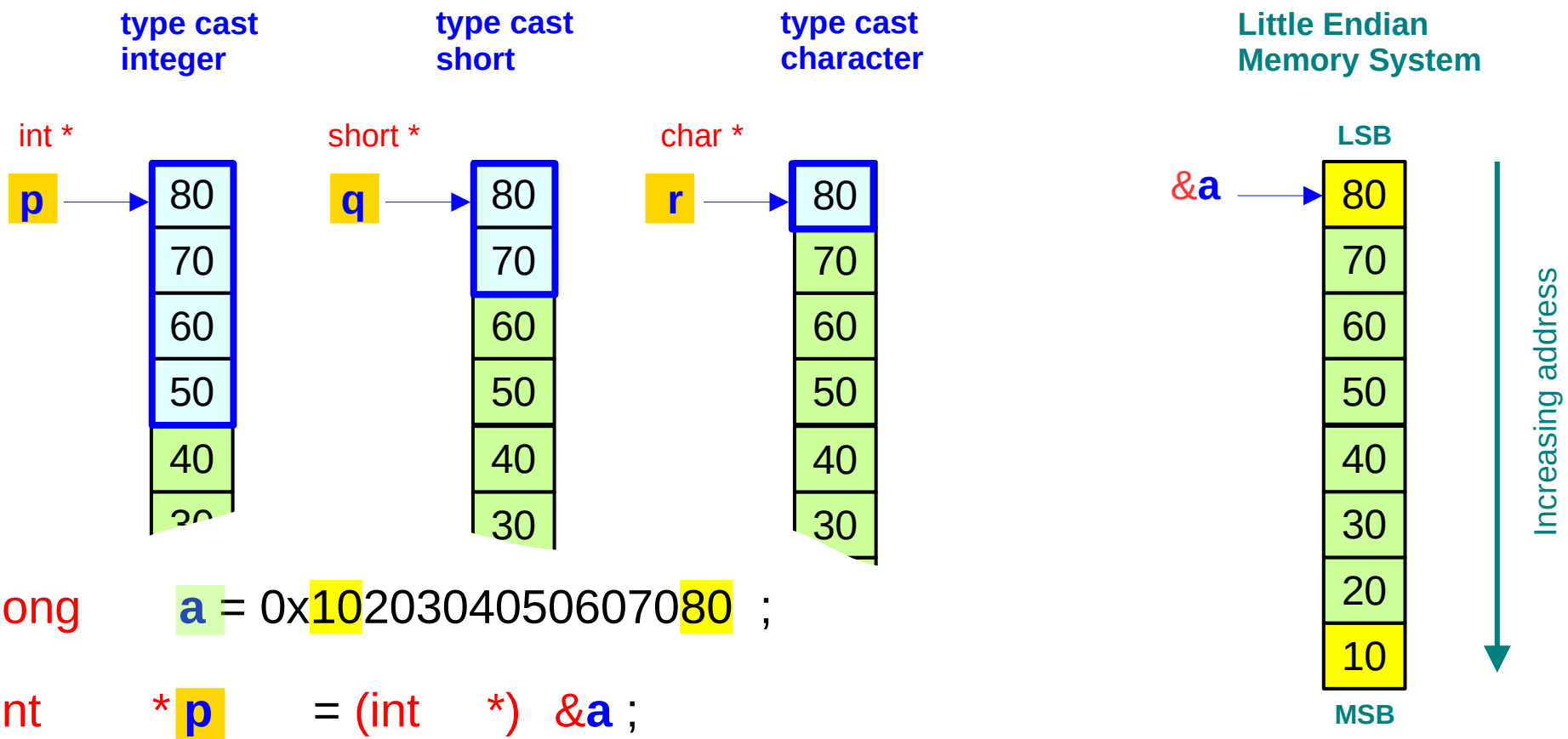




MSB LSB

```
long    a = 0x1020304050607080 ;

int     * p  ;
short   * q  ;
char    * r  ;
```

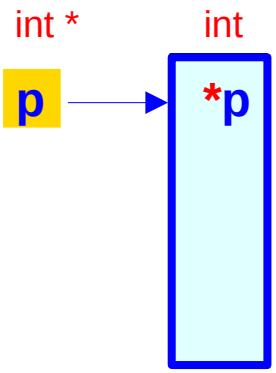
Applying type cast pointers to a memory location



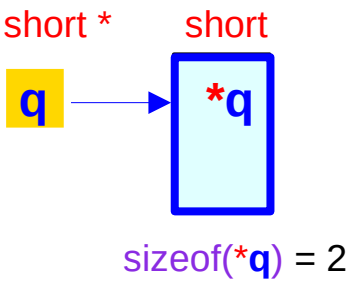

```
long    a = 0x10203040506070 80 ;

int     * p     = (int   *)  &a ;
short   * q     = (short *)  &a ;
char    * r     = (char  *)  &a ;
```

De-referencing type cast pointers

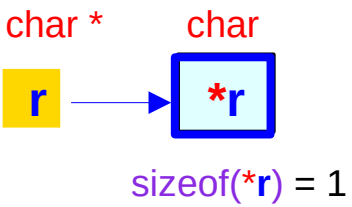
type cast integer pointer



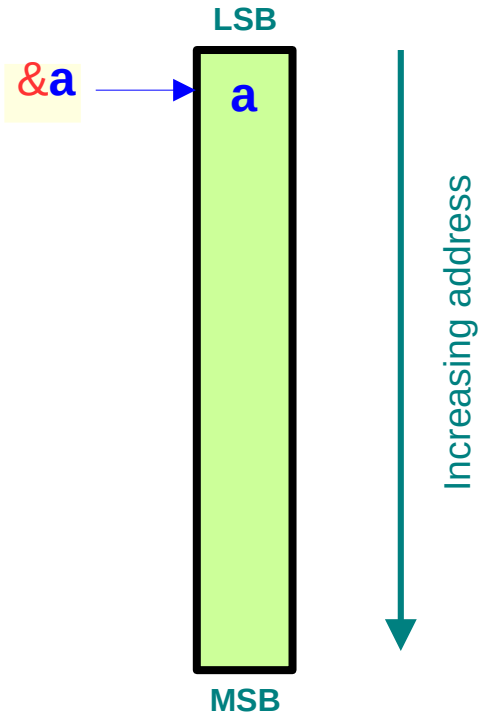
type cast short pointer



type cast character pointer



Little Endian Memory System

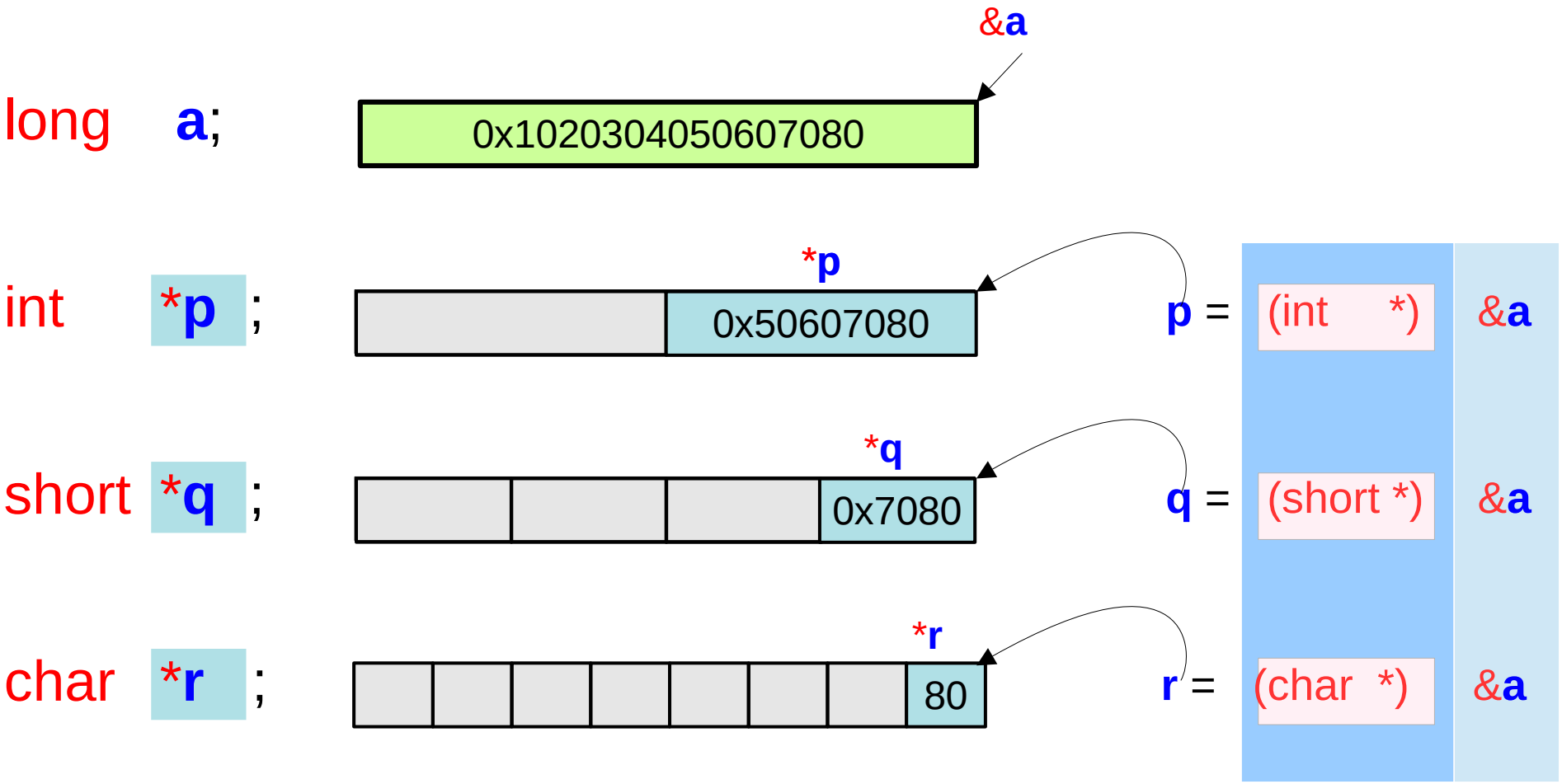



```
long    a = 0x10203040506070 80 ;
```

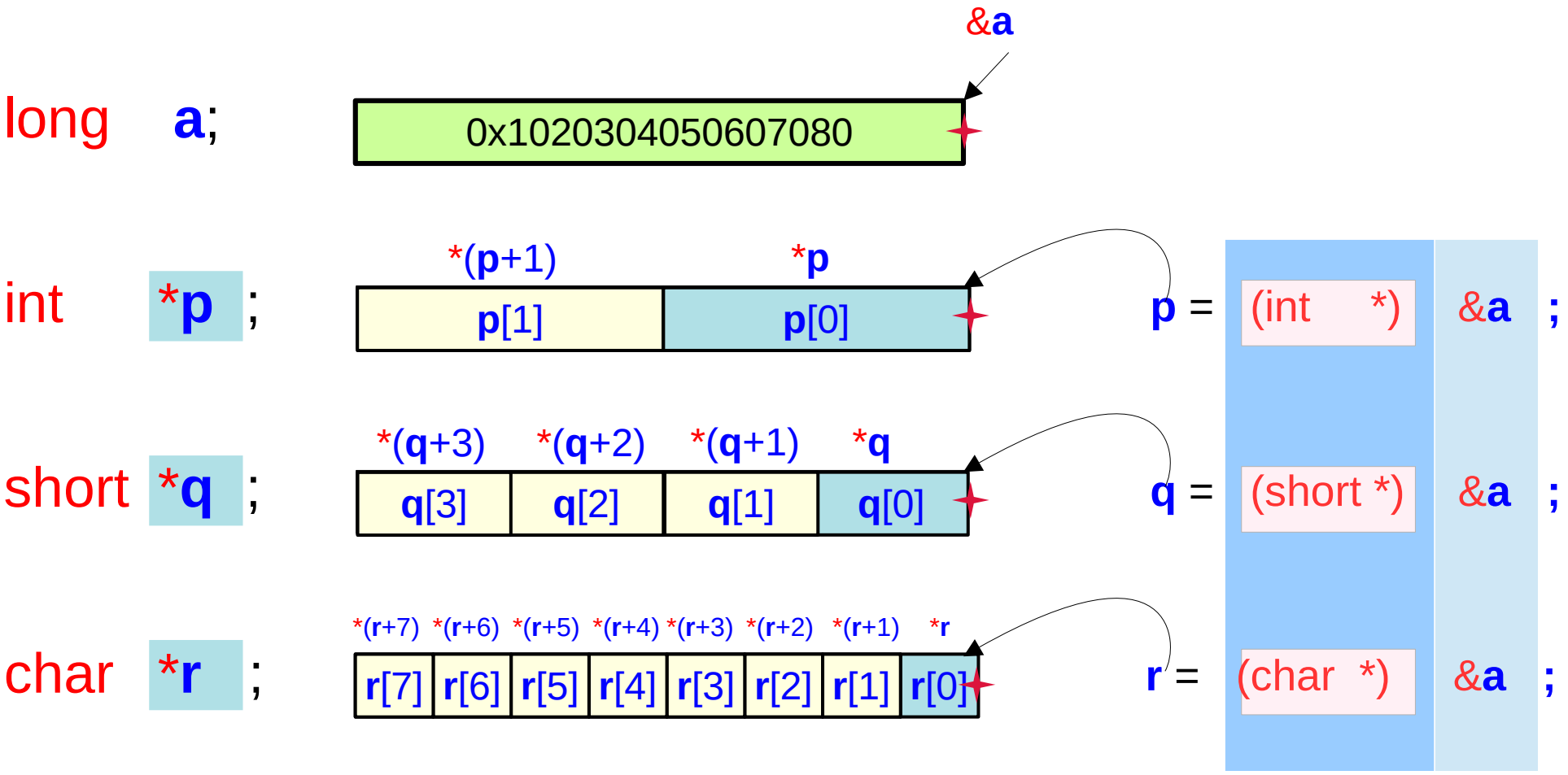
```
int     *p = (int   *) &a ;
short   *q = (short *) &a ;
char    *r = (char  *) &a ;
```

```
*p  ≡ 0x5060708 0
*q  ≡ 0x708 0
*r  ≡ 0x 80
```

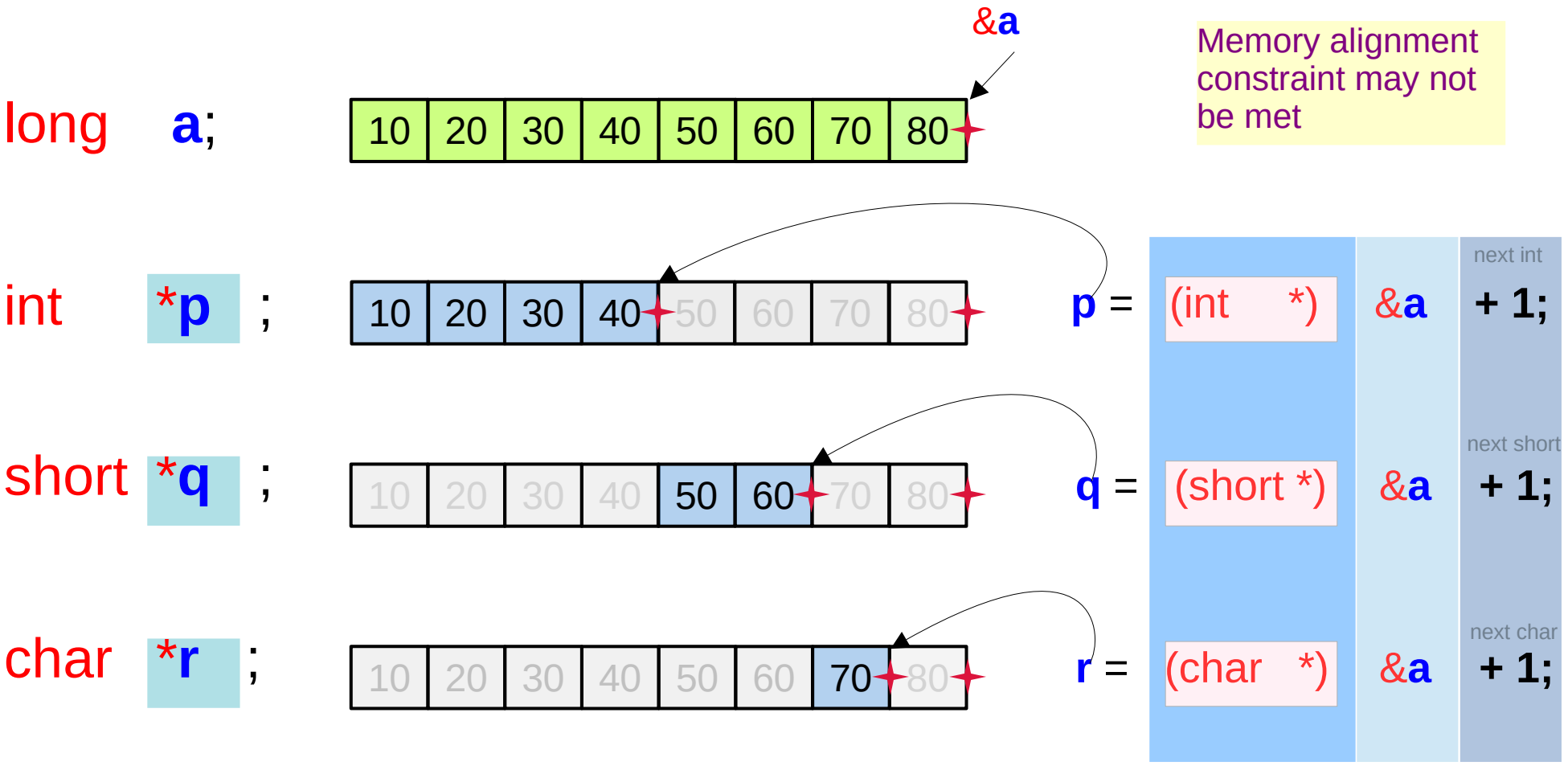
Re-interpretation of memory data – case I



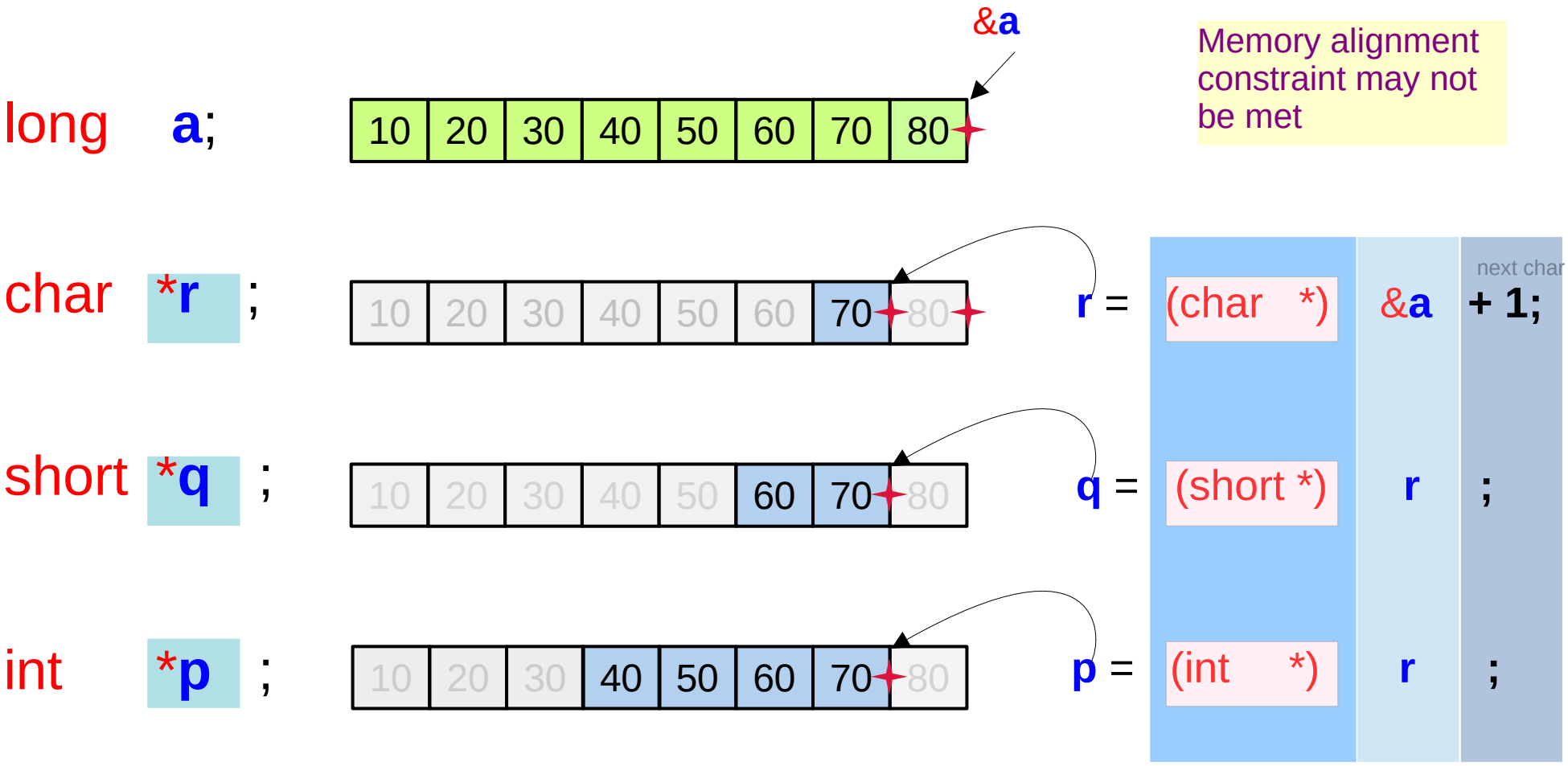
Re-interpretation of memory data – case I



Re-interpretation of memory data – case II



Re-interpretation of memory data – case III



const pointers

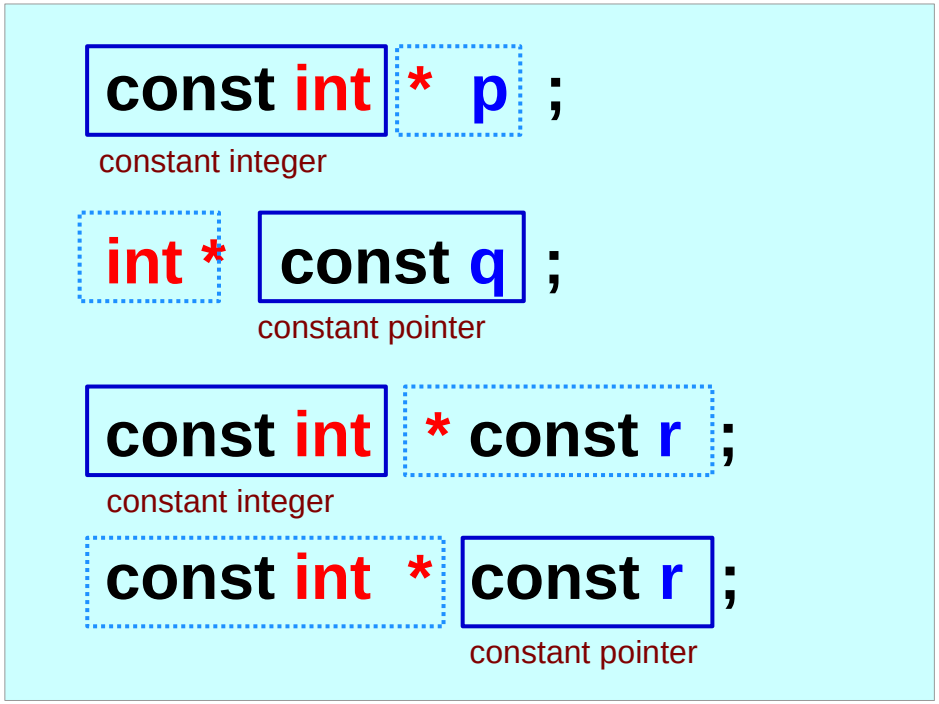
const type, const pointer type (1)

```
const int  *  p;          <=>   int  *  p;
int * const q  ;          <=>   int  *  q  ;
const int  *  const r  ;  <=>   int  *   r  ;
```

constant

must not be changed
must not be updated
must not be written
must not be assigned

const type, const pointer type (2)

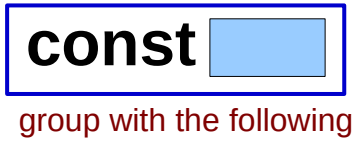


*p : constant integer value

q : constant (int *) pointer

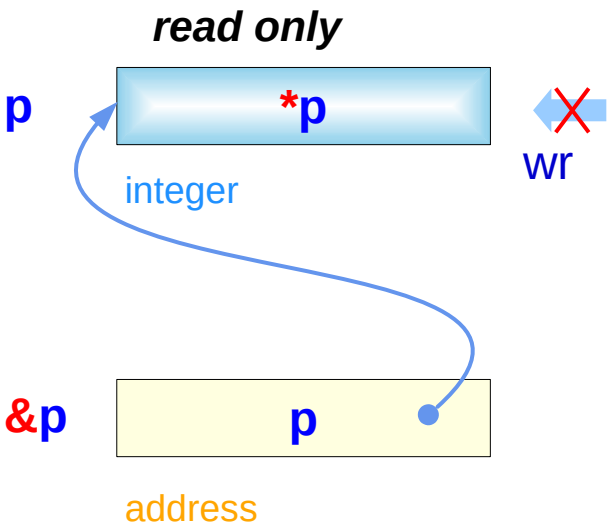
*r : constant integer value

r : constant (int *) pointer

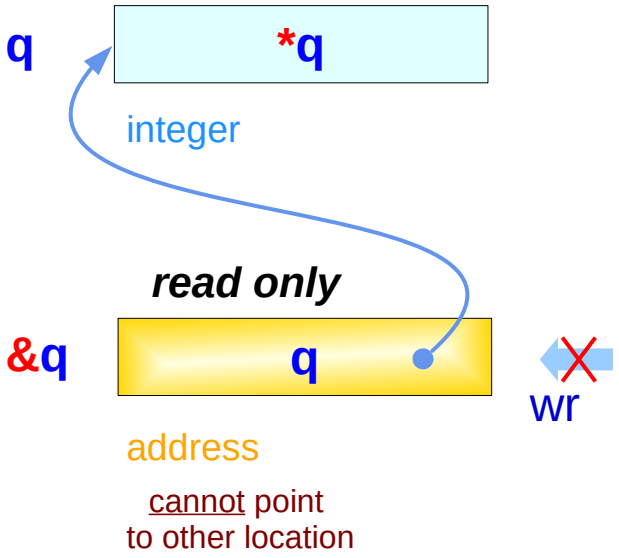


const type, const pointer type (3)

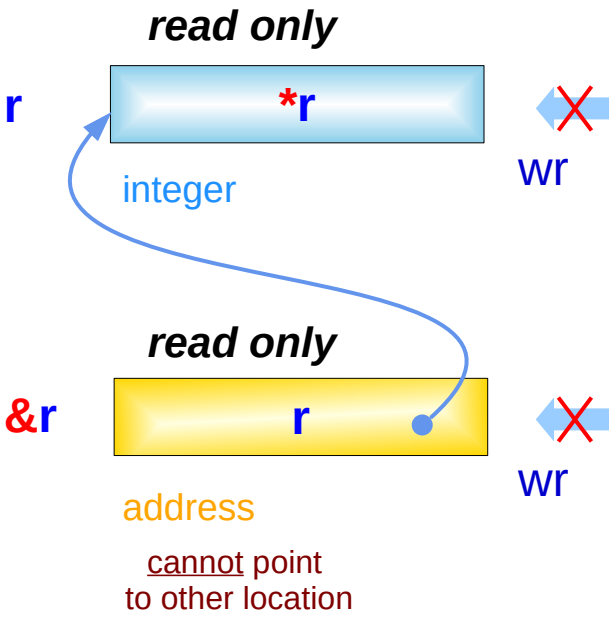
const int *p;



int * const q ;



const int * const r ;



const examples (1)

```
const int * pc;
      int * p, i;
const int   ic;
```

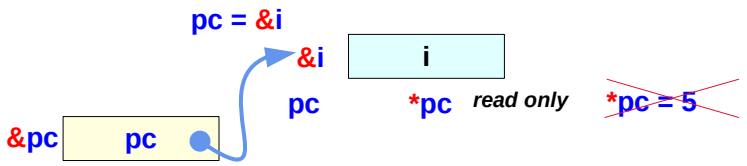
```
  pc = &i;      // (const int *) ← (int *)
*pc  = 5;       // (const int) error
```

Writing to the writable memory location (**i**)
is forbidden via **pc** … (no harm, OK)

```
  p   = &ic;    // (int *) ← (const int *) warning
*p    = 5;      // (int)
```

Writing to the read only memory location (**ic**)
is not forbidden via **p** … (hazardous, not OK)

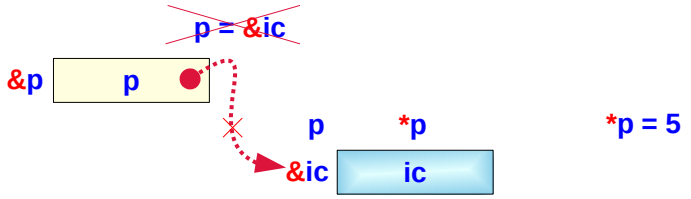
C A Reference Manual, Harbison & Steele Jr.



pc can point to **i**
***pc** must be **const**

the same memory location
that can be written via **i**
cannot be written via ***pc**

***pc** should not write
the writable memory location



Assume **p** points to **const ic**

the same memory location
that cannot be written via **ic,**
can be written via ***p**

thus ***p** can write
the **const** memory location

therefore, **p** should not point to **const ic**

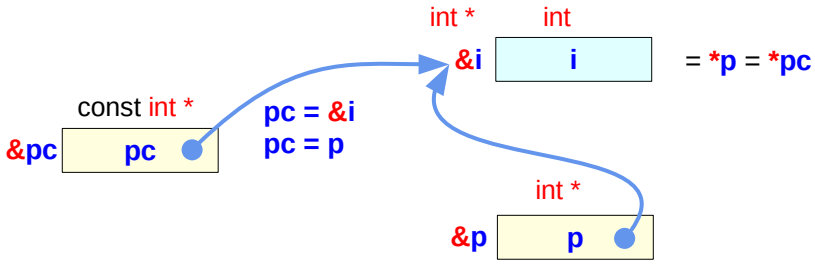
const examples (2)

```
const int * pc;
        int * p, i = 5;
const int   ic = 7;
```

```
p   = &i;
pc  = &ic
```

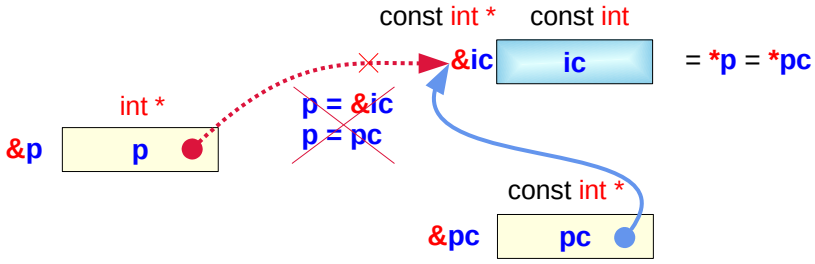
// more constrained type ← general type (O)

```
pc  = &i;     // (const int * ← int *)
pc  =  p;     // (const int * ← int *)
```



// general type ← more constrained type (X)

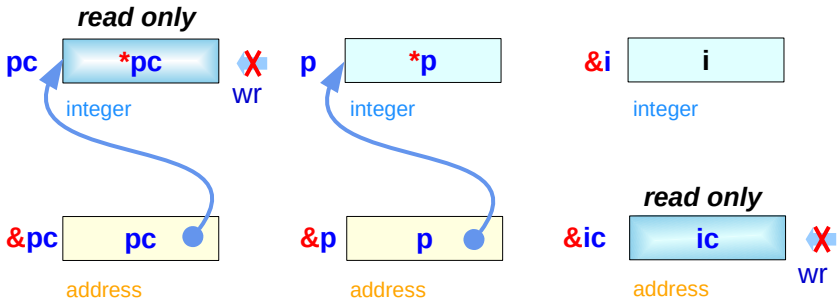
```
p   = &ic;   // (int * ← const int *)  warning
p   =  pc;   // (int * ← const int *)  warning
```



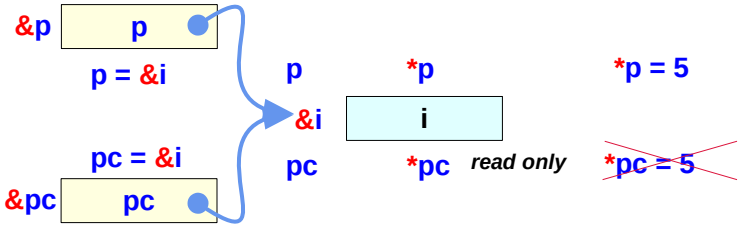
C A Reference Manual, Harbison & Steele Jr.

const examples (3)

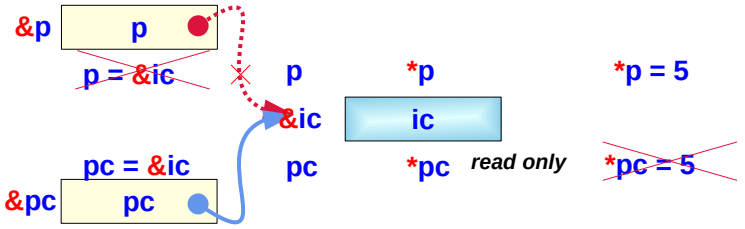
```
const int * pc;
      int * p, i;
const int   ic;
```




```
 p   = &i;     // (int *) ← (int *)
*p   = 5;      // (int)
 pc  = &i;     // (const int *) ← (int *)
*pc  = 5;      // (const int) error
```



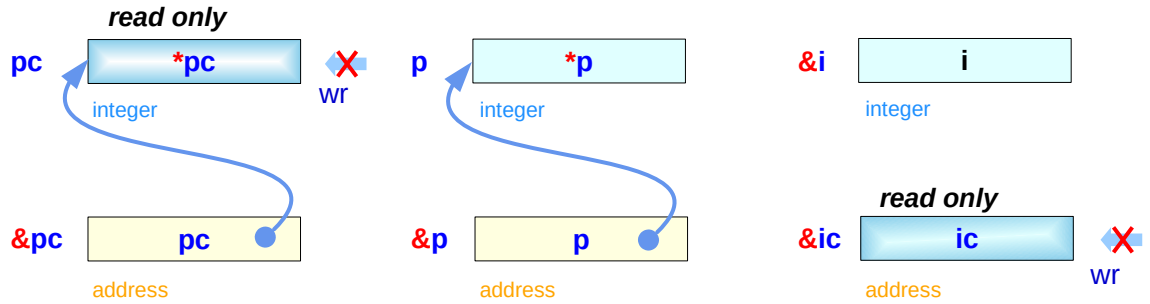

```
 p   = &ic;    // (int *) ← (const int *) warning
*p   = 5;      // (int)
 pc  = &ic;    // (const int *) ← (const int *)
*pc  = 5;      // (const int) error
```



C A Reference Manual, Harbison & Steele Jr.

const examples (4)

```
const int * pc;
      int * p, i;
const int   ic;
```




```
 pc =   p = &i;
 pc = &ic
*p   =  5;
*pc =  5;              // invalid      *pc :: cons int

pc  = &i;              //             (const int * ← int *)
pc  =  p;              //             (const int * ← int *)
p   = &ic;             // invalid     (int * ← const int *)
p   =  pc;             // invalid     (int * ← const int *)
p   = (int *) &ic;     // type cast
p   = (int *)  pc;     // type cast
```

C A Reference Manual, Harbison & Steele Jr.

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