Pointers (1A)

Young Won Lim
5/24/24

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 data

type : int
size : 4 bytes
value : integer

```
a = 100 ;
```

a holds the integer 100

address	data
&a	a ← 100

Pointer Variables

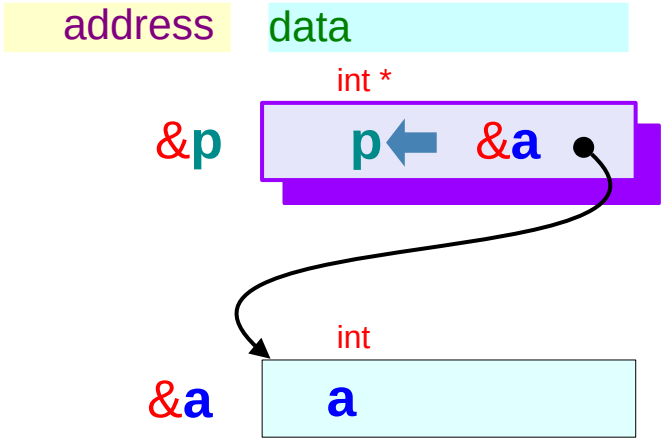
```
int  *  p ;
```

p can hold an address of an integer data

type : int *
size : 4 bytes (32-bit system)
: 8 bytes (64-bit system)
value : address

```
p = &a ;
```

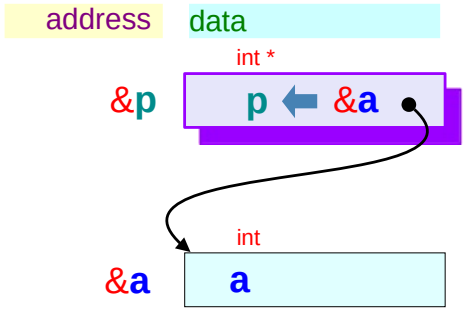
p can hold an address (**&a**) of an integer data (**a**)



Dereferenced Pointer variables

```
int * p ;
```

p can hold an address of an integer data



type *variable*

```
int * p ;
```

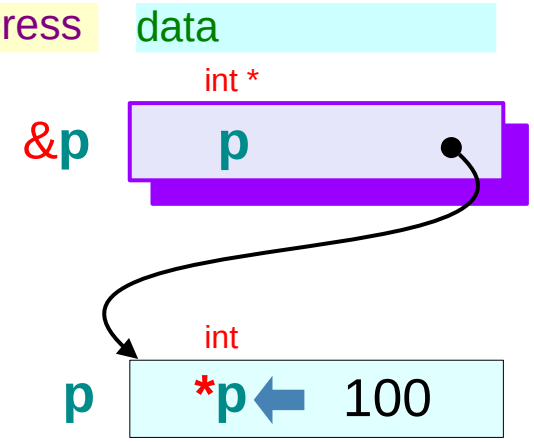
p holds the address

pointer to int

```
int * p ;
```

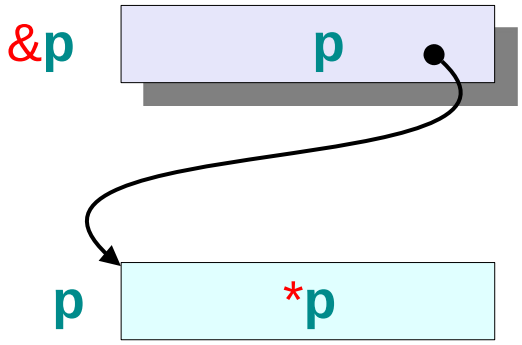
***p** holds an integer data

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***

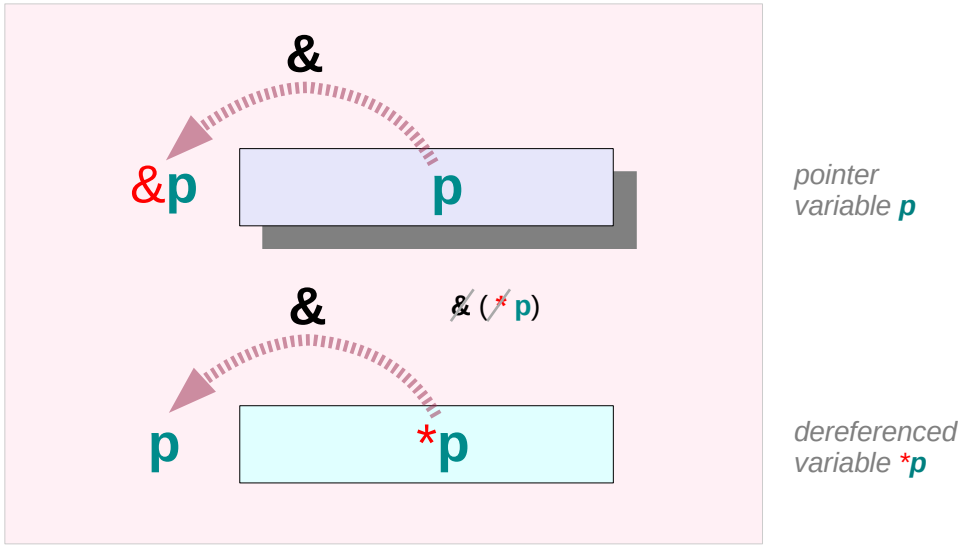
*dereferenced variable ***p***

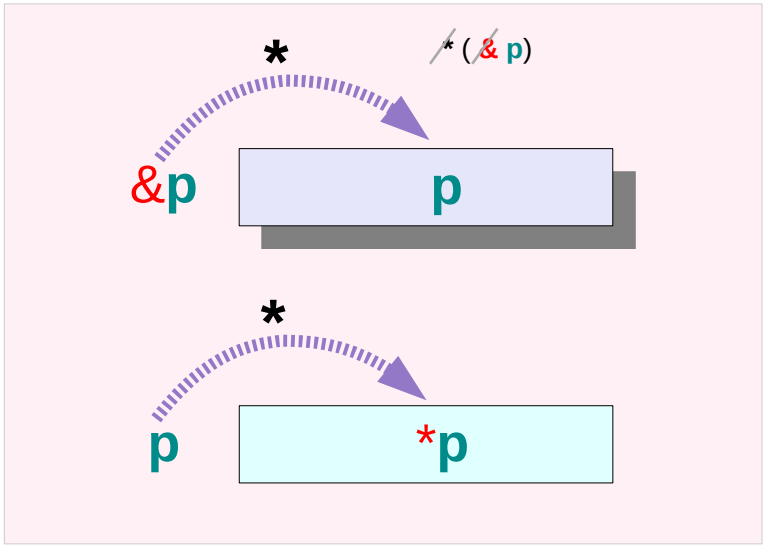
p* and **p are variables*
*&**p** is an address value*

Address-of operator and dereferencing operator (1)

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

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





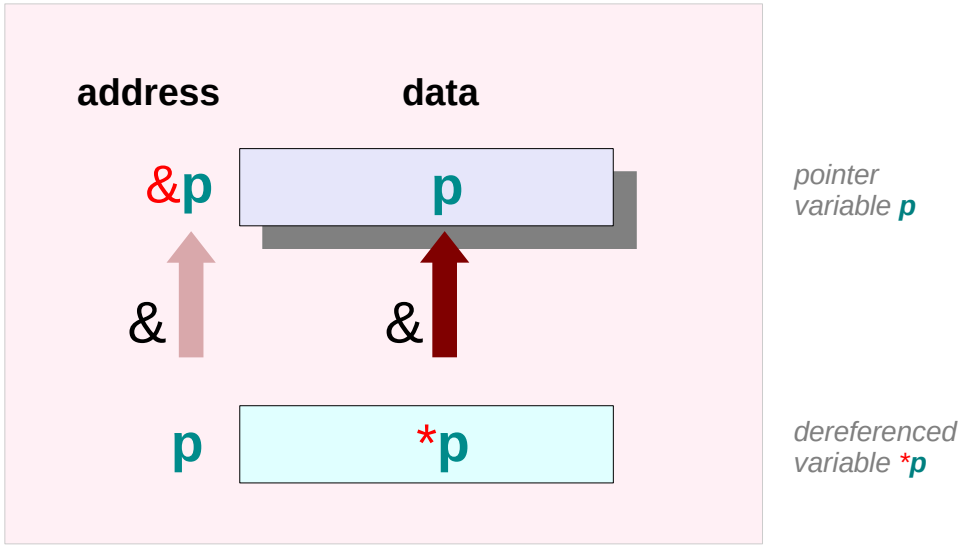
& (**p**) ≡ value(&**p**) … *value*
& (***p**) ≡ value(**p**) … *value*

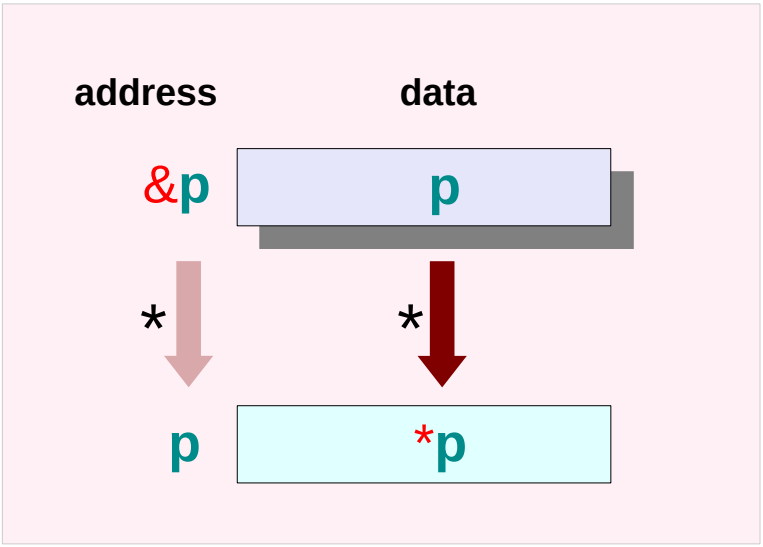
* (&**p**) ≡ **p** … *variable*
* (**p**) ≡ ***p** … *variable*

Address-of operator and dereferencing operator (2)

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

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



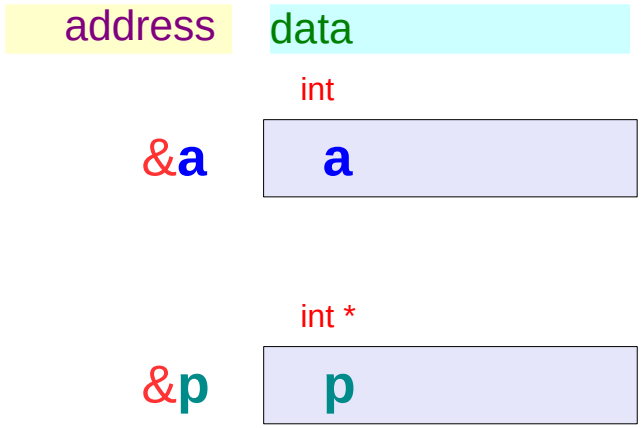


& (**p**) ≡ value(**&p**) … *value*
& (***p**) ≡ value(**p**) … *value*

***** (**&p**) ≡ **p** … *variable*
***** (**p**) ≡ ***p** … *variable*

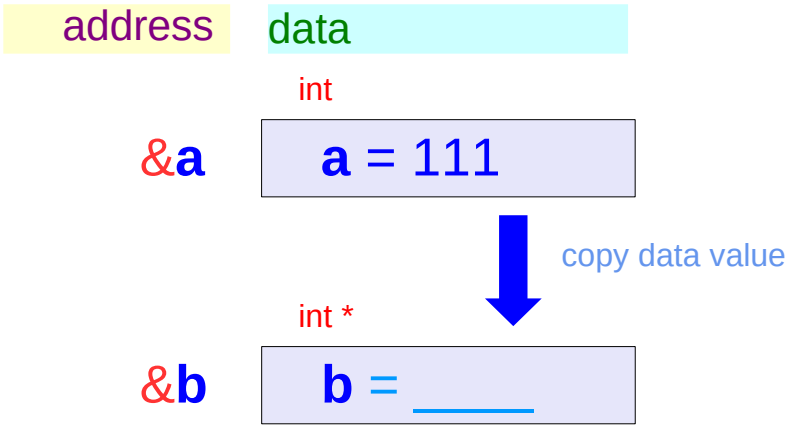
Variables and their addresses

```
int     a ;

int   * p ;
```



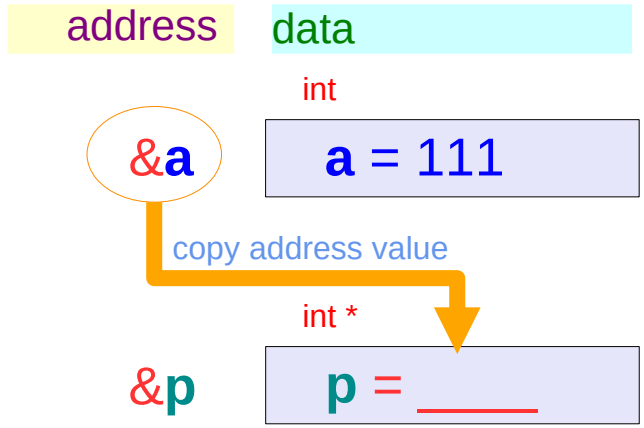
Assignment of a value

```
int     a ;

int     b ;

        b = a ;
```



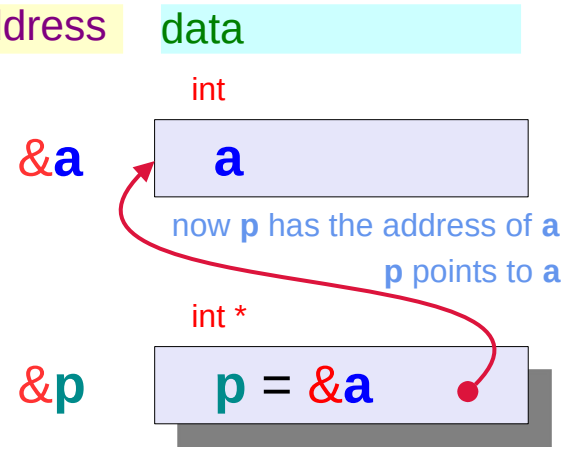
Assignment of an address value

```
int     a ;

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



Arrow notations

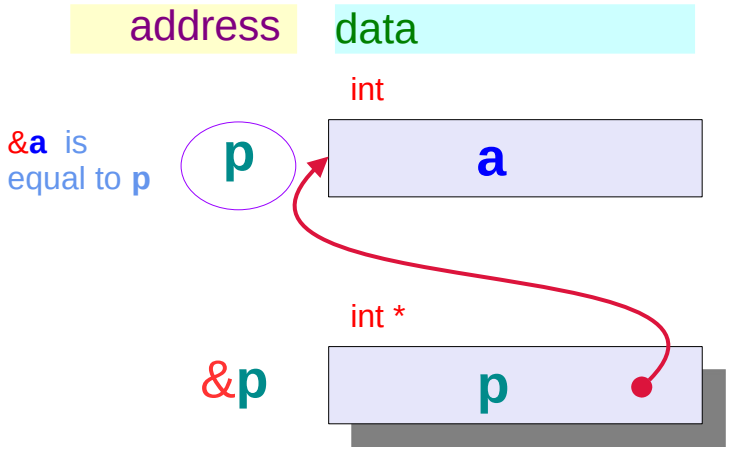
```
int     a ;

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



Pointed address : **p**

```
int     a;

int  *  p = &a;
```

p = &a;

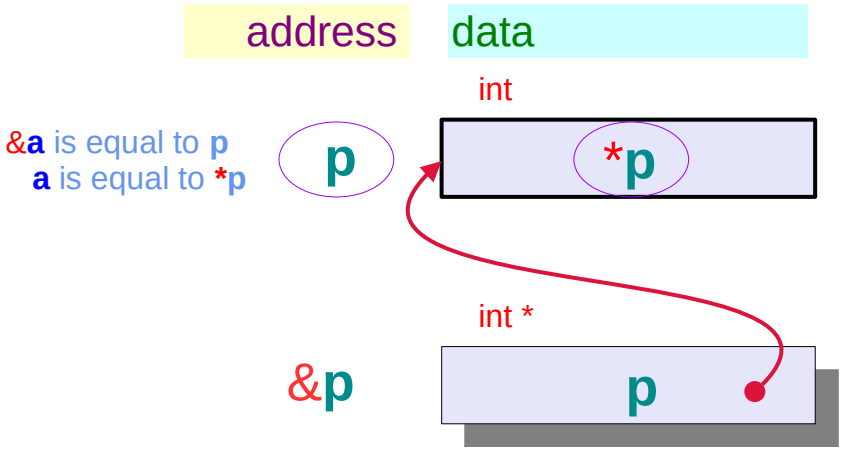


p ≡ &a

Dereferenced variable : *p

```
int     a;

int  *  p = &a;
```

p = &a;



equivalence

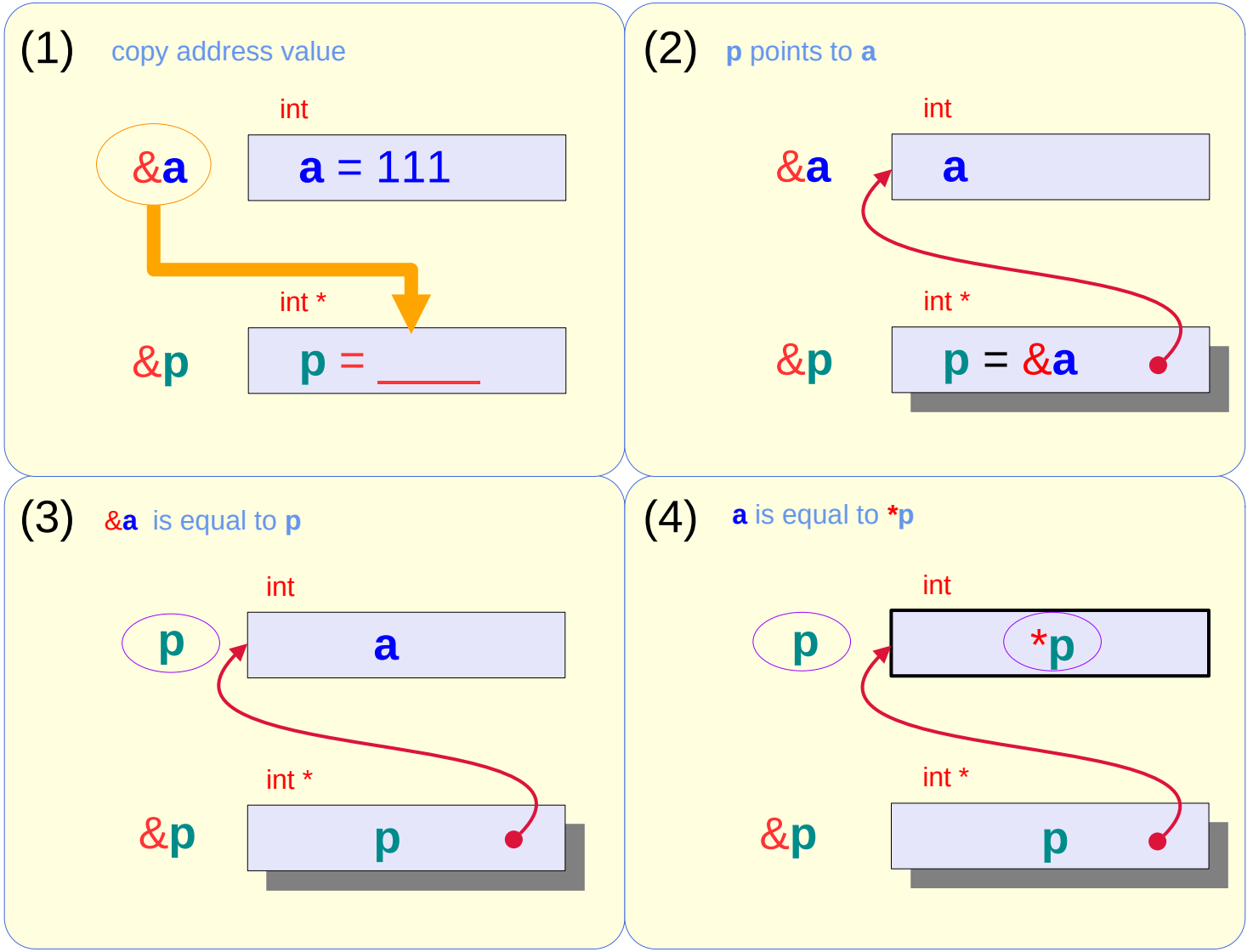
p ≡ &a

* (p) ≡ * (&a)

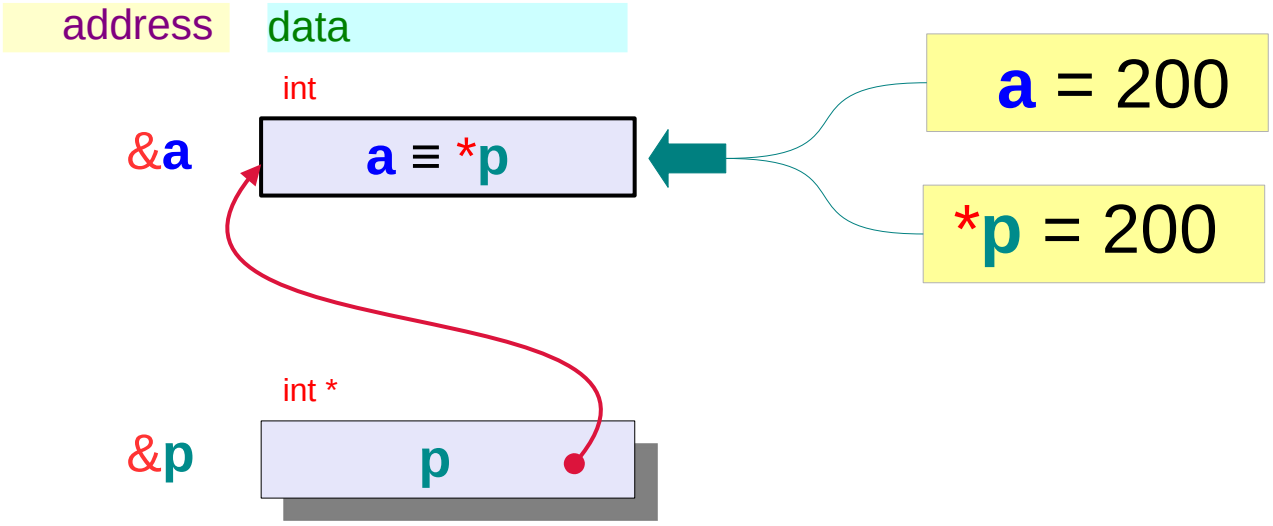
*p ≡ a

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 data

```
int *  *q ;
```

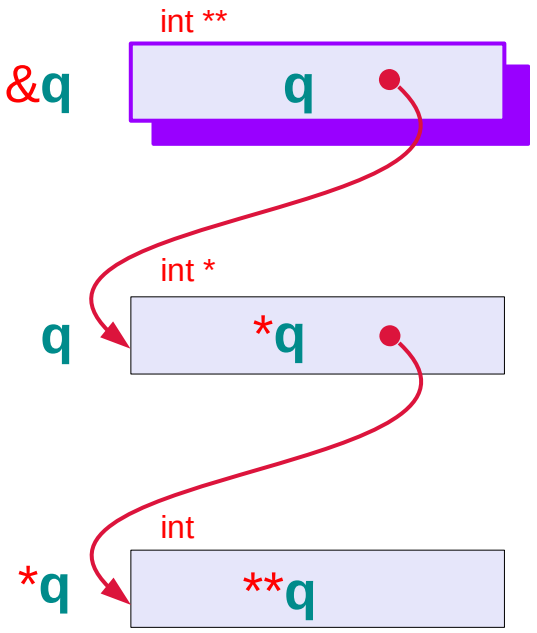
a pointer to
an integer

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

```
int  * *q ;
```

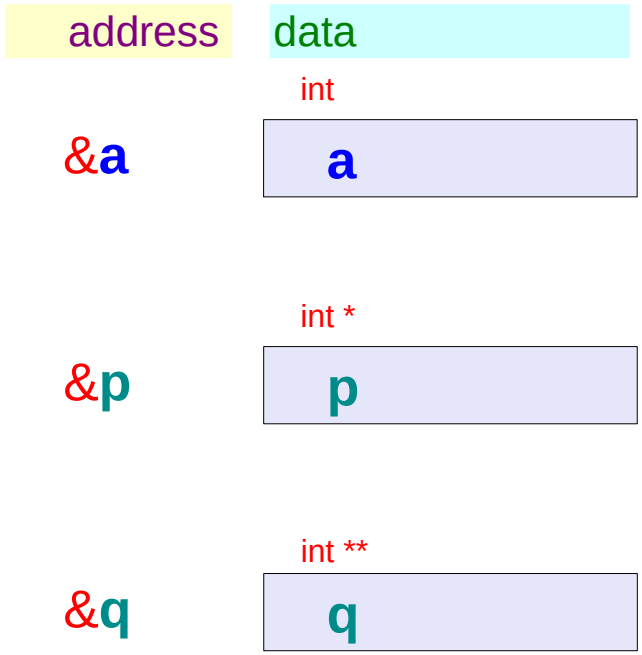
an integer

****q** holds an integer data



Variables and their addresses

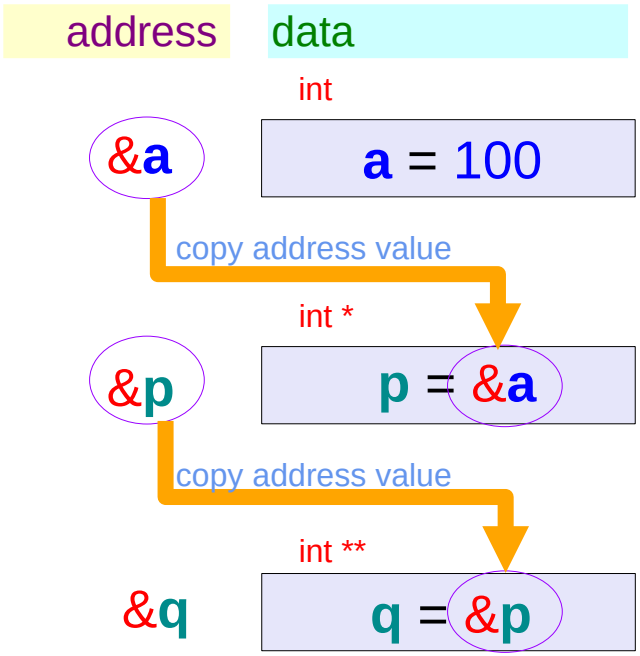
```
int     a ;
int   * p ;
int   ** q ;
```



Assignments of address values

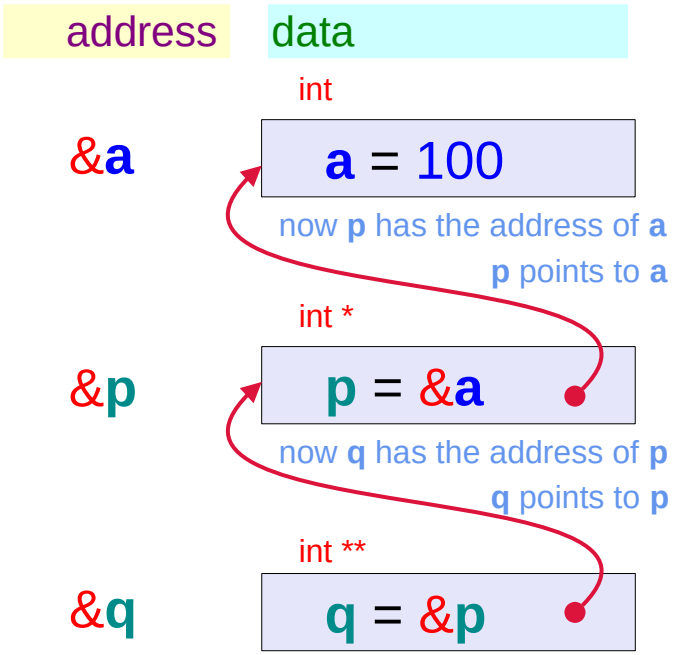
```
int     a = 100 ;
int   * p = &a ;
int   ** q = &p ;
```

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



Arrow notations

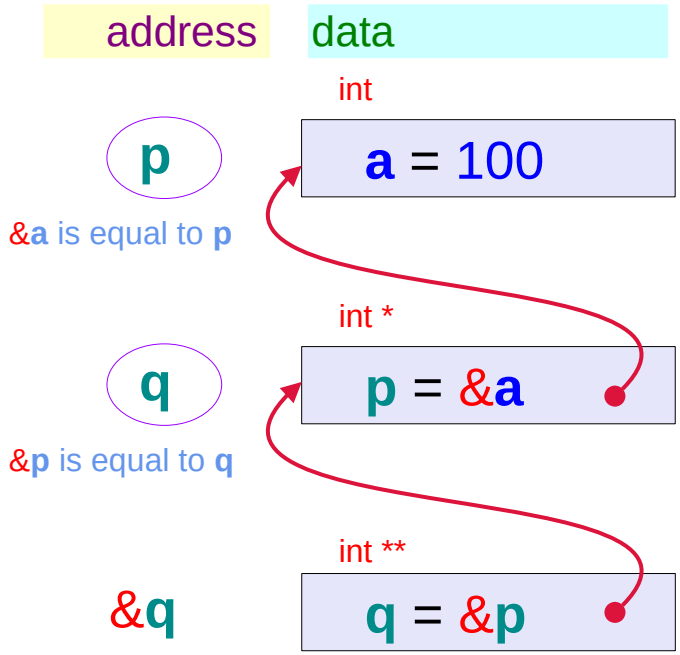
```
int     a = 100 ;

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

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



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

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

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

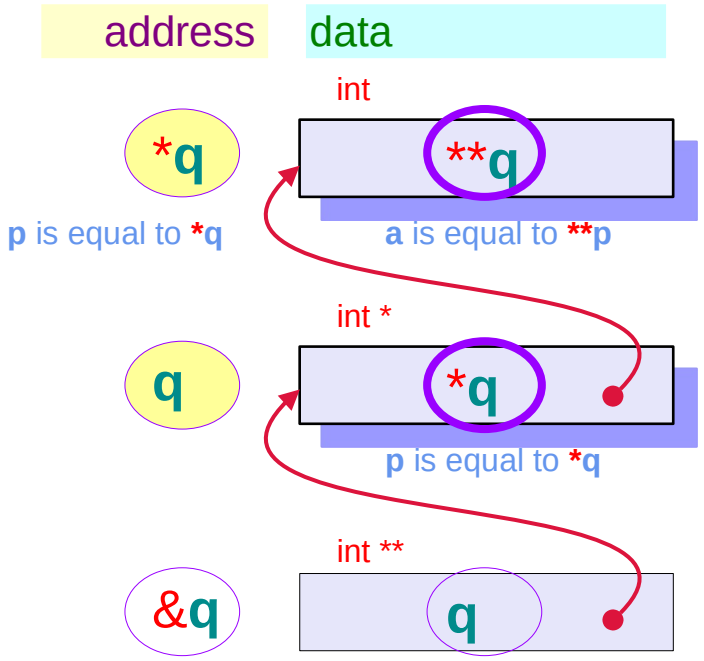


Dereferenced variables : *q, **q

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

**q ≡ a

*q ≡ p

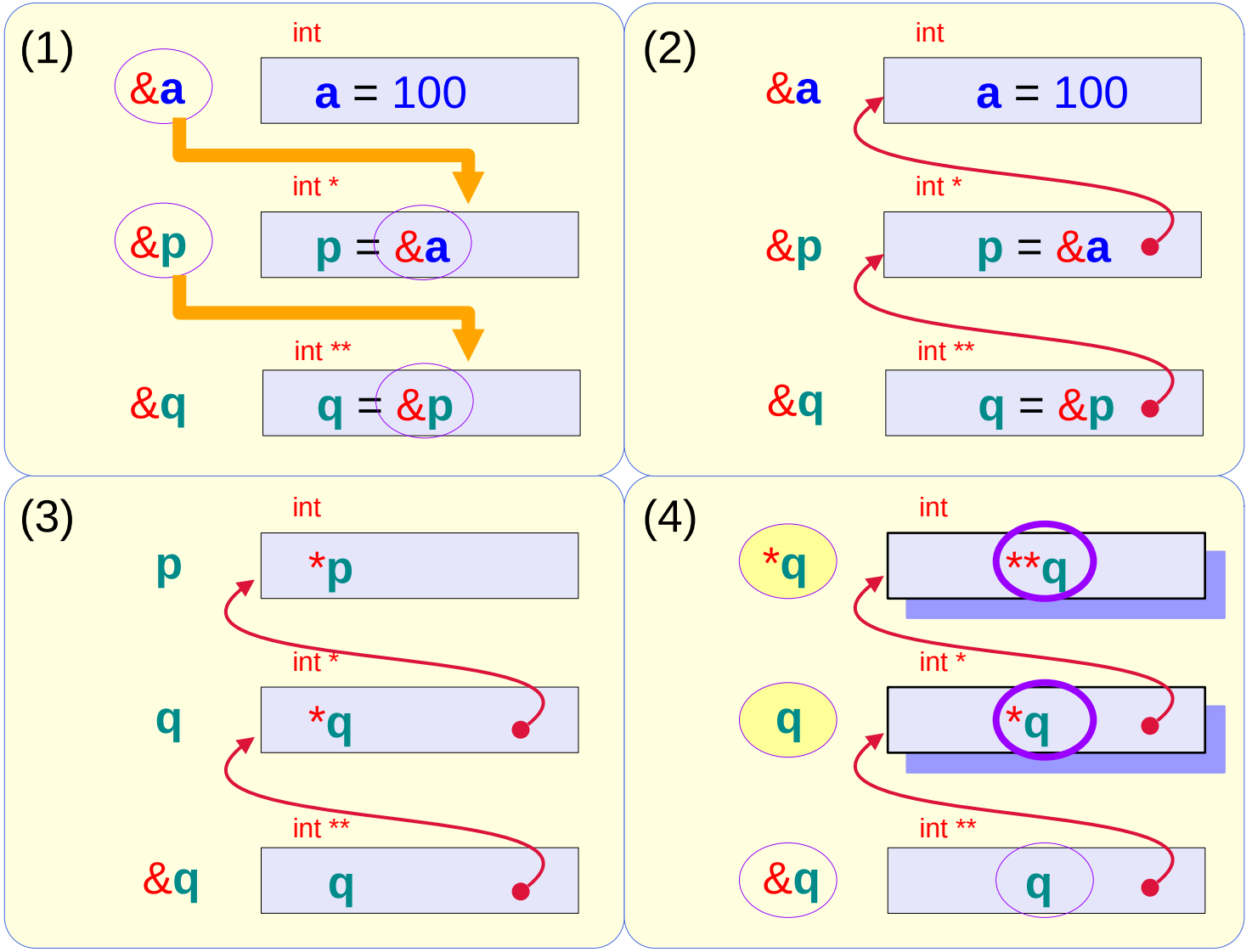


Variable **p**, ***p**, and **q**, ***q**, ****q**

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

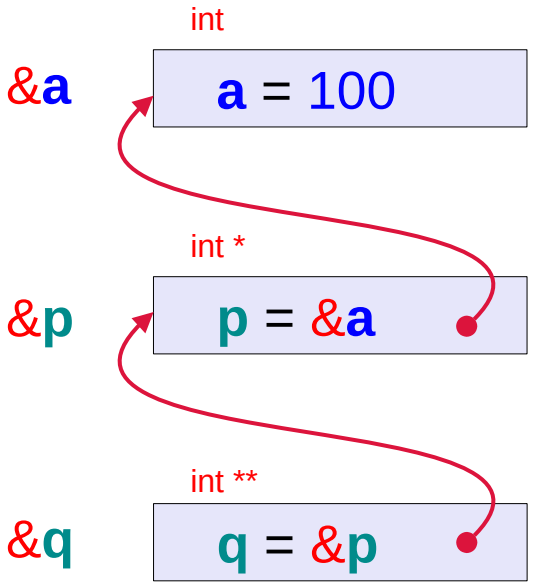
p = &a;

q = &p;



Aliased variables : *p,*q, **q

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



Address assignment **Variable aliasing**

p = &a ; ➡ *p ≡ a

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

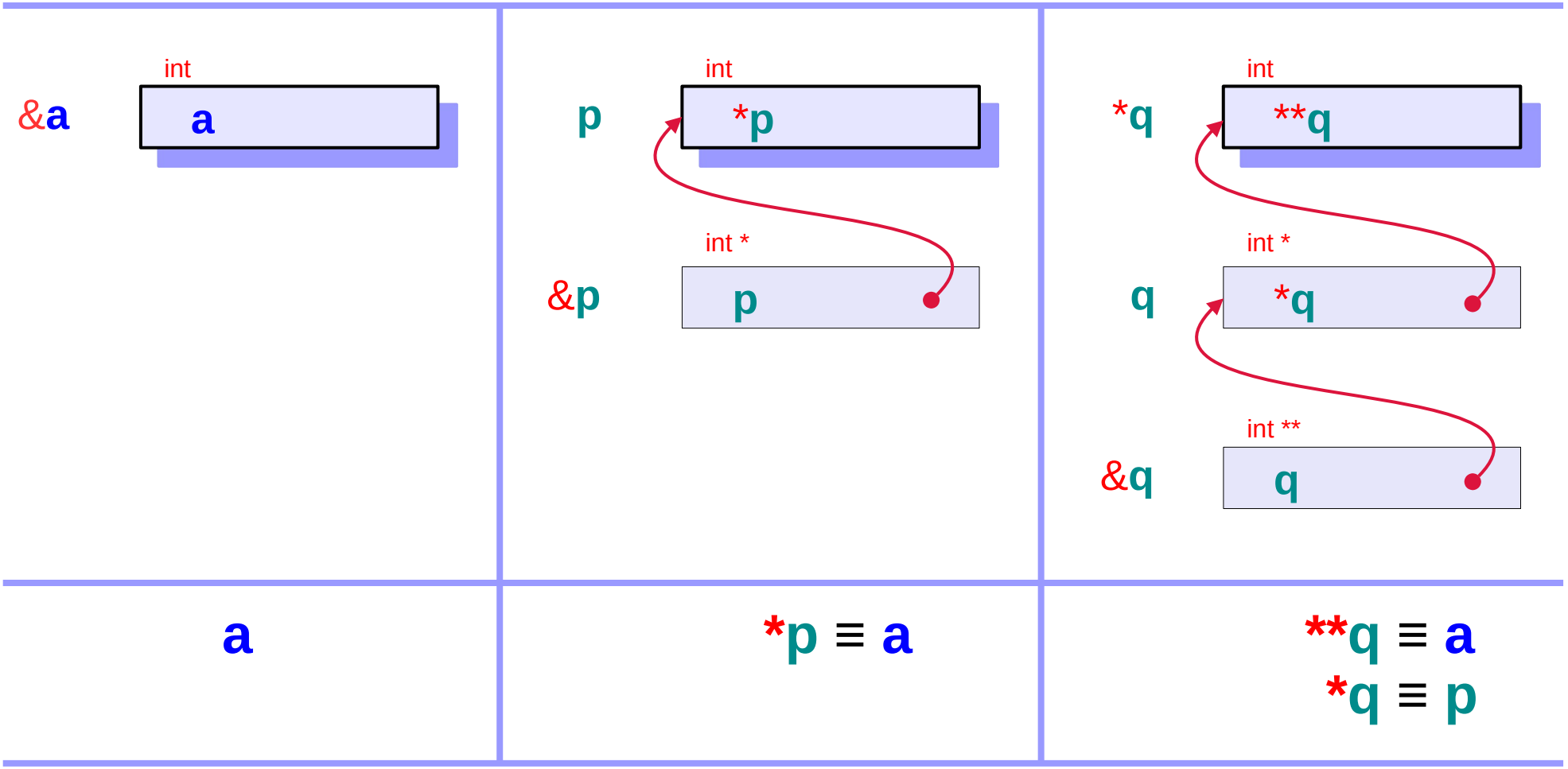
q = &p ; ➡ *q ≡ p

➡ **q ≡ a

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

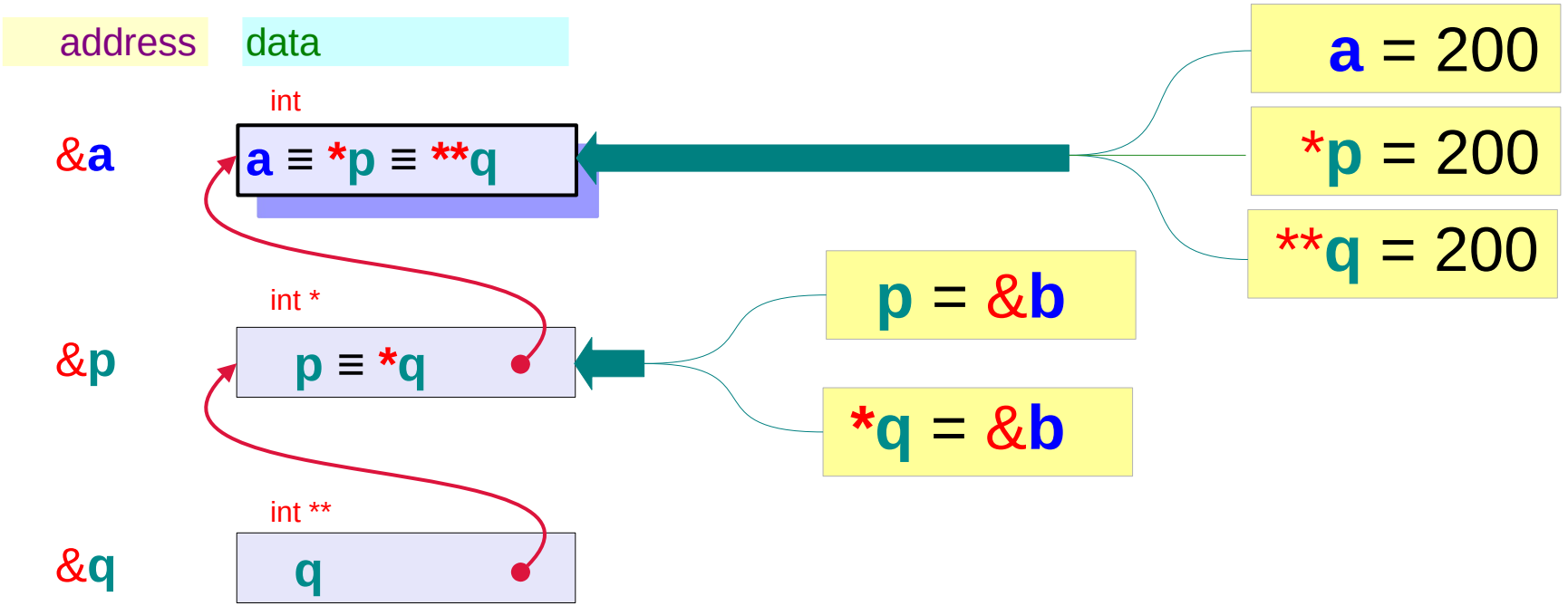
equivalent relations after address assignment

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



a	***p ≡ a**	****q ≡ a** ***q ≡ p**

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



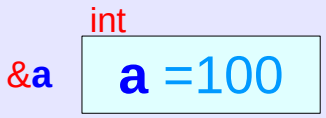
1) Read / Write **p**
2) Read / Write ***q**

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

Single and Double Pointers

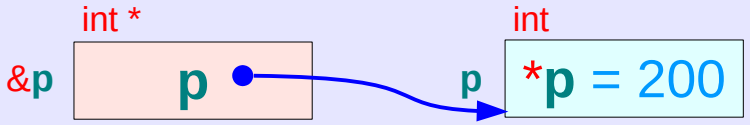
Pointed Addresses and Data

int **a** ;



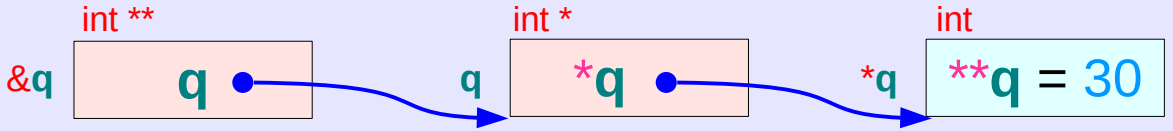
The integer variable **a** holds an integer data

int * **p** ;



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

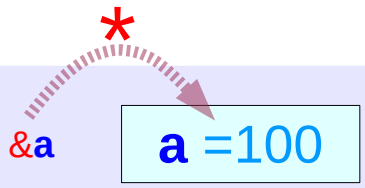
int * * **q** ;



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 *

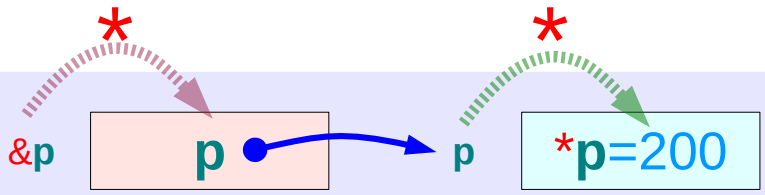
int **a** ;



the expression **a** is a *variable* that can be assigned

*(&**a**) ≡ **a**

int * **p** ;

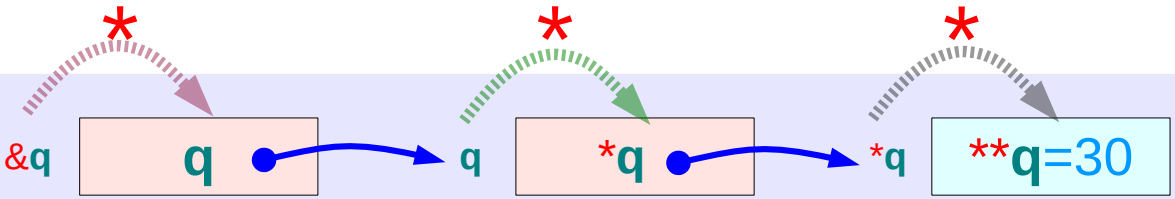


the expression ***p** is a *variable* that can be assigned

*(&**p**) ≡ **p**

*(**p**) ≡ ***p**

int * * **q** ;



the expression ***q**, ****q** are *variables* that can be assigned

*(&**q**) ≡ **q**

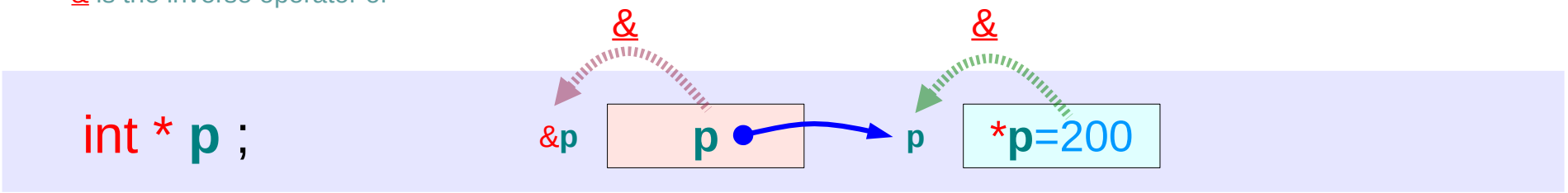
*(**q**) ≡ ***q**

*(*q) ≡ ****q**

Math operator & : the inverse operator of *

*&x returns x
&*y returns y
& is the inverse operator of *

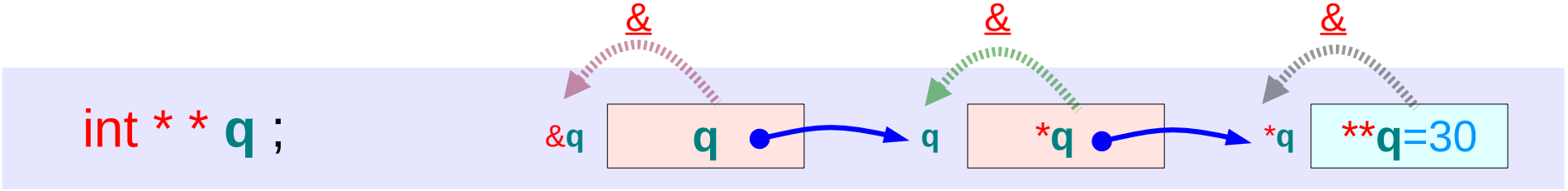
&a



*&x returns x
&*y returns y
& is the inverse operator of *

&p

&(*p) ≡ p



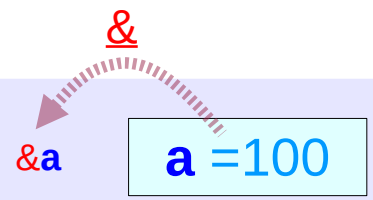
*&x returns x
&*y returns y
& is the inverse operator of *

&q

&(*q) ≡ q

&(**q) ≡ *q

C operator & : address-of operator

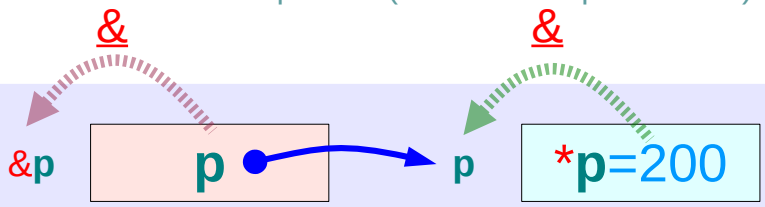


int **a** ;

the C expression **&a** returns
an *address value*
that cannot be assigned

&a ≡ value(**&a**)

& is a mathematical operator (the inverse operator of ***)



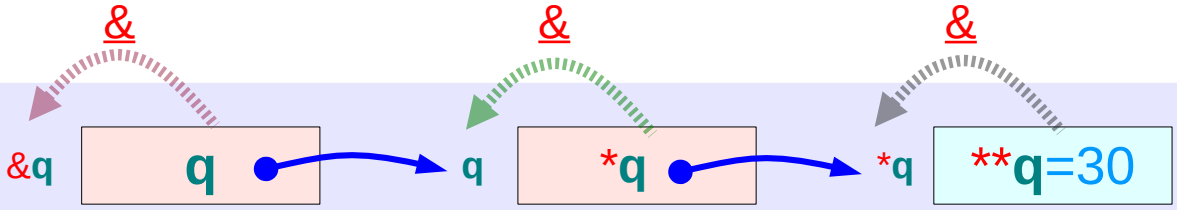
int * **p** ;

the C expression **&p** returns
an *address value*
that cannot be assigned

&p ≡ value(**&p**)

&(*p) ≡ value(**p**)

& is a mathematical operator (the inverse operator of ***)



int * * **q** ;

the C expression **&q** returns
an *address value*
that cannot be assigned

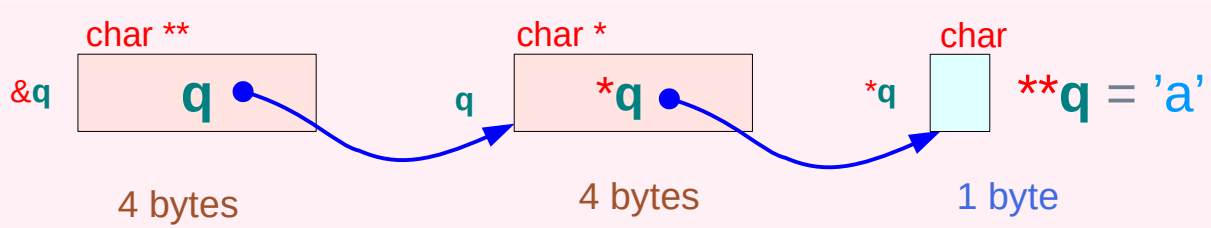
&q ≡ value(**&q**)

&(*q) ≡ value(**q**)

&(q)** ≡ value(***q**)

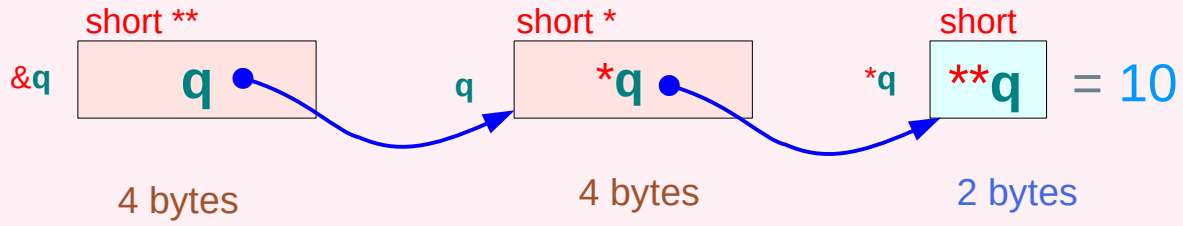
& is a mathematical operator (the inverse operator of ***)

Pointers to various types on 32-bit system (1)



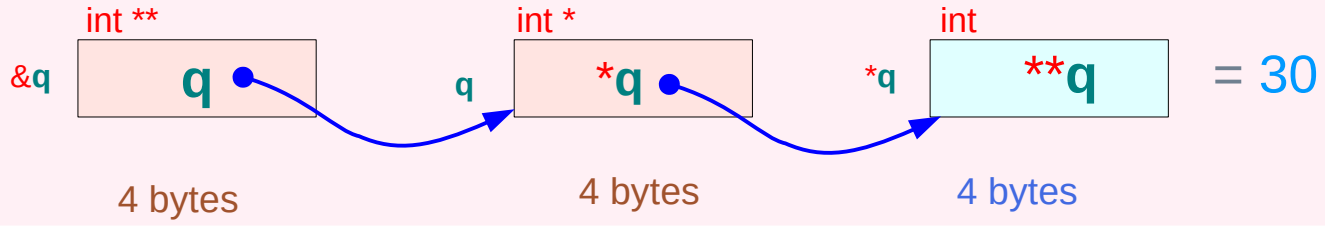

```
char a ;
char * p = &a;
char * * q = &p;
```

on a 32-bit system




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

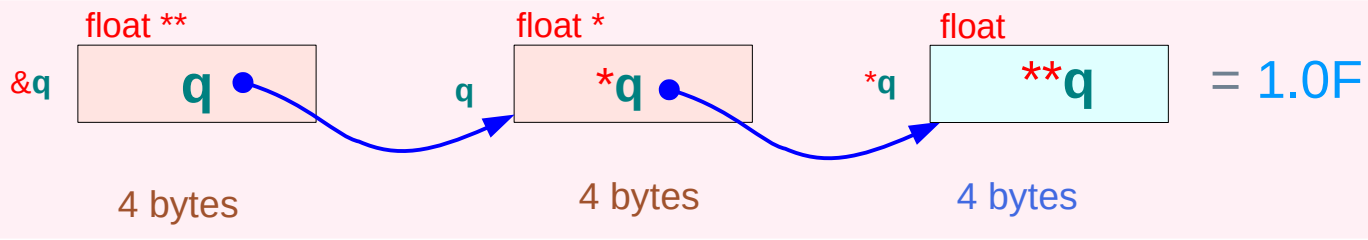
on a 32-bit system




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

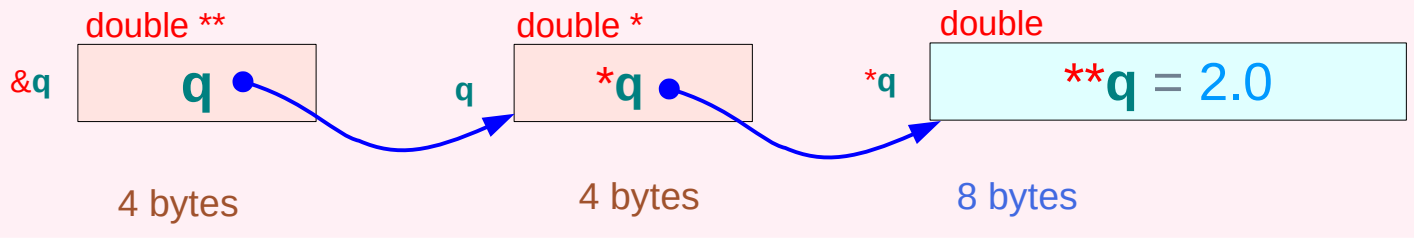
on a 32-bit system

Pointers to various types on 32-bit system (2)




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

on a 32-bit system

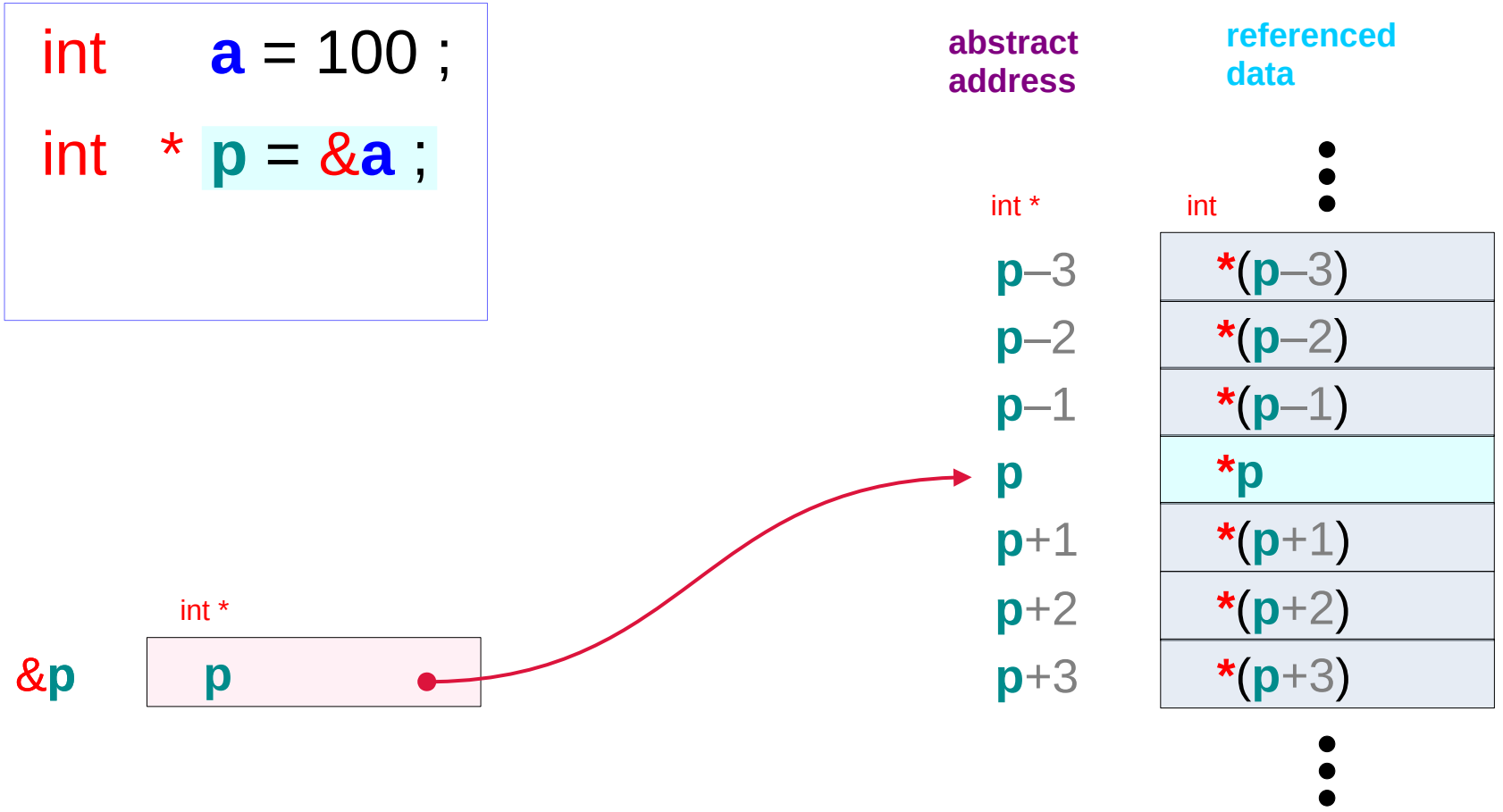



```
double a ;
double * p = &a;
double * * q = &p;
```

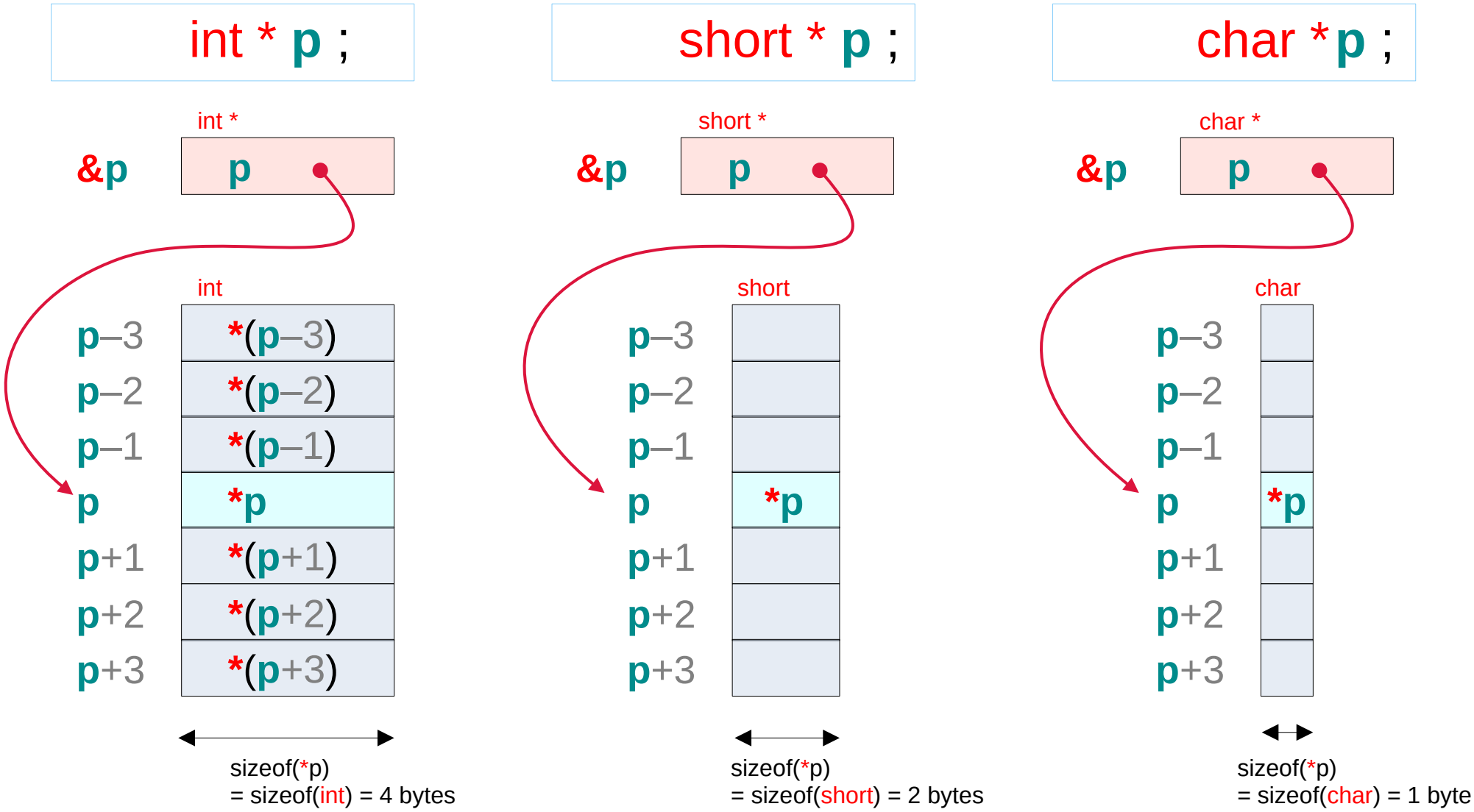
on a 32-bit system

Neighborhood of *p

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

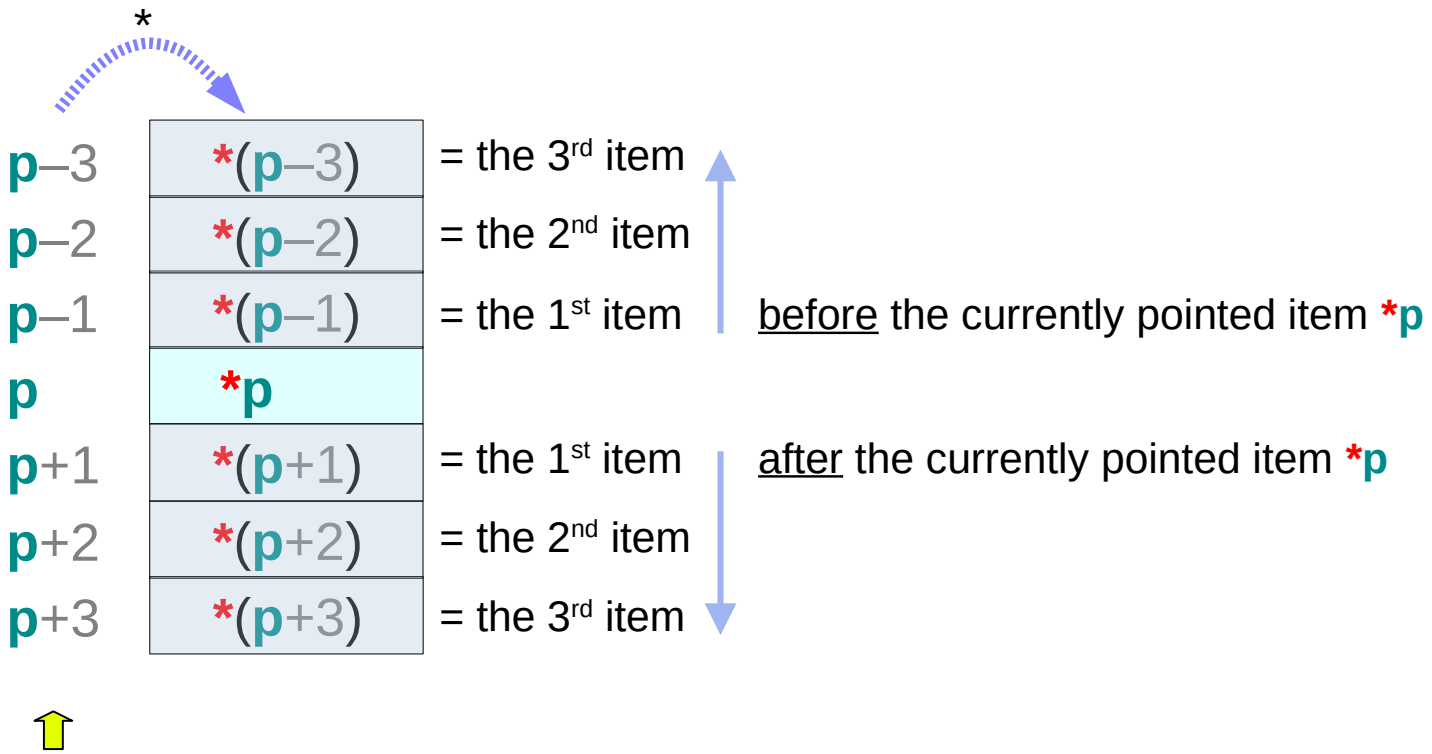


Neighborhood of *p – various sizes of referenced data

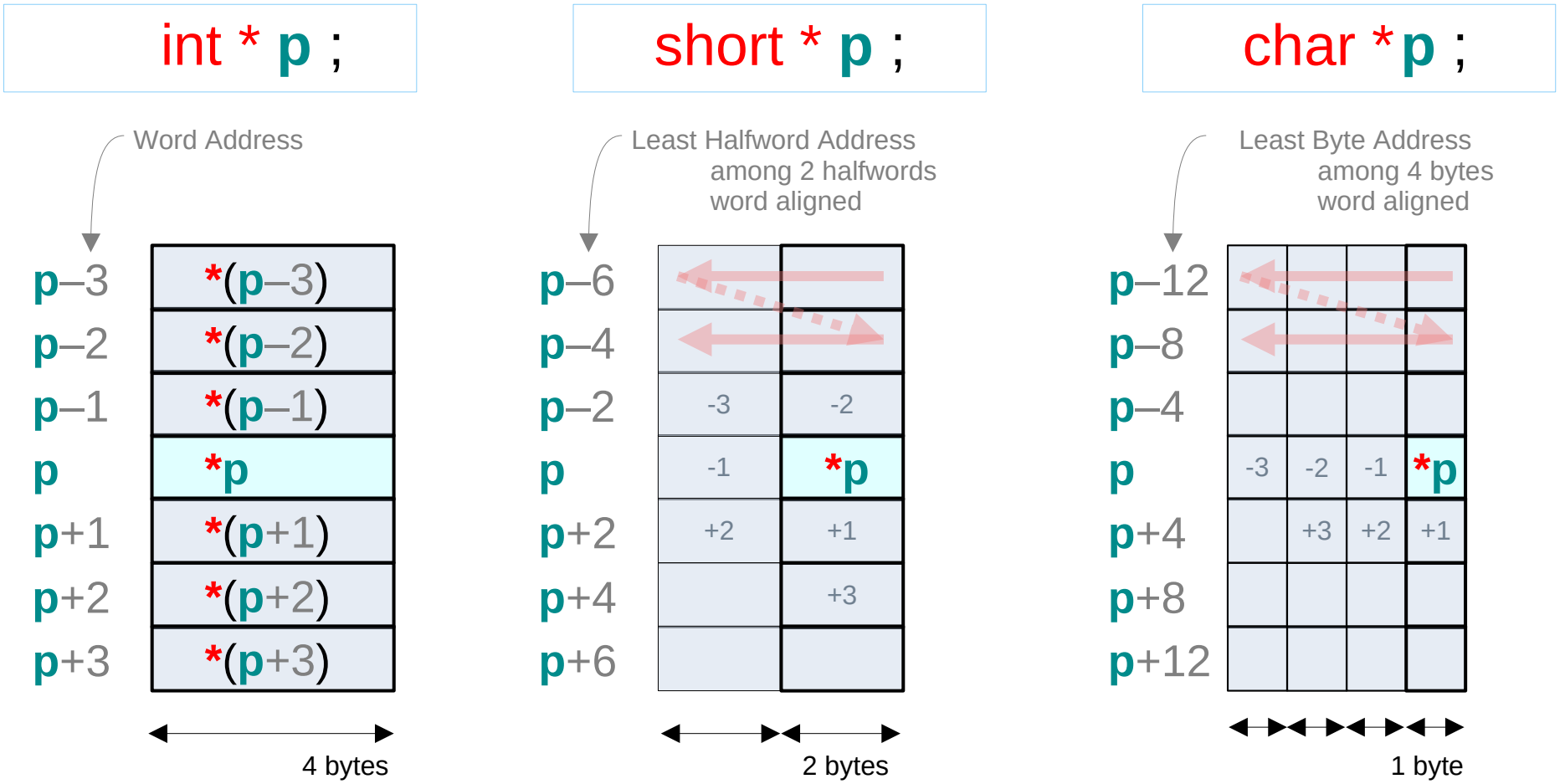


Neighborhood of *p – the same abstract address

int * p ;	short * p ;	char * p ;



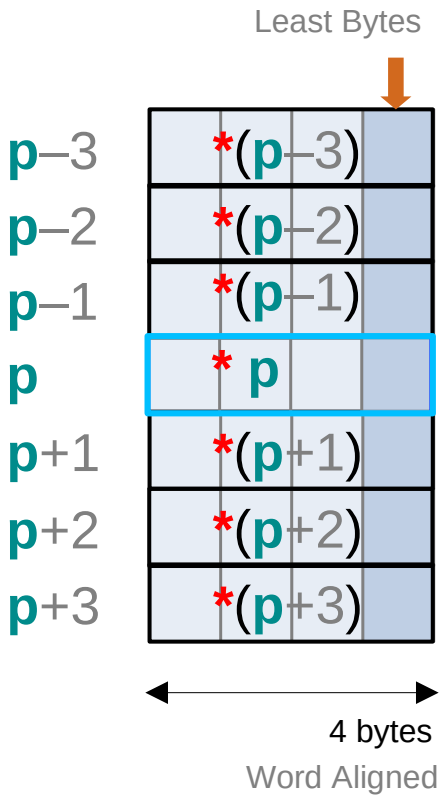
Neighborhood of *p – word aligned view



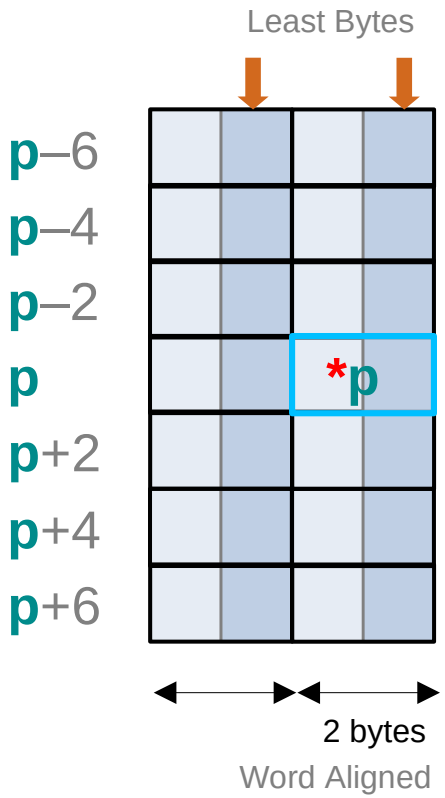
word view

Neighborhood of *p – least bytes

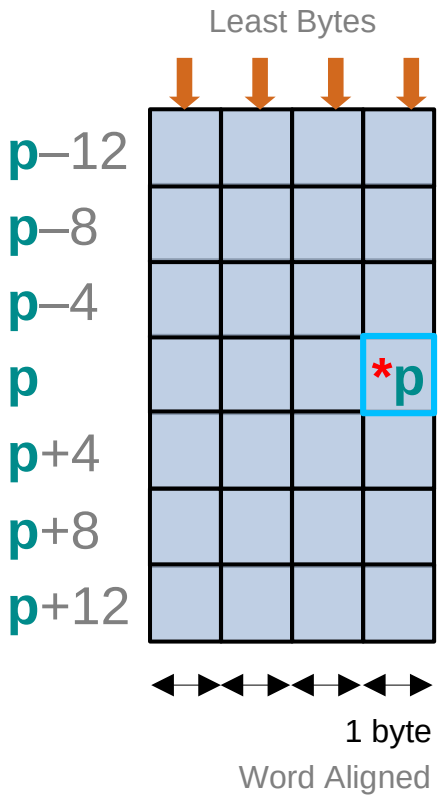
int * p ;



short * p ;

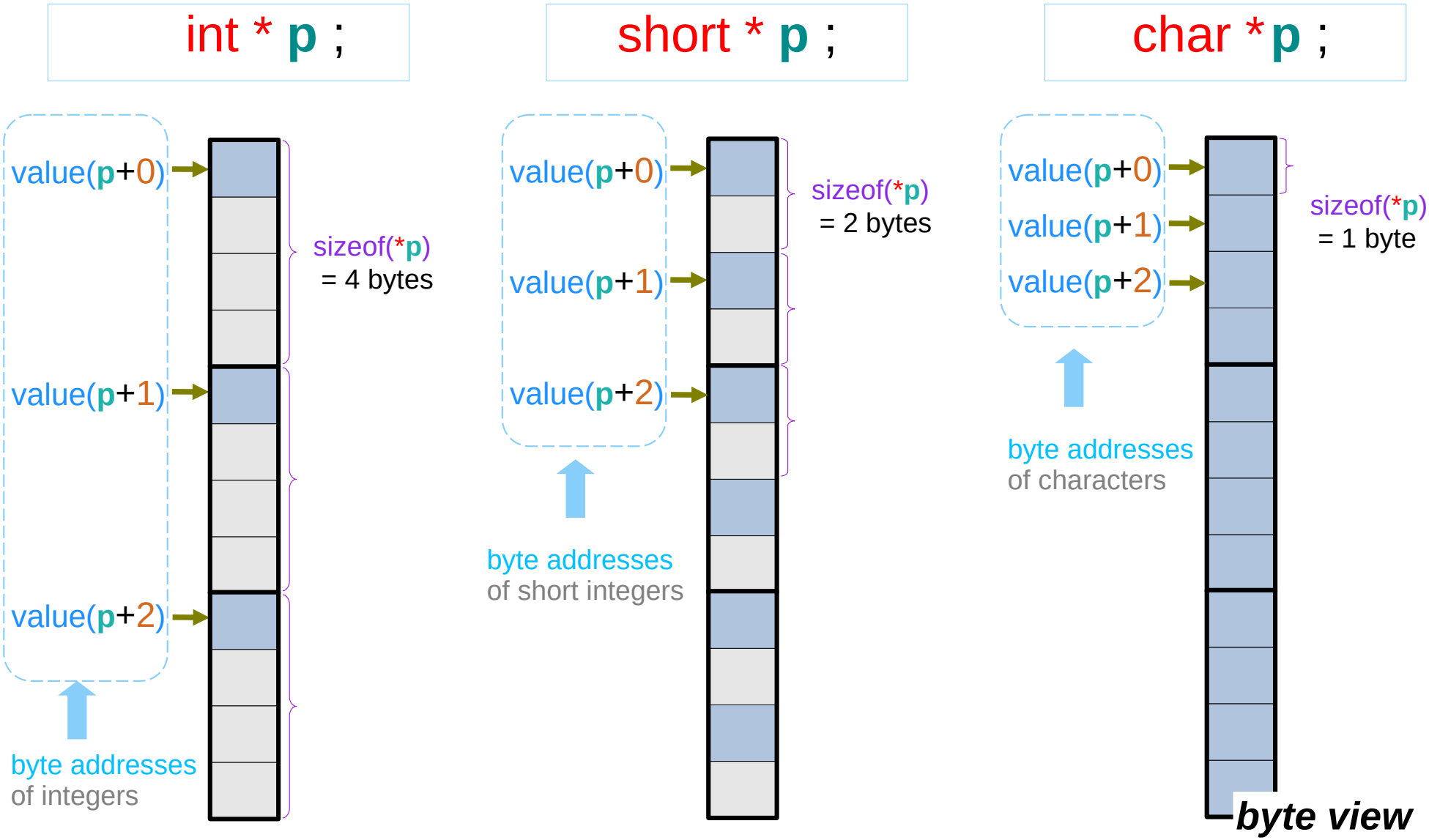


char *p ;



word view

Neighborhood of *p – least byte addresses

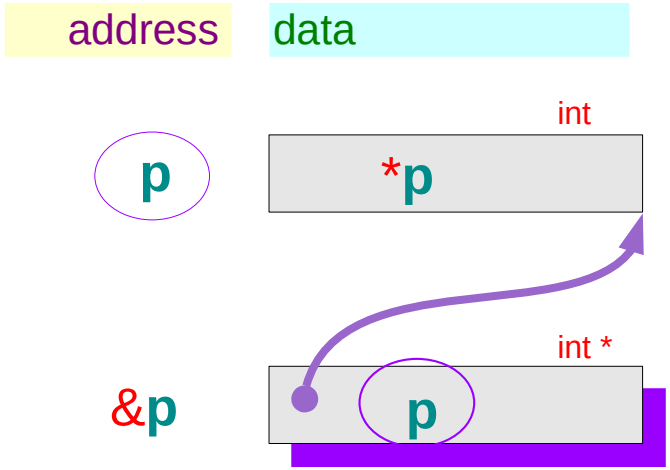


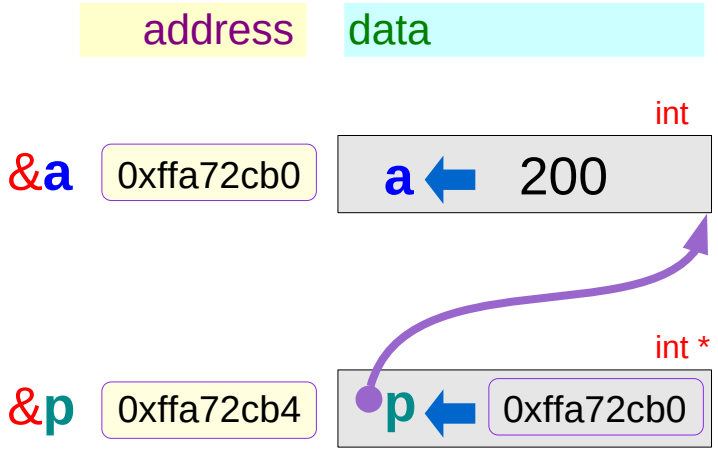
Neighborhood of *p – incremented pointer values

int * p ;	short * p ;	char * p ;
abstract address expression p+1	**abstract address** expression p+1	**abstract address** expression p+1
least byte address of int type *(p+1) value(p+1) = value(p) + sizeof(*p) value(p+i) = value(p) + i*4	**least byte address** of short type *(p+1) value(p+1) = value(p) + sizeof(*p) value(p+i) = value(p) + i*2	**least byte address** of char type *(p+1) value(p+1) = value(p) + sizeof(*p) value(p+i) = value(p) + i*1

Pointer Variable Example

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

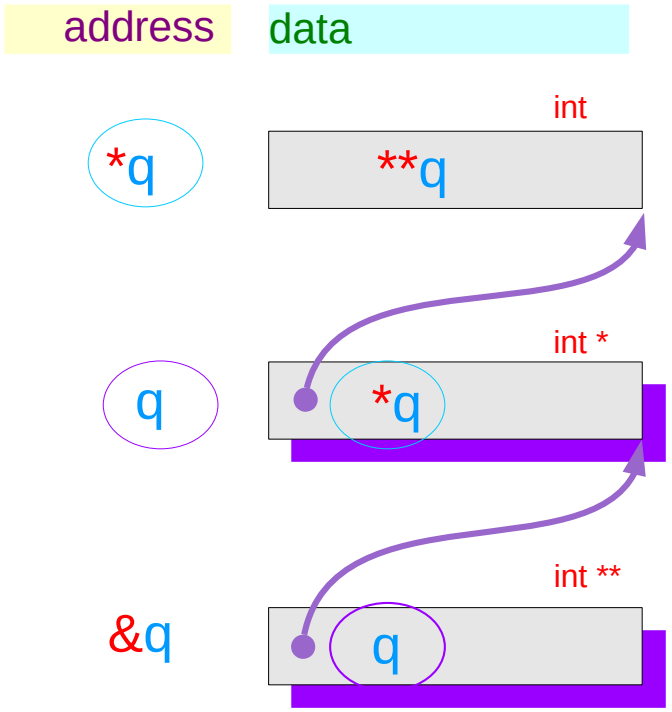


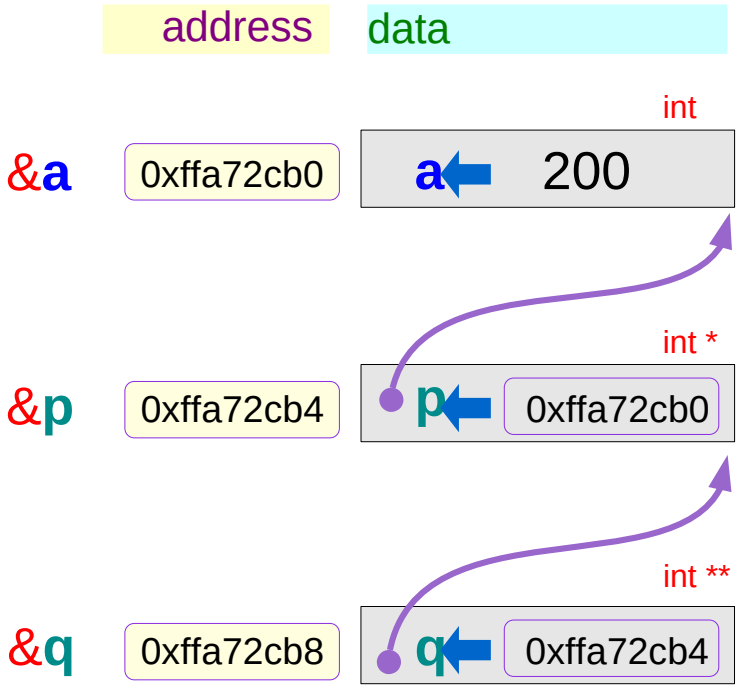



```
int     a = 200;
int *   p = &a;
```

&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

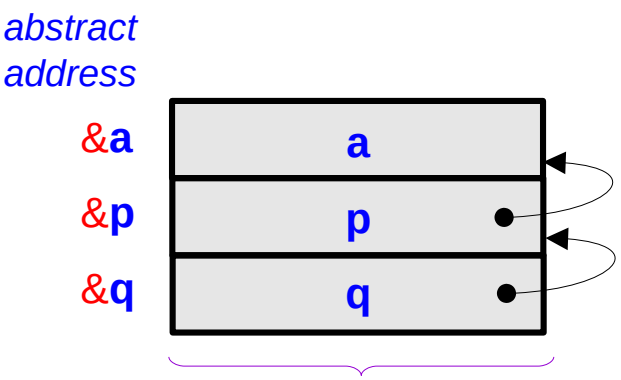
```
// t.c
#include <stdio.h>

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);

}
```

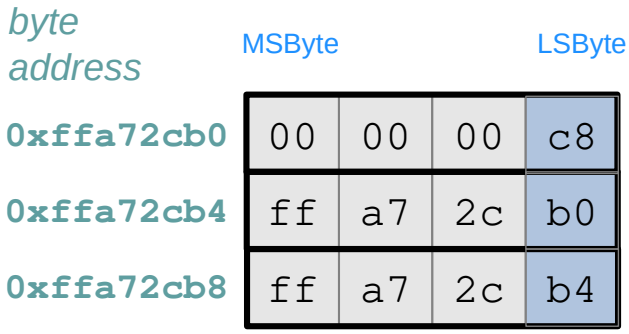
```
  printf("  &p=%12p \n",   &p);
  printf("   p=%12p \n",    p);
  printf("  *p=%12d \n",   *p);
  printf("\n");

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

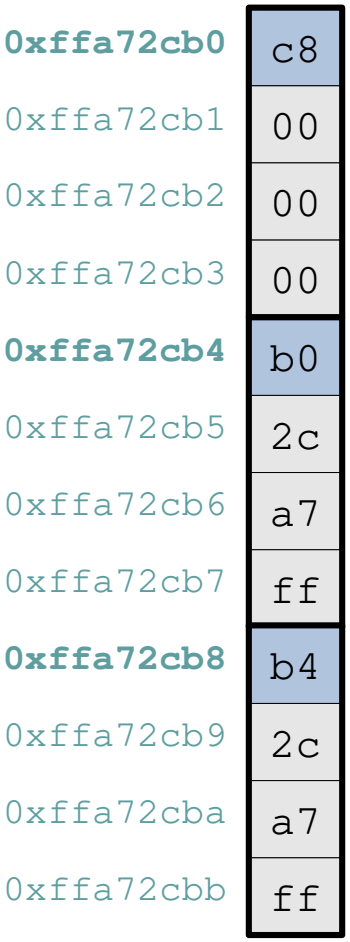
```
gcc -Wall -m32 t.c
./a.out

&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







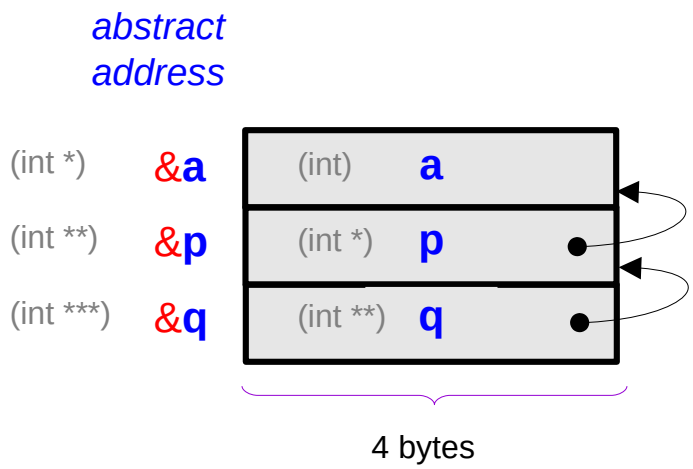

```
int   a = 200 ;
int  *p = &a ;
int **q = &p ;

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

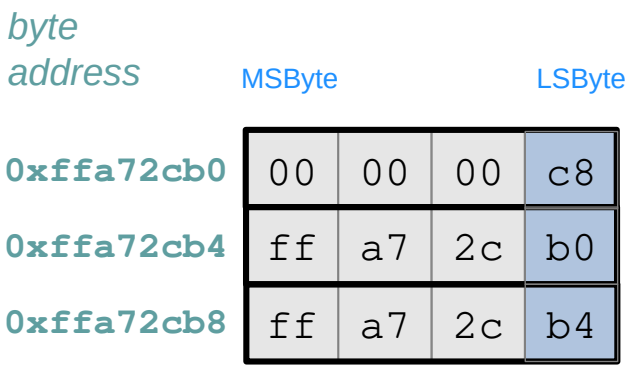
here, the values of &a, &p, &q
are displayed

address value

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

Abstract vs. byte addresses





*abstract address contains information of **type**, **size**, **value***

*byte address contains **value** information only*

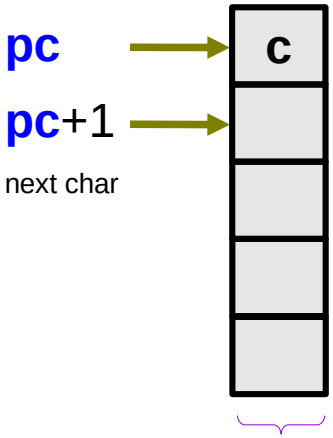
Abstract Address	*Type*	*Size*	*Value (address value)*
&a	int *	4 bytes	0xffa72cb0
&p	int **	4 bytes	0xffa72cb4
&q	int ***	4 bytes	0xffa72cb8

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

Endianness and memory alignment

Pointers and abstract addresses

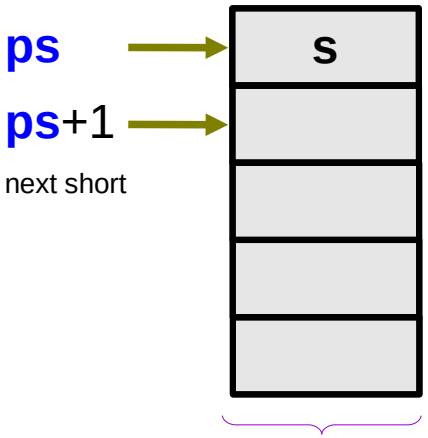
```
char c ;
char *pc = &c ;
```



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

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

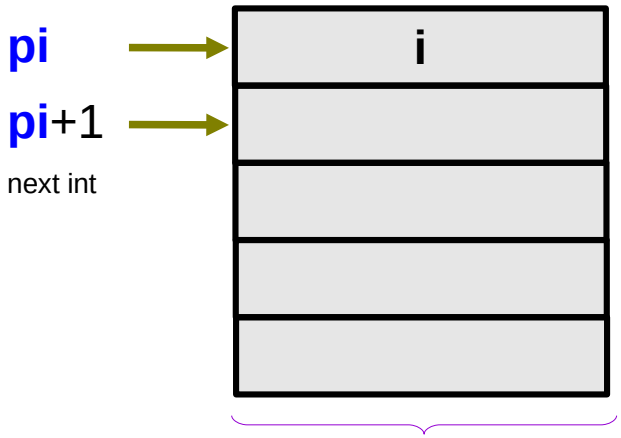
```
short s ;
short *ps = &s ;
```



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 ;
```

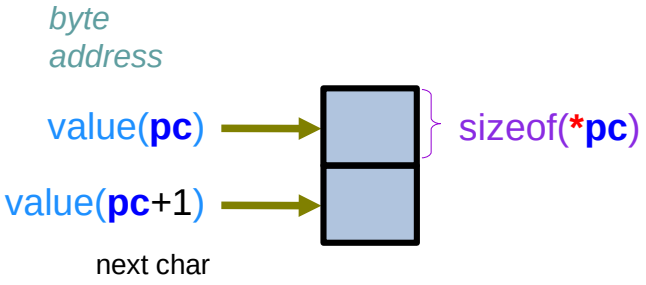


sizeof(*pi) =
sizeof(i) =
sizeof(int) = 4 bytes

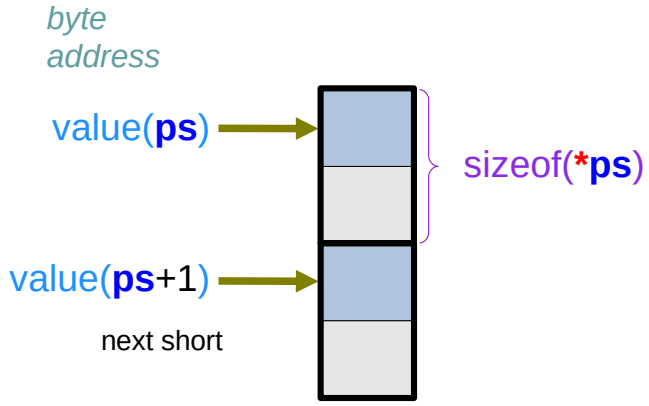
sizeof(pi) = 4 or 8 bytes
type(pi) = int *
value(pi) = byte address

Pointer values and byte addresses

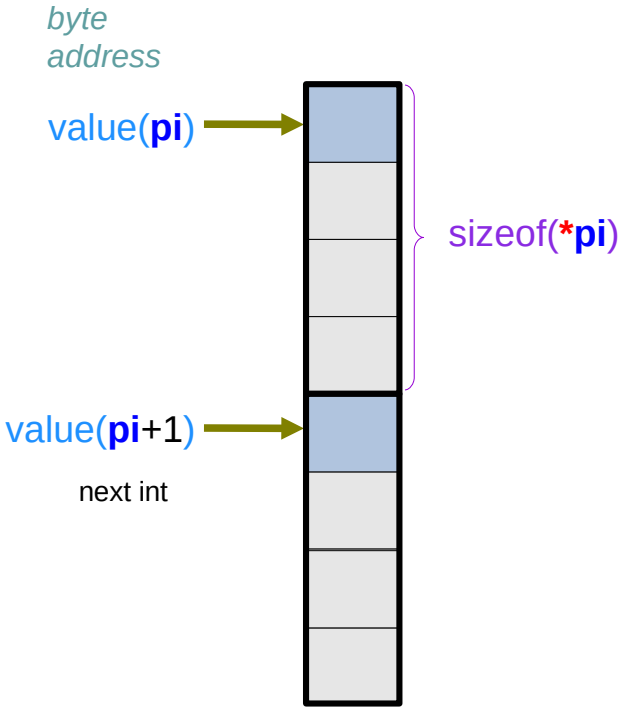
char ***pc** ;



short ***ps** ;



int ***pi** ;



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**)

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

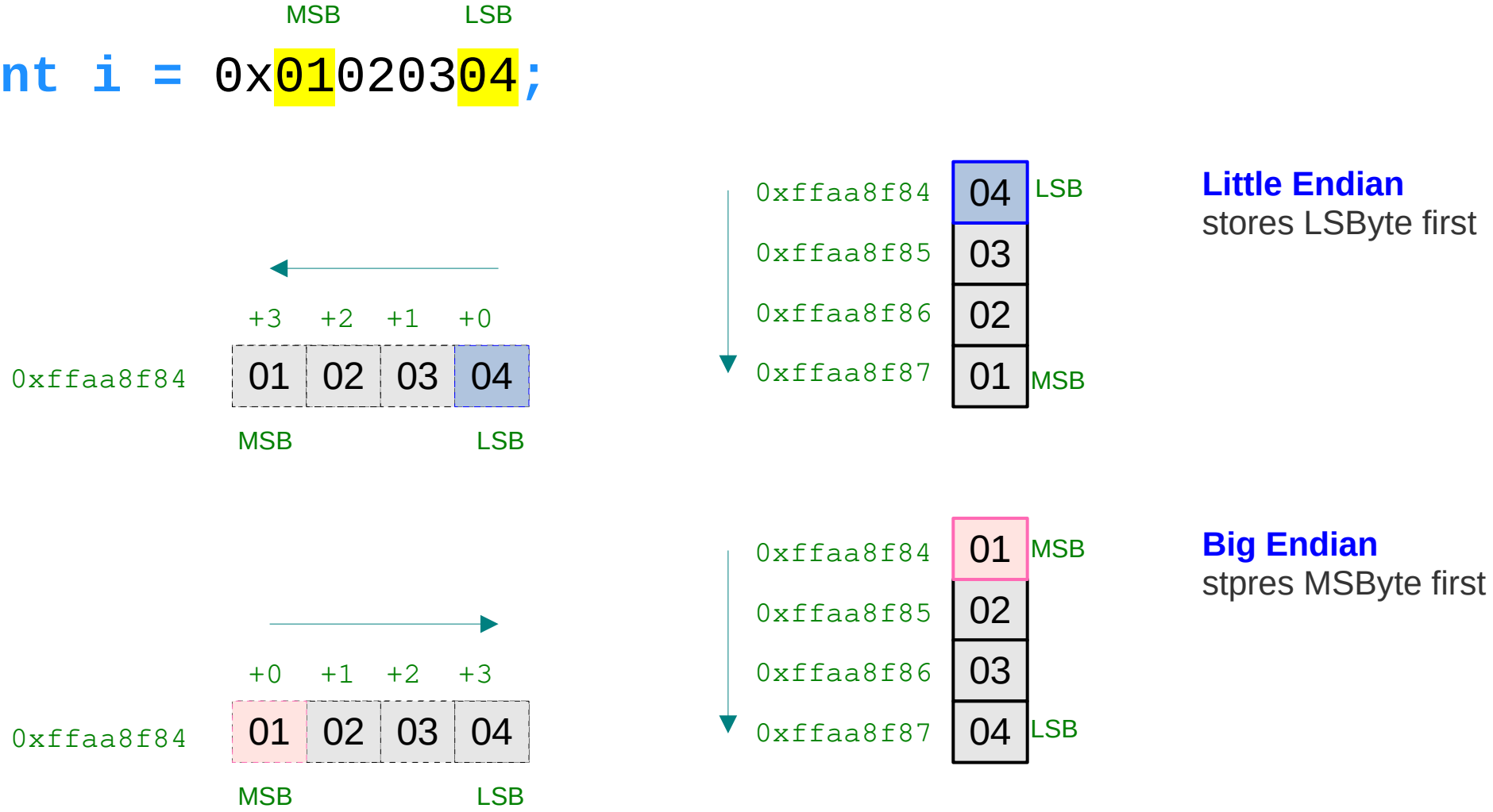
value(**ps**+1) = value(**ps**) + sizeof(***ps**)

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

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

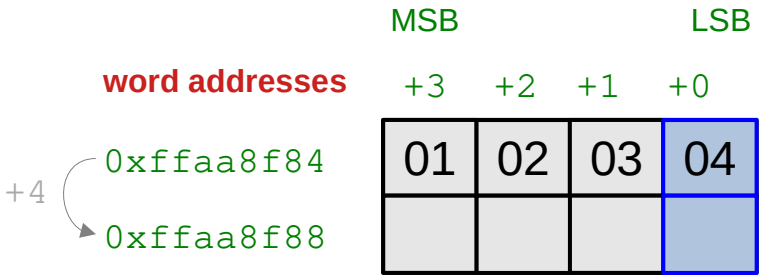


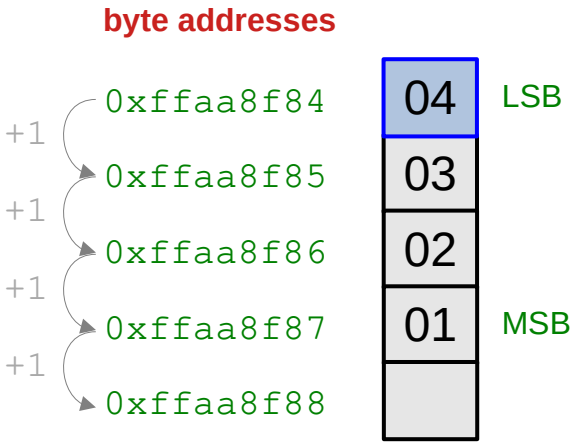
Little Endian
stores LSByte first

Big Endian
stpres MSByte first

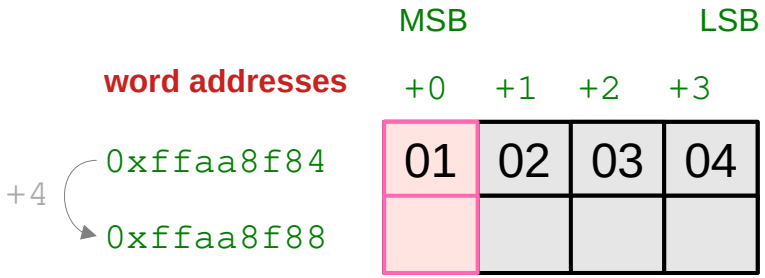
Word addresses and byte addresses

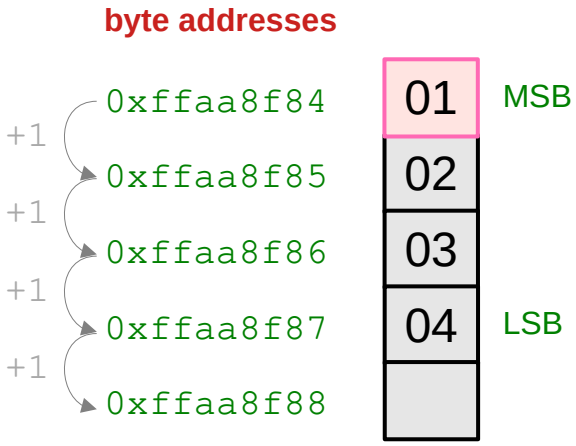
Little Endian store LSByte first





Big Endian store MSByte first

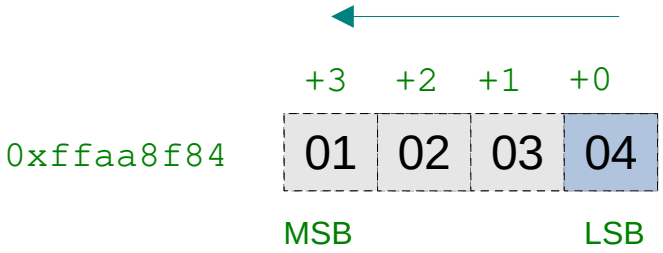




Little Endian Examples

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

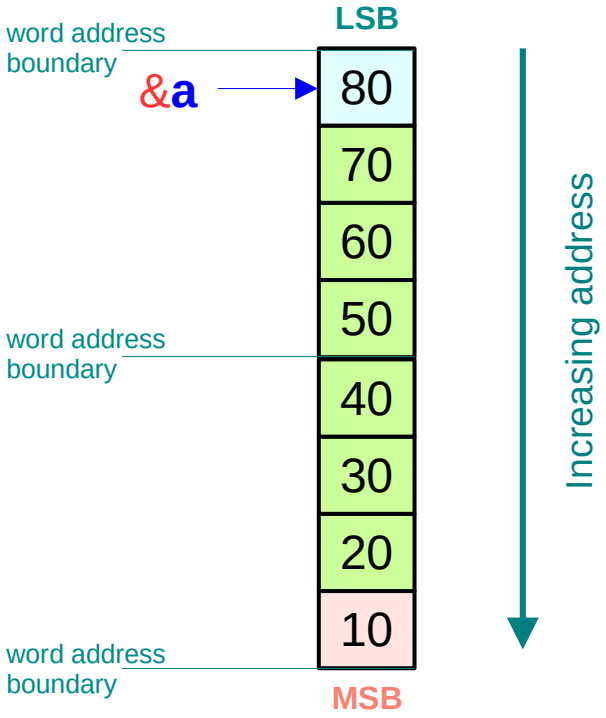
long **a** = 0x10203040506070**80** ;

MSB LSB

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

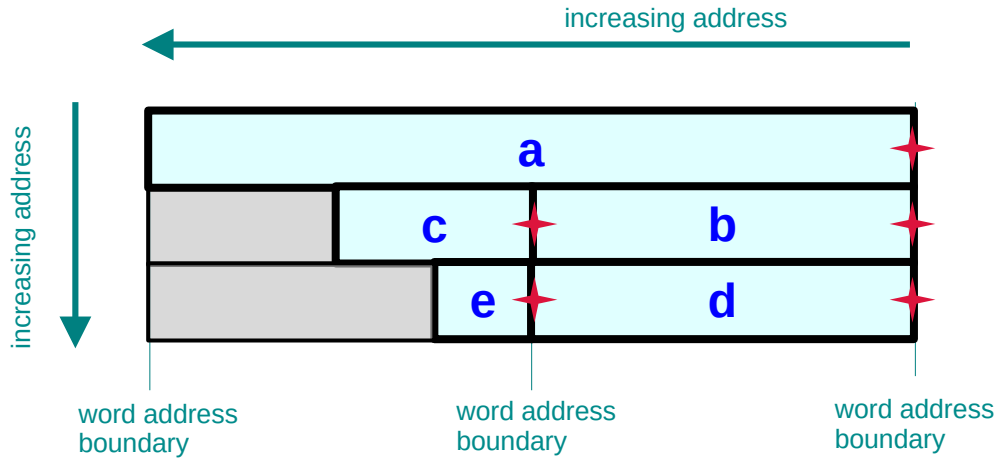
Little Endian System

store the LSB first

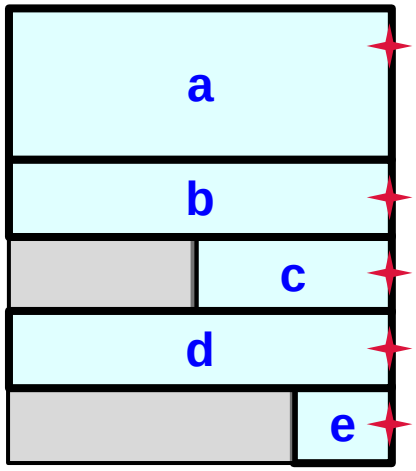


Memory alignment in a little endian system

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

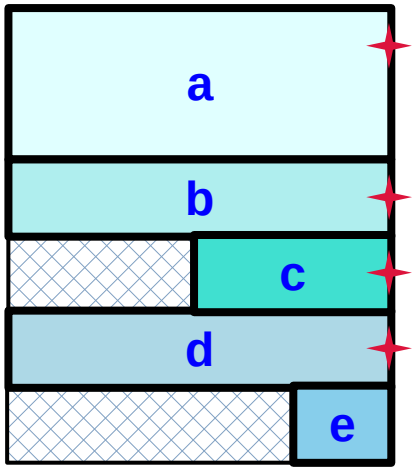



```
long    a = 0x10203040506070 80 ;
int     b = 0x1122334 4 ;
short   c = 0x12 34 ;
int     d = 0x0102030 4 ;
char    e = 0x 99 ;
```

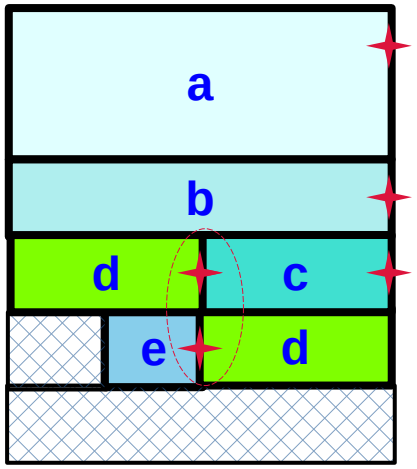


Memory alignment – performance

Memory alignment



No memory alignment



saves memory

loses performance :

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

memory hardware constraints

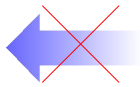
```
long    a = 0x1020304050607080 ;   MSB ... LSB
int     b =         0x11223344 ;   MSB ... LSB
short   c =               0x1234 ; MSB ... LSB
int     d =         0x01020304 ;   MSB ... LSB
char    e =               0x99 ;   LSB
```

Pointer Type Cast

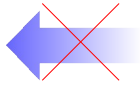
Assigning pointer variables

long a ; &a address of a long value

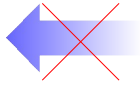
address of an int value

int * p ; p  address of a long value &a

address of a short value

short * q ; q  address of a long value &a

address of a char value

char * r ; r  address of a long value &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 ;
```

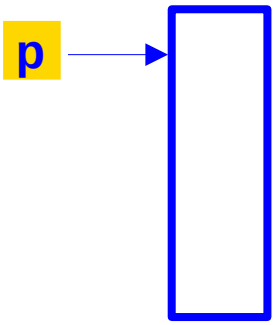
address of a char value `r`

char address conversion `(char *) &a`

View windows – type cast pointer

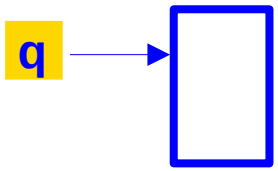
integer view window

int *



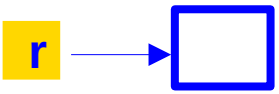
short view window

short *

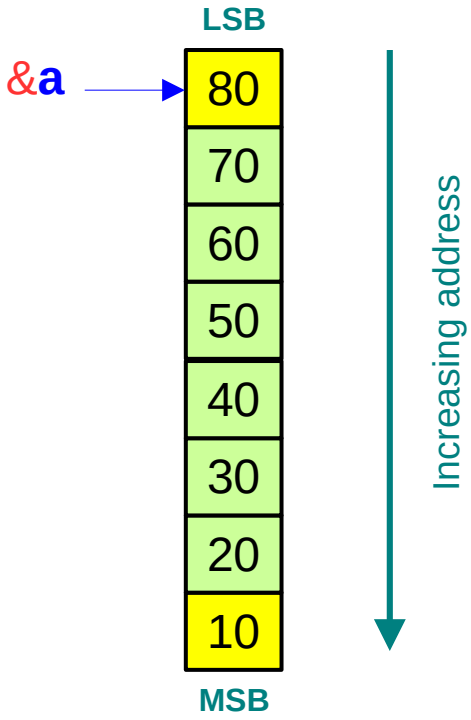


character view window

char *

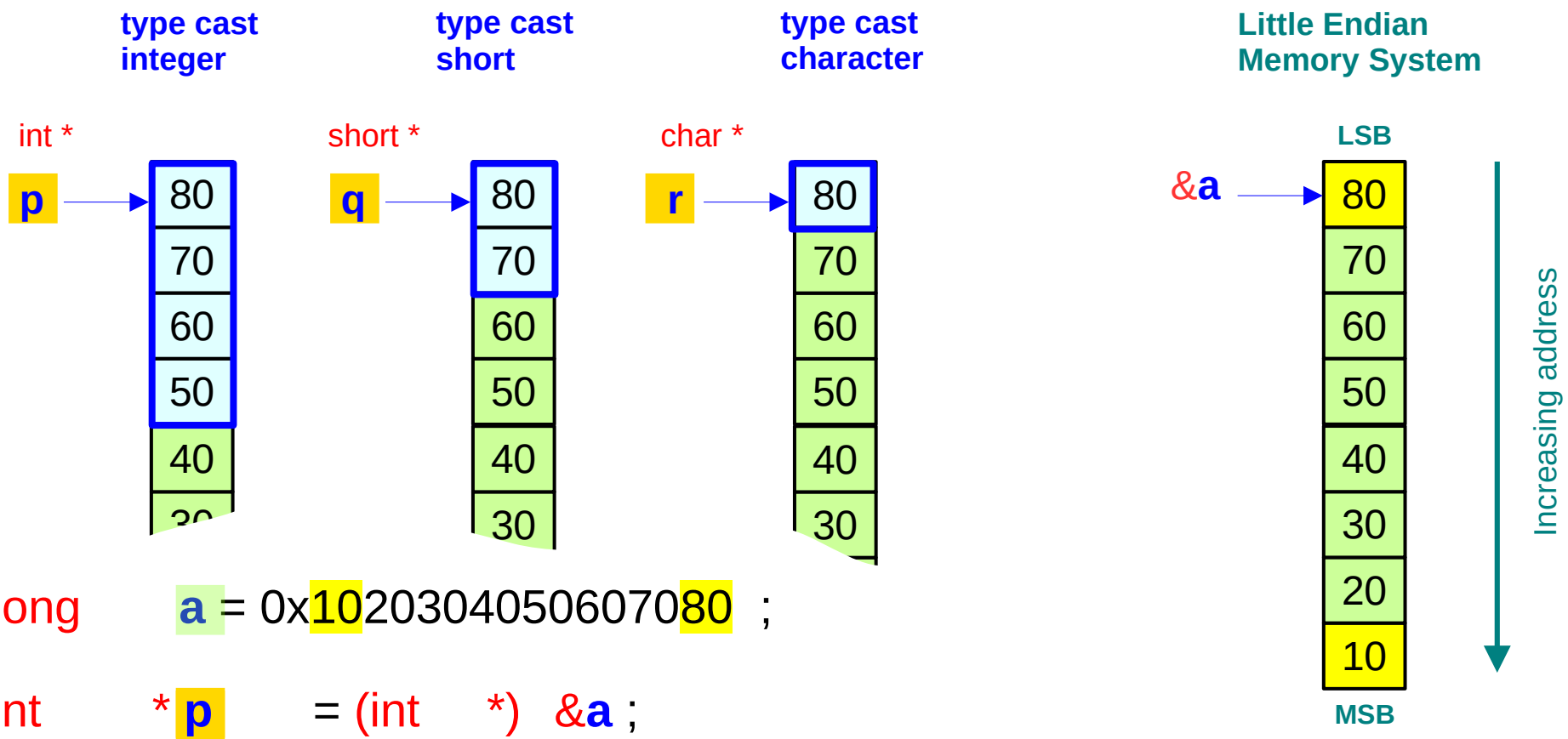


Little Endian Memory System




```
                MSB                 LSB
long     a = 0x1020304050607080 ;

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

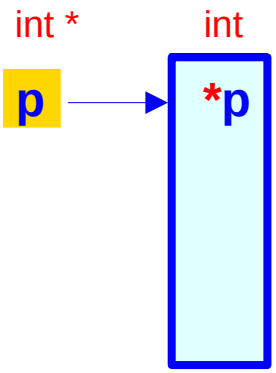
Applying type cast pointers to a memory location



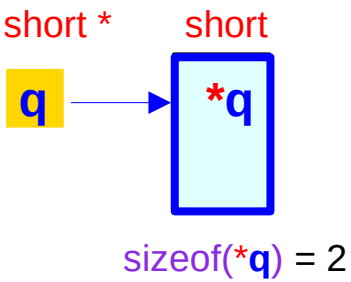

```
long    a = 0x10203040506070 80 ;

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

De-referencing type cast pointers

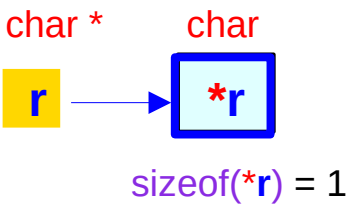
type cast integer pointer



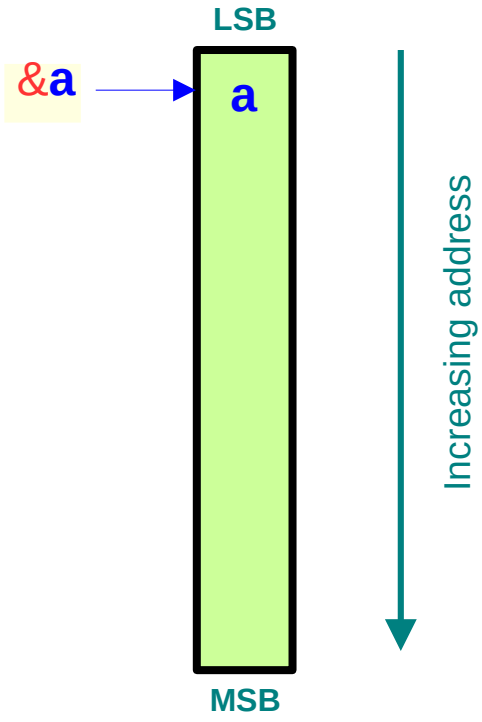
type cast short pointer



type cast character pointer



Little Endian Memory System

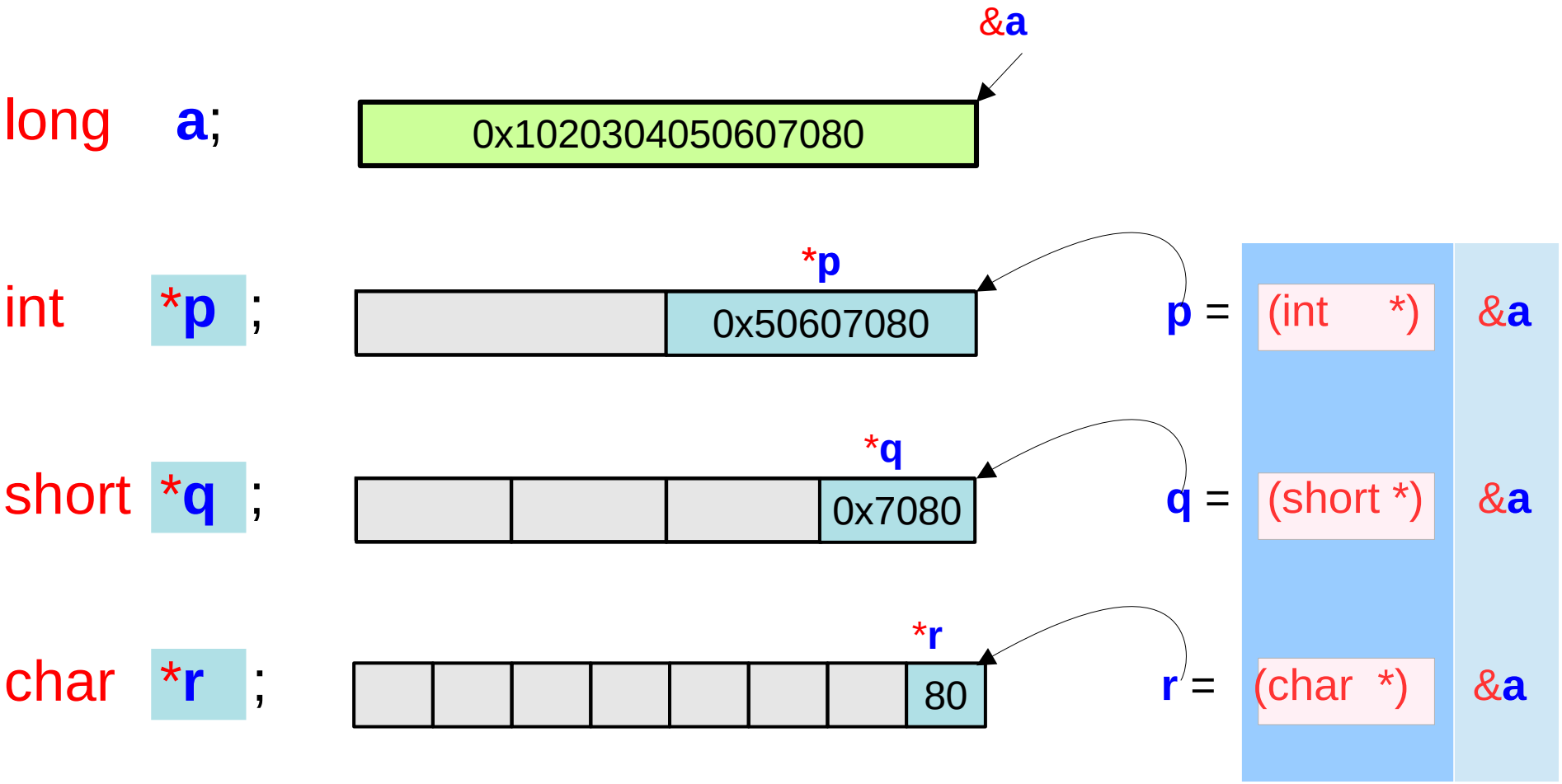



```
long    a = 0x10203040506070 80 ;
```

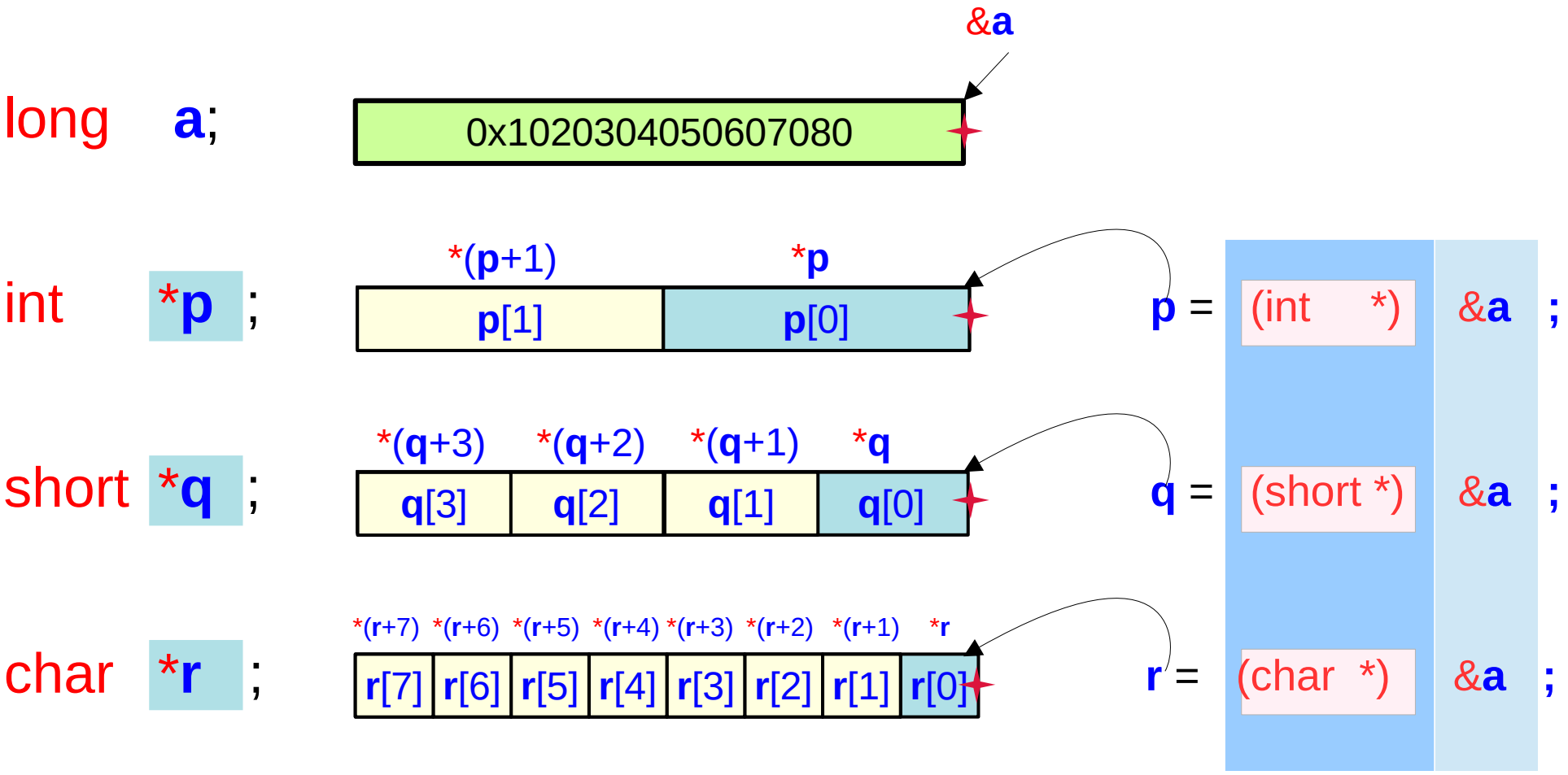
```
int     * p = (int   *) &a ;
short   * q = (short *) &a ;
char    * r = (char  *) &a ;
```

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

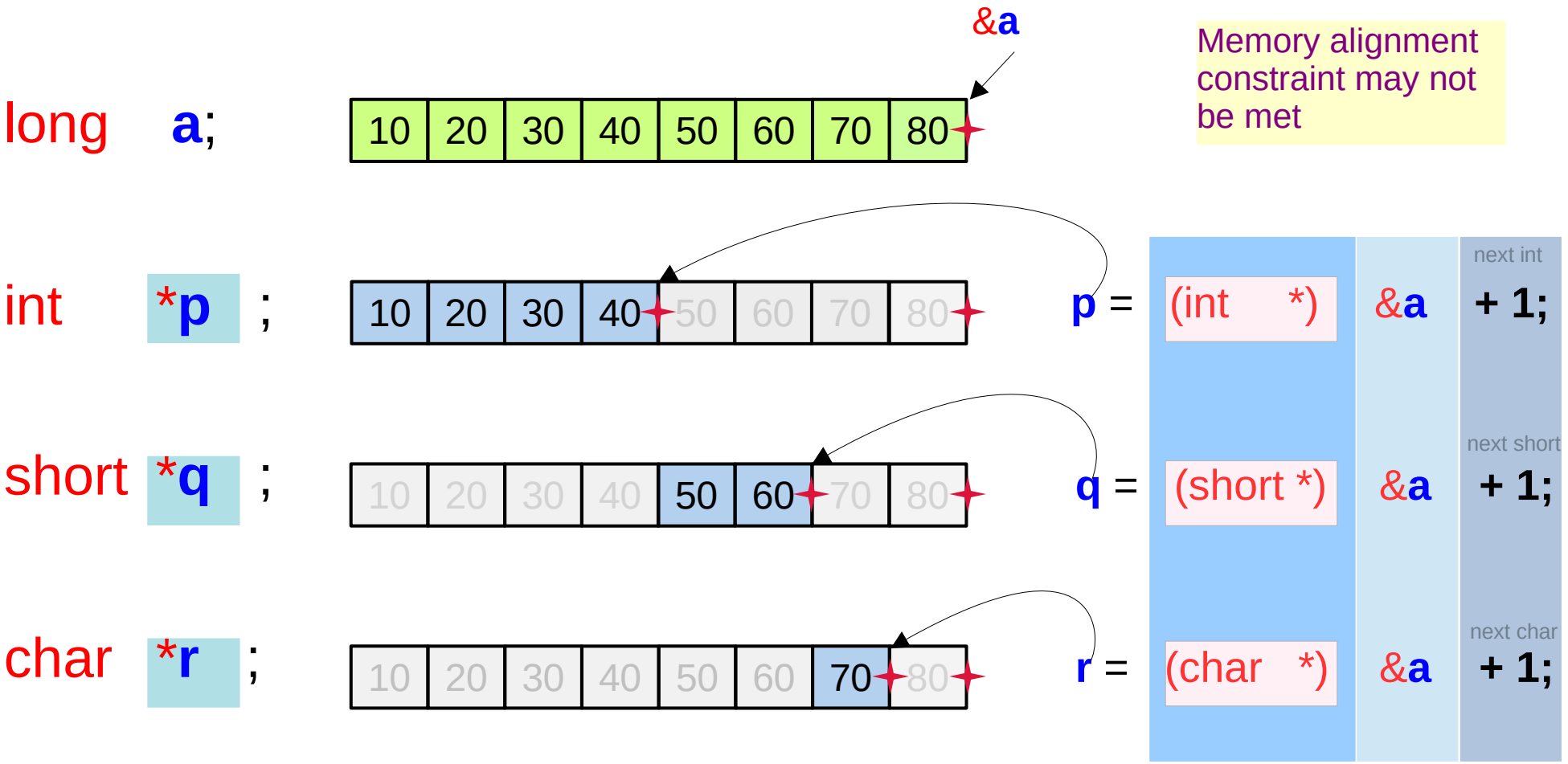
Re-interpretation of memory data – case I



Re-interpretation of memory data – case I

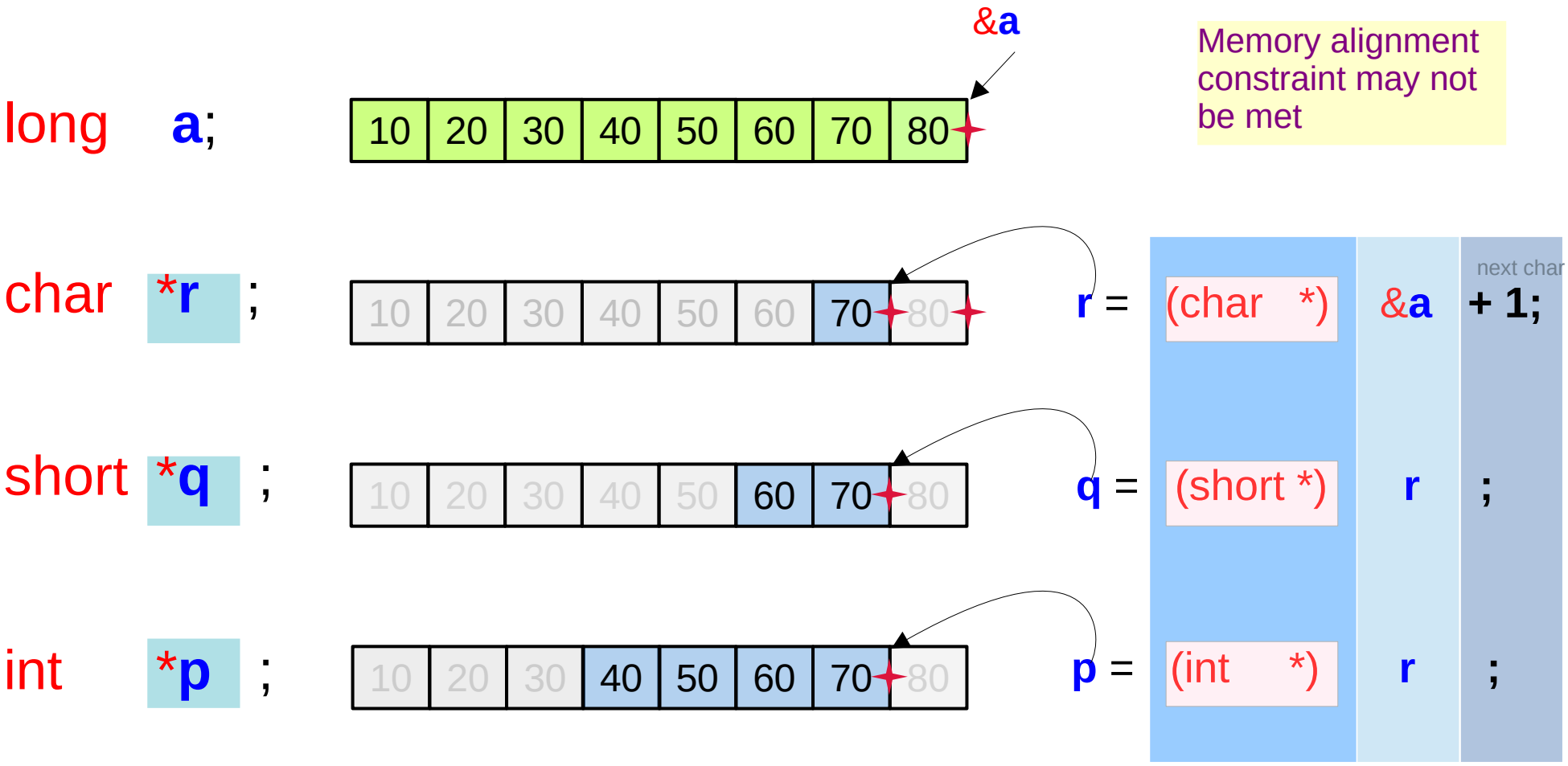


Re-interpretation of memory data – case II



Memory alignment constraint may not be met

Re-interpretation of memory data – case III



Arrays

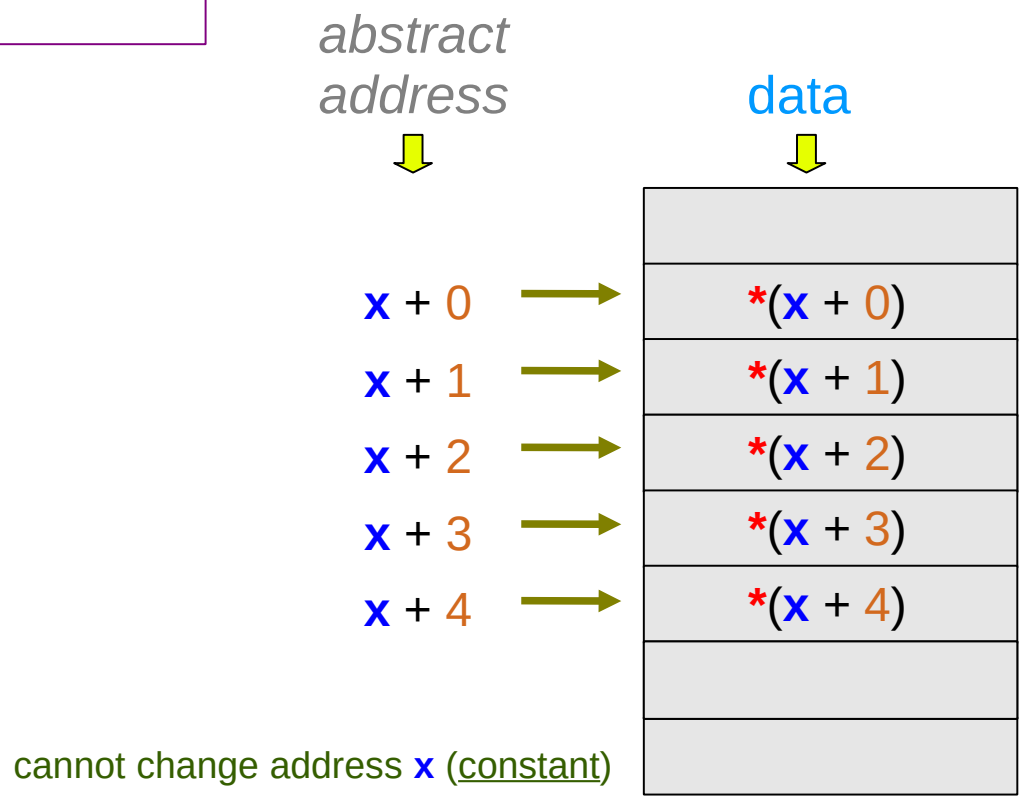
Accessing array elements – using abstract addresses

```
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*

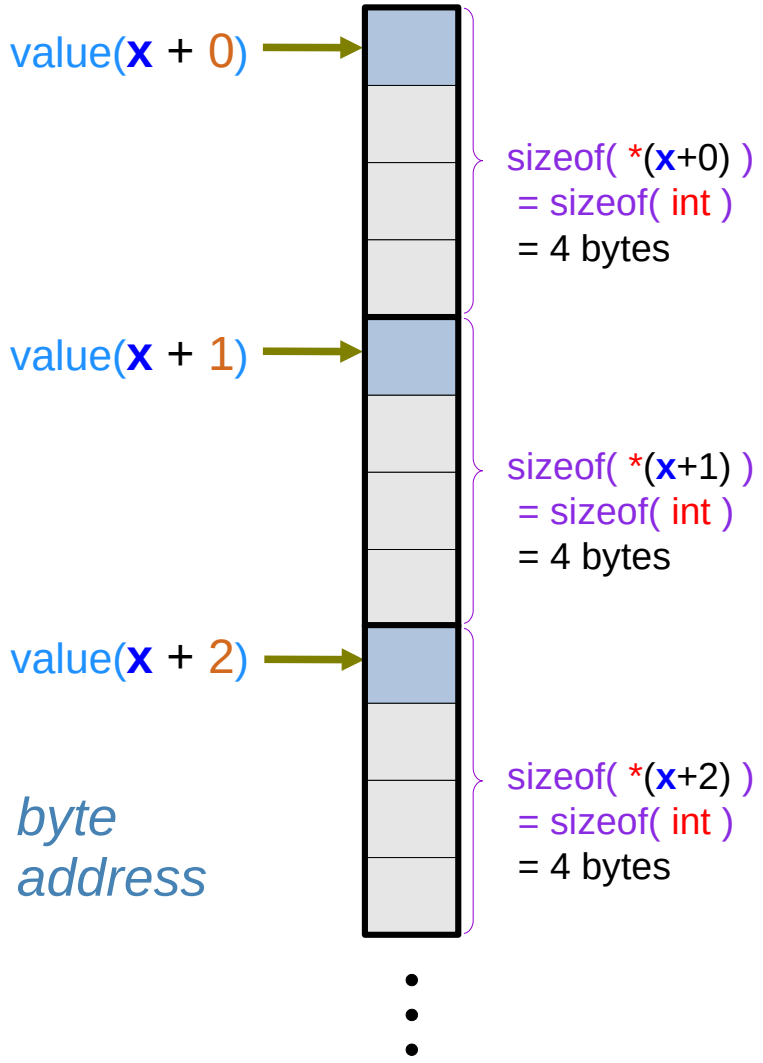
5 int variables

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



Two aspects of **x**

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

x is an array of 5 integer elements

abstract data x ..... size

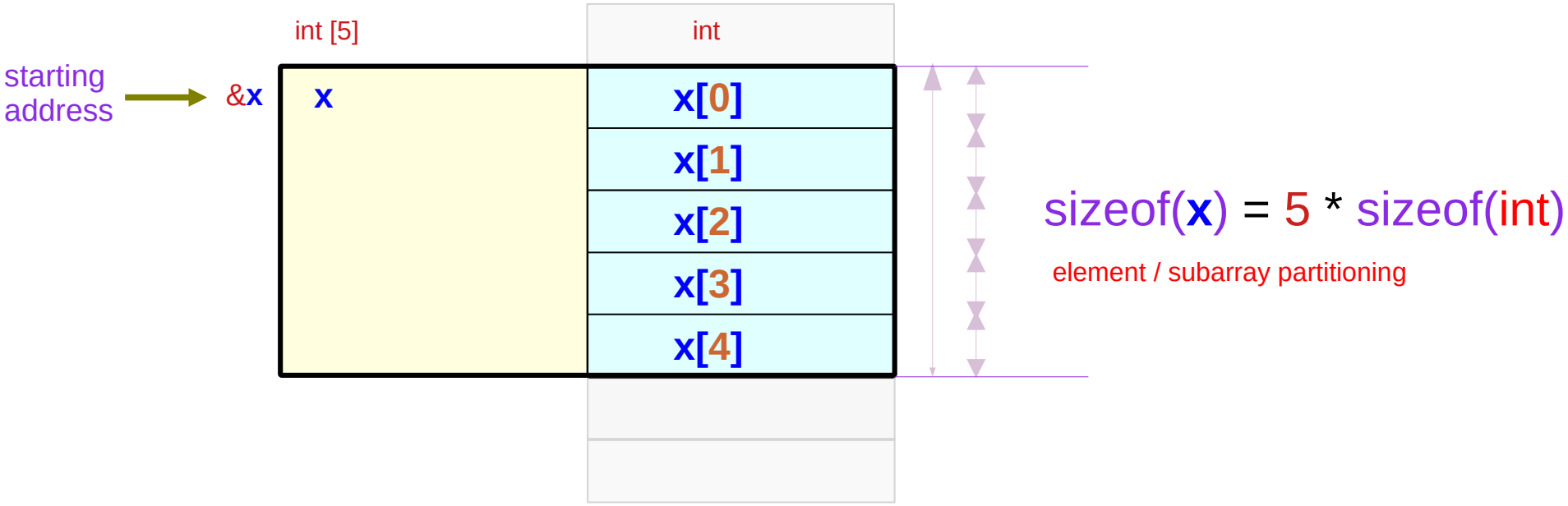
pointer x .....starting address

Abstract data **x**

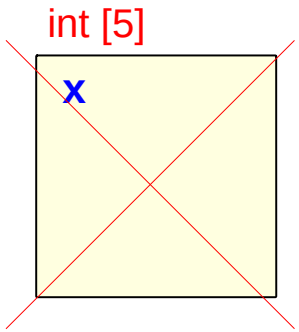
```
int     x[5];
```

the size of **x** is the total size
of **5** consecutive **int** variables

abstract data x **size**



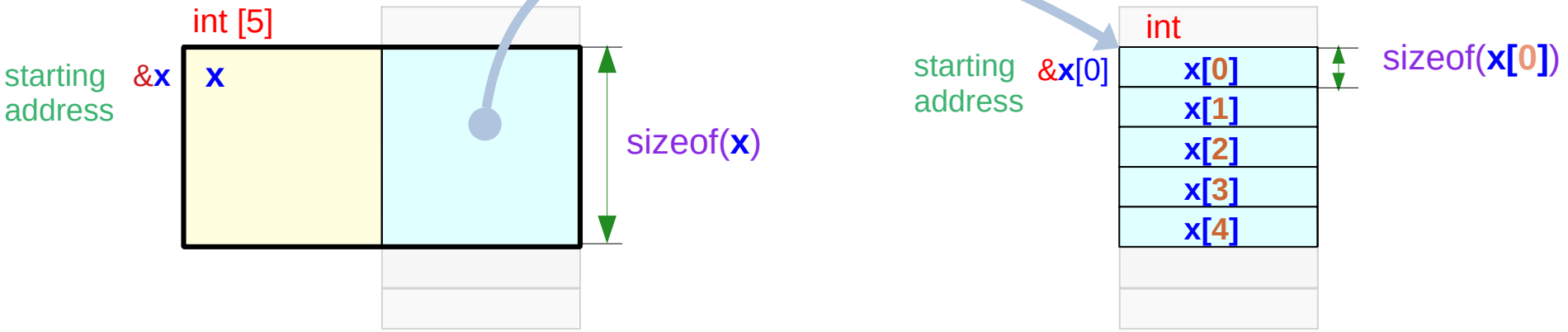
Recursive structure



int [5] type data must include
5 int elements (primitive data)

nested (recursive) structure

in a multi-dimensional array,
nested sub-arrays
instead of primitive data



abstract data x

primitive data x[i]

Pointer **x**

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

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

x[i] ≡ *(x+i)

only for i = 0, … , 4

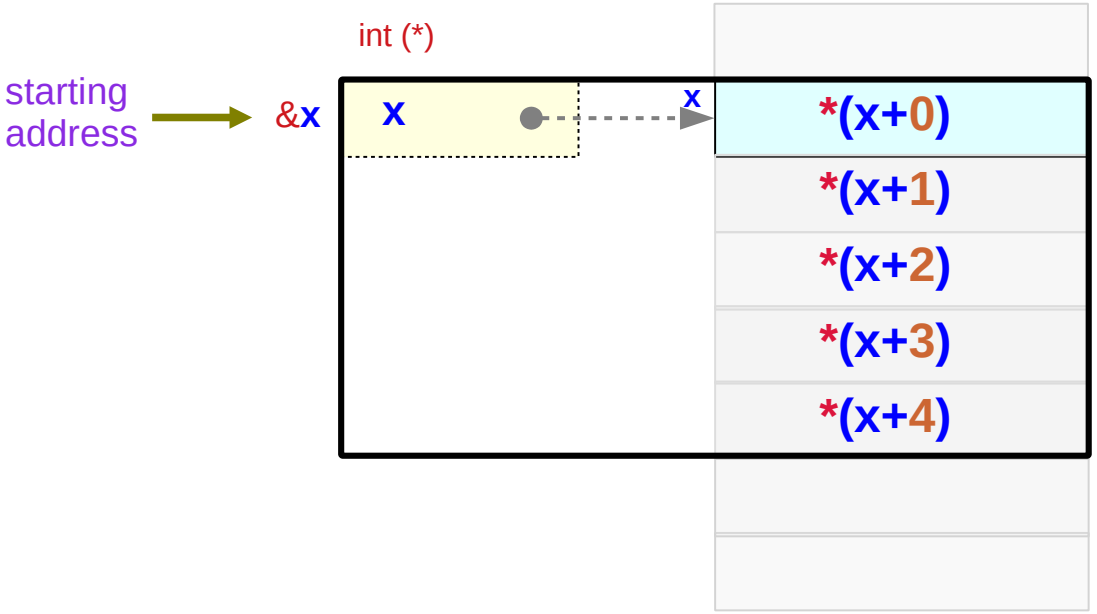
x[0] ≡ *x

&x[0] ≡ x

equivalence relations

pointer x

starting address



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

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

the starting address (&**x**) of the array **x**

value(&**x**) = value(**x**)

address replication

Aggregate Data Type – **1-d** Array

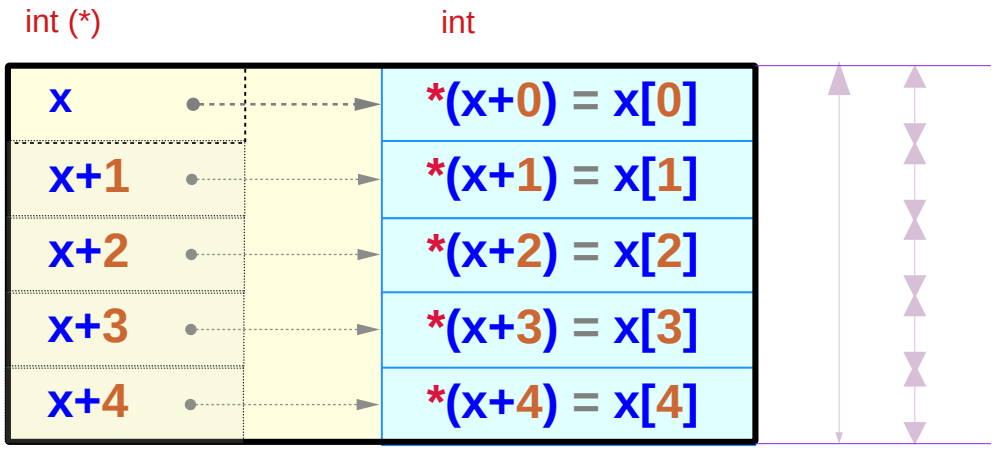
```
int     x[5];
```

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

***the start address of each element* :**

value(**x**+**0**) = value(**x**) + **0** * sizeof(***x**)
value(**x**+**1**) = value(**x**) + **1** * sizeof(***x**)
value(**x**+**2**) = value(**x**) + **2** * sizeof(***x**)
value(**x**+**3**) = value(**x**) + **3** * sizeof(***x**)
value(**x**+**4**) = value(**x**) + **4** * sizeof(***x**)

Aggregate data with 5 elements



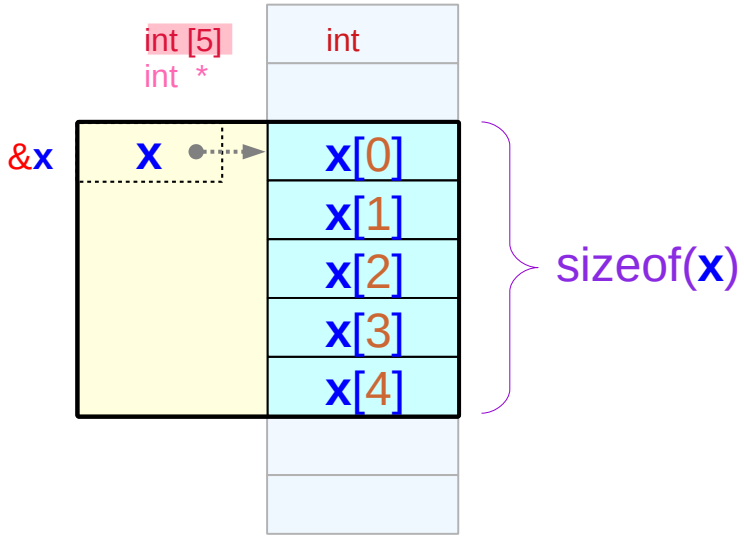
value(**x**) = ***the start address of the array***

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

- **An array x and a pointer p**

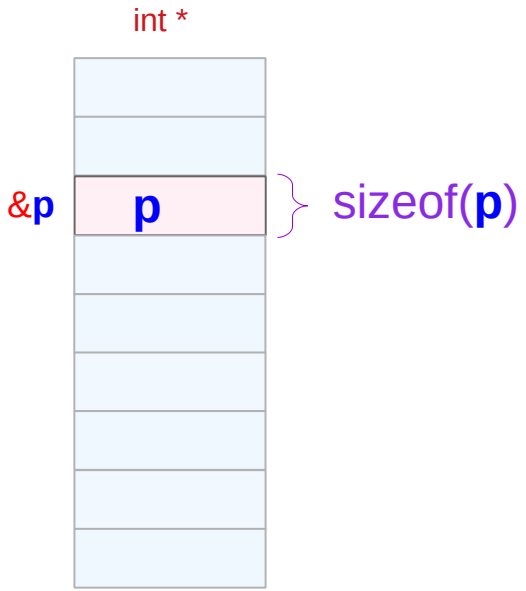
Sizes of an array **x** and a pointer **p**

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



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

```
int  * p ;
```



sizeof(**p**) = size of a pointer

= 4 bytes (32-bit system)

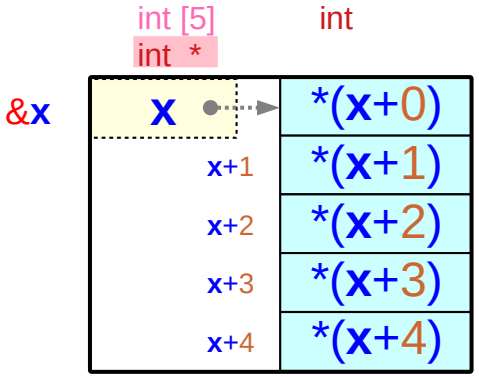
8 bytes (64-bit system)

Address values of an array **x** and a pointer **p**

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

x : an array variable name (constant)

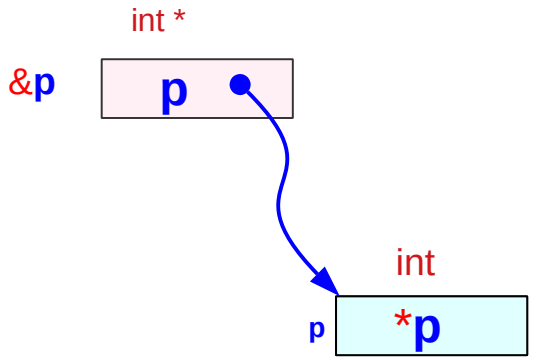
value(**x**) : the starting address of
5 consecutive **int** variables




```
int  * p ;
```

p : an variable name

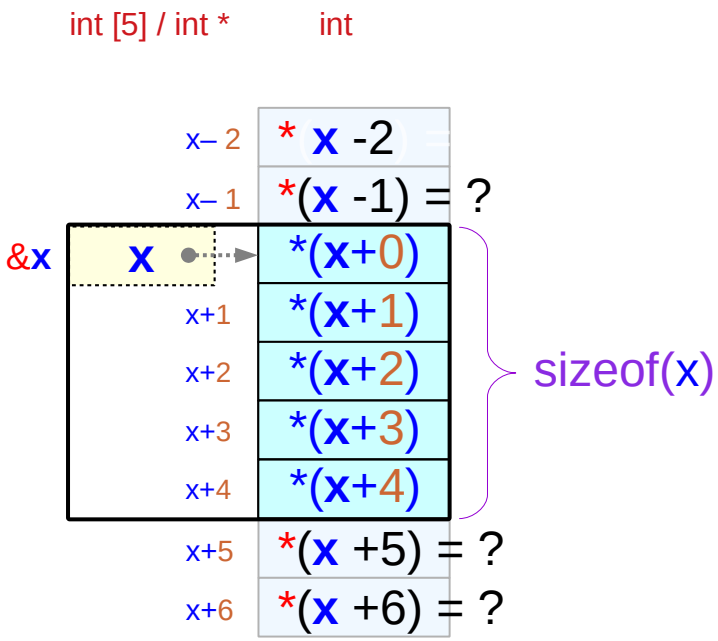
value(**p**) : the address
of an **int** variable

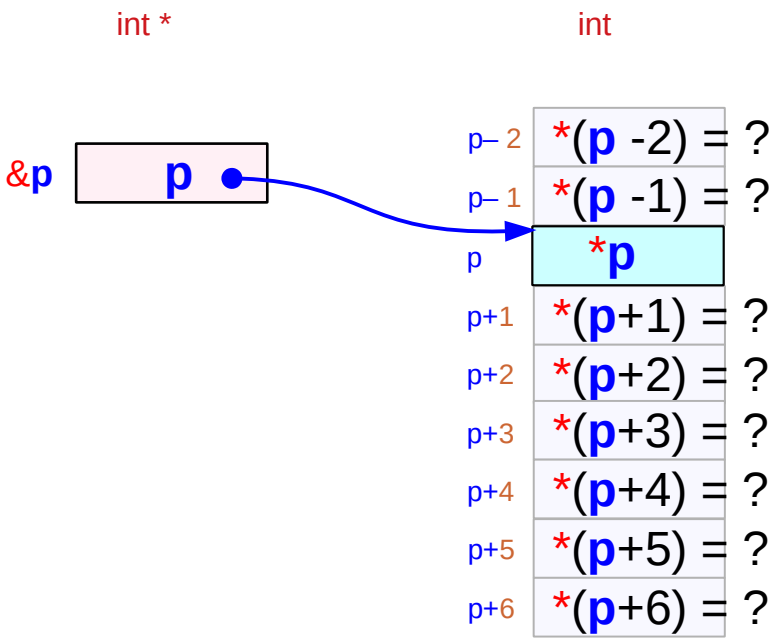


Out of range index

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

```
int  * p ;
```

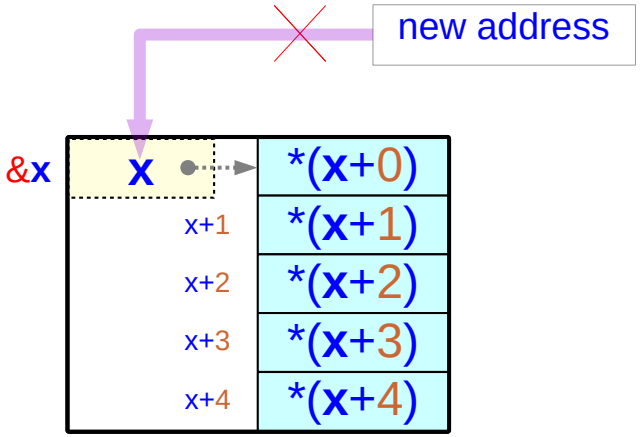




A programmer's responsibility

Assignment of a new address

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

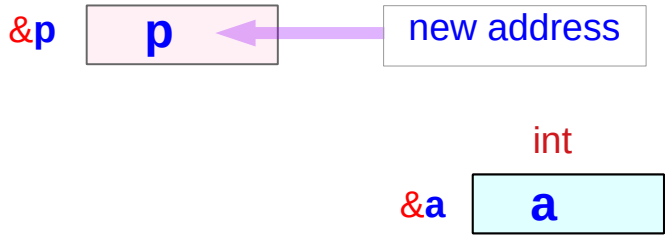


x is a constant variable
(not an *lvalue*)

x and &**x** give the same
value of &**x**[0] (address replication)

this address value(**x**) is assigned by
the compiler and cannot be changed

```
int  * p ;
```



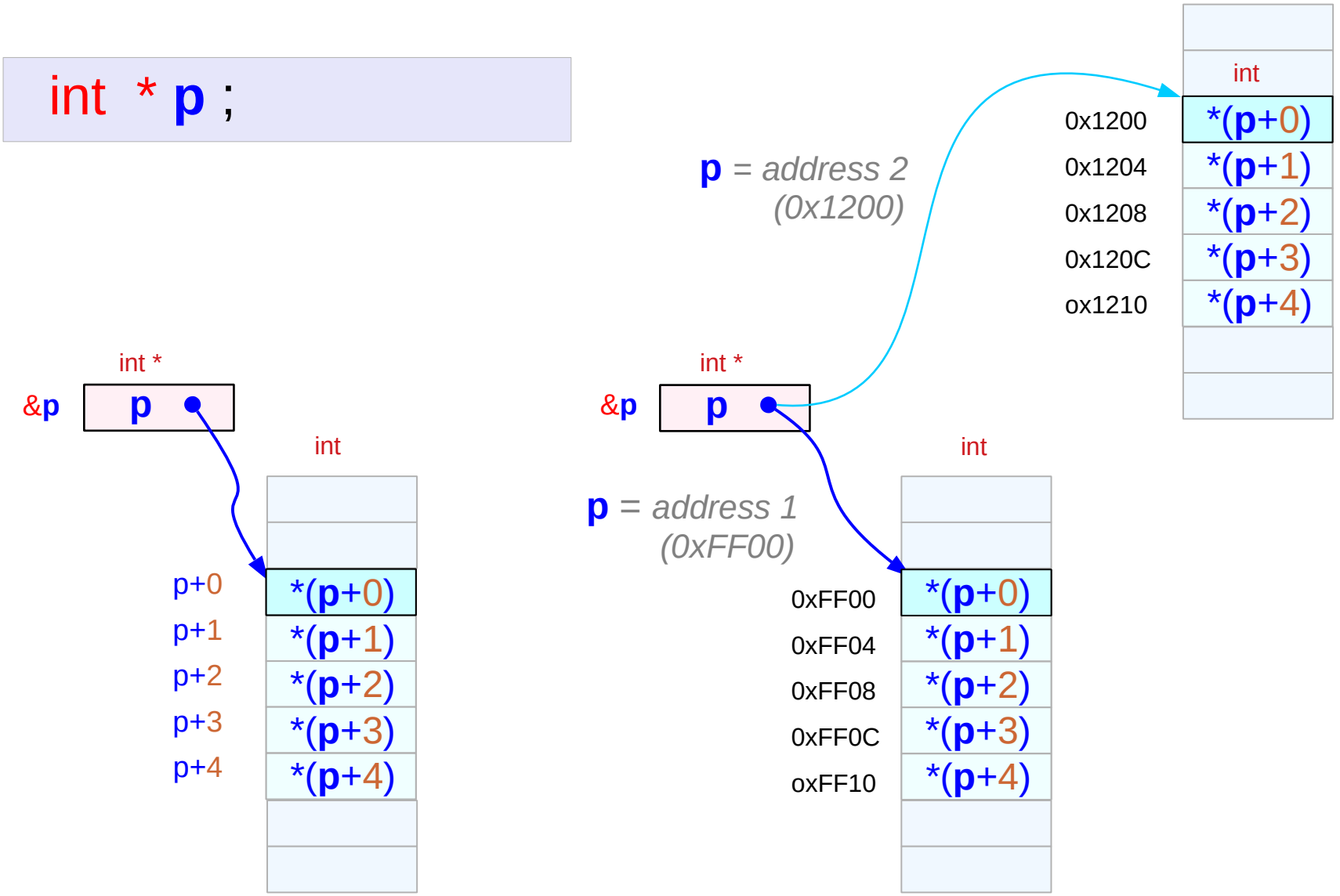
p is a pointer variable
(a *lvalue*)

a memory location is allocated
for a variable **p**

value(**p**) can be changed by an
assignment statement, i.e,
p = &**a**

Pointer variable can point different locations

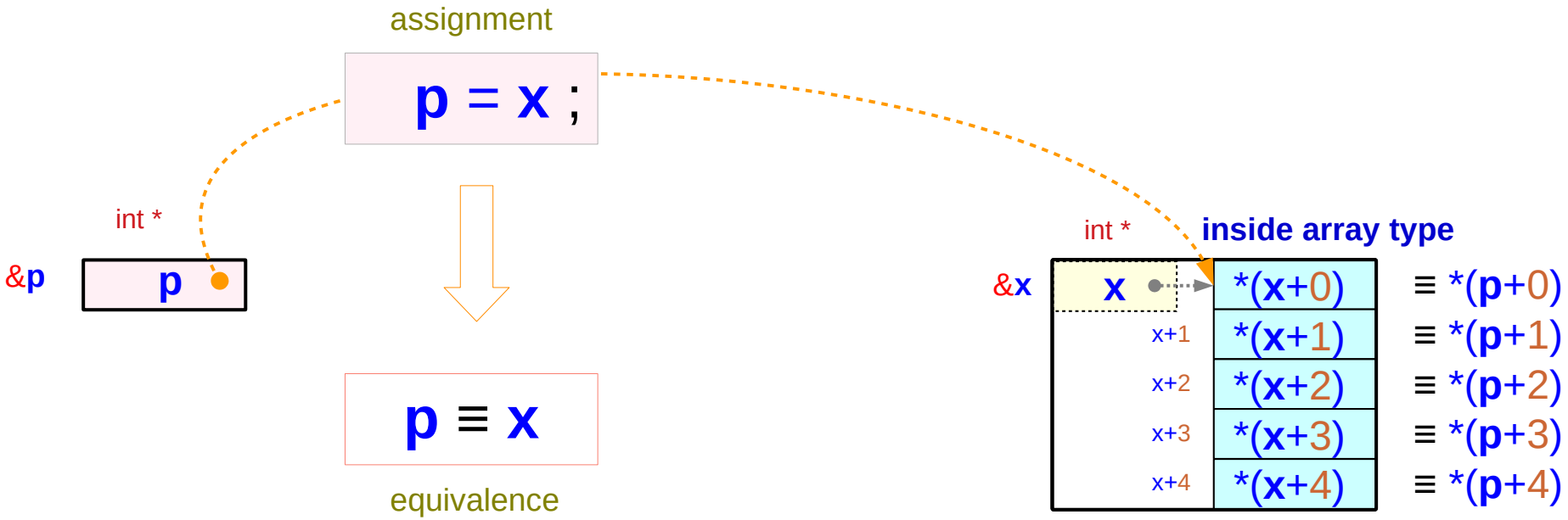
```
int * p ;
```



Pointer to an array element

```
int  (*p) ;
```

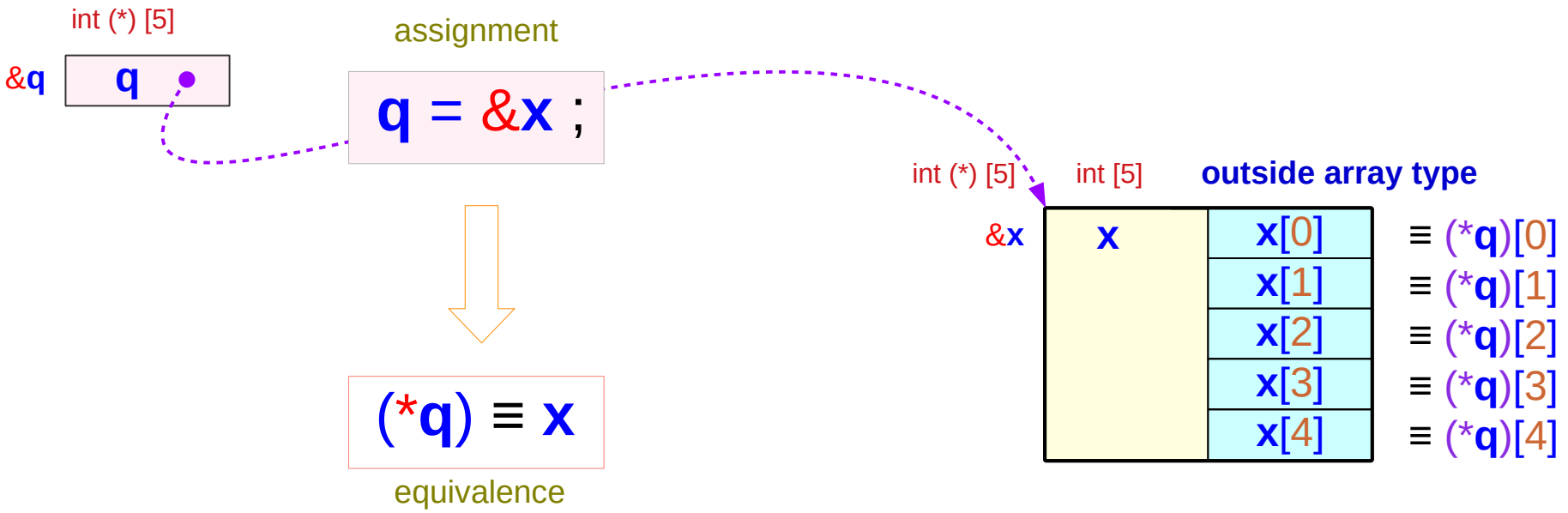
```
int x [5] ;
```



Pointer to an array

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

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



q is a pointer variable that points to an array with 5 integer variables

Pass by Reference

Variable Scopes

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


   ...
   ...
}
```

i and j's
variable scope

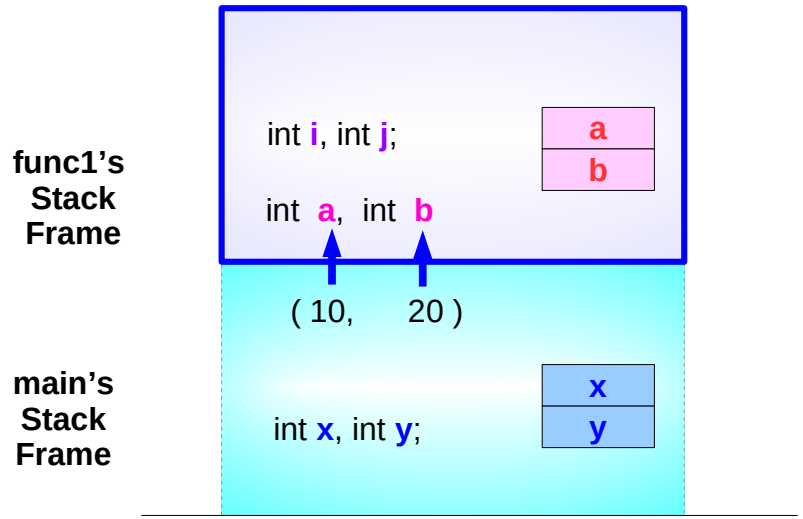
cannot access
each other

```
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;
   ...
   ...


   ...
   ...
}
```

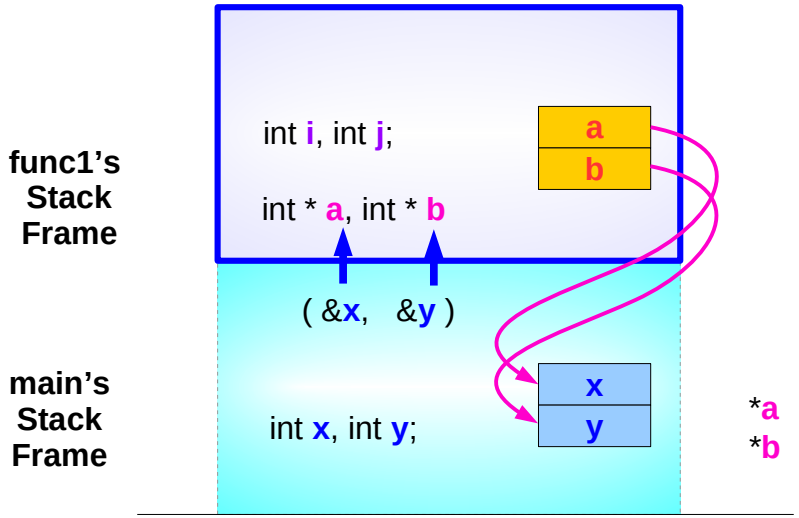
```
int main ()
{
   int x, int y;
   ...
   ...

   func1 ( &x, &y );

   ...
   ...
}
```

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

*a
*b

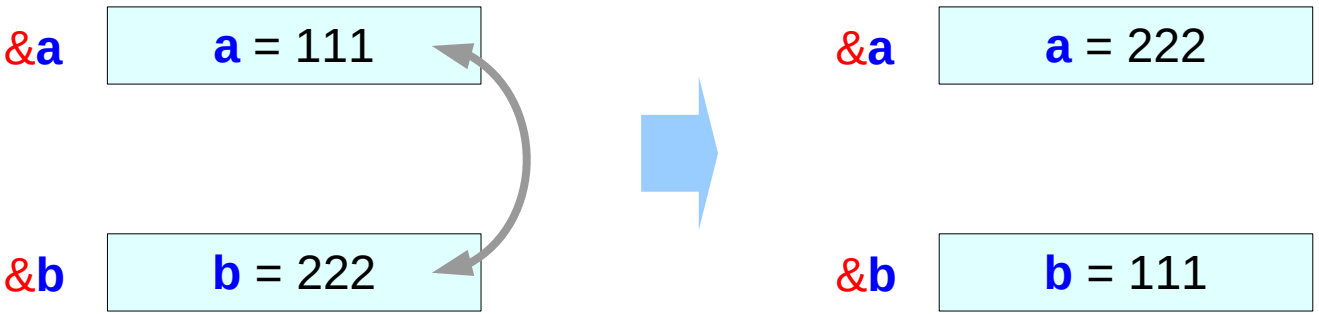
a = &x
b = &y

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;
}
```

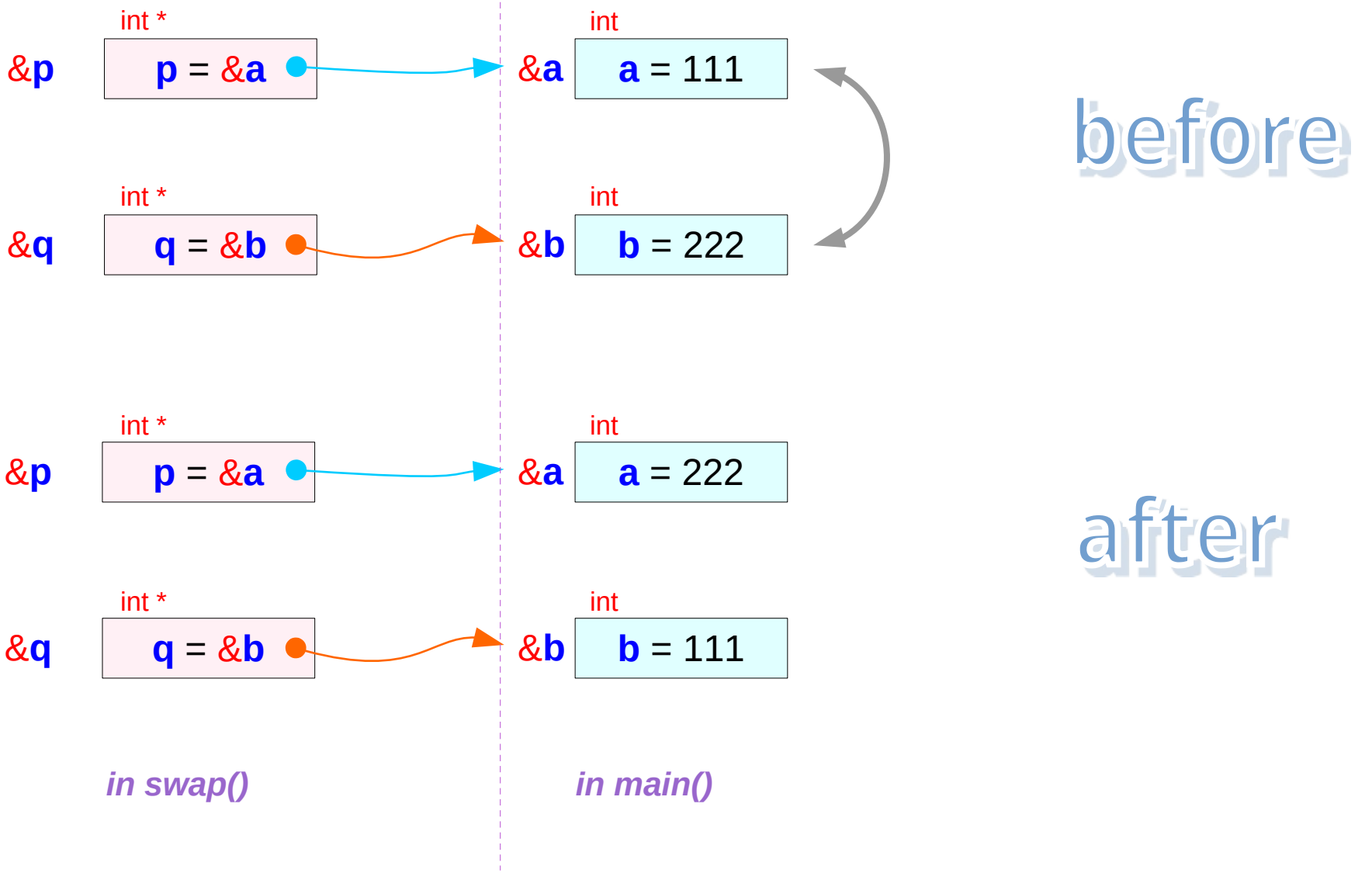
int *	p
int	*p

int *	q
int	*q

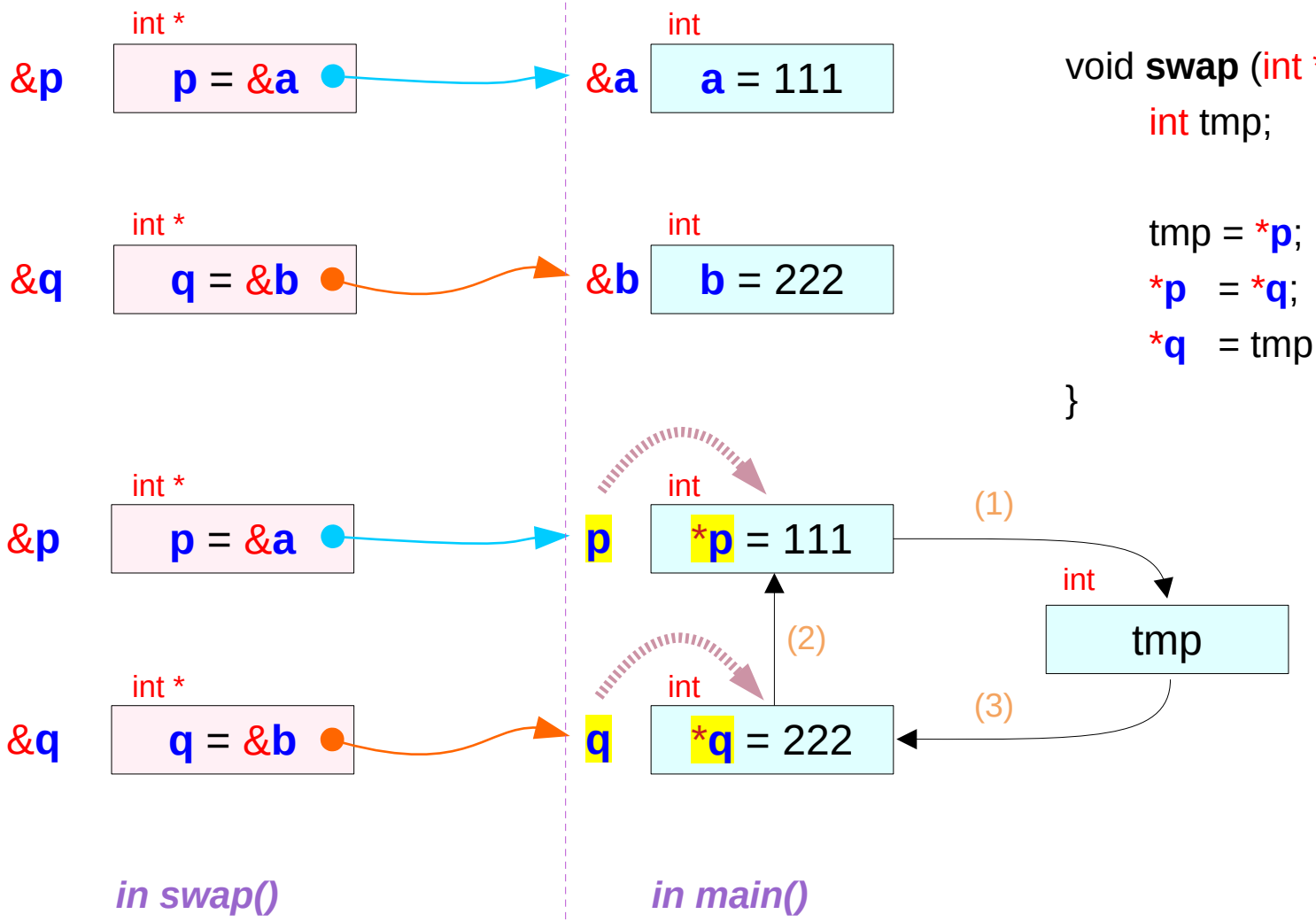
int	tmp

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

    …
swap( &a, &b );
```

Swapping integers

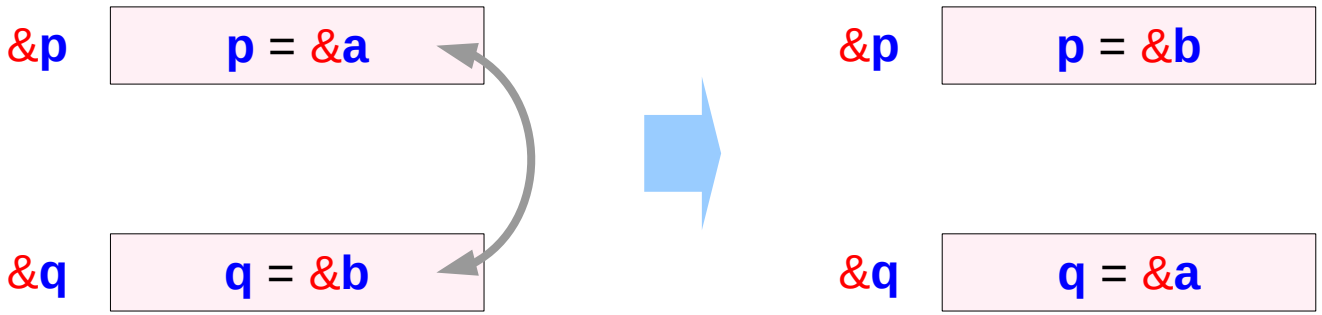


Swapping integers via integer pointers




```
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;
}
```

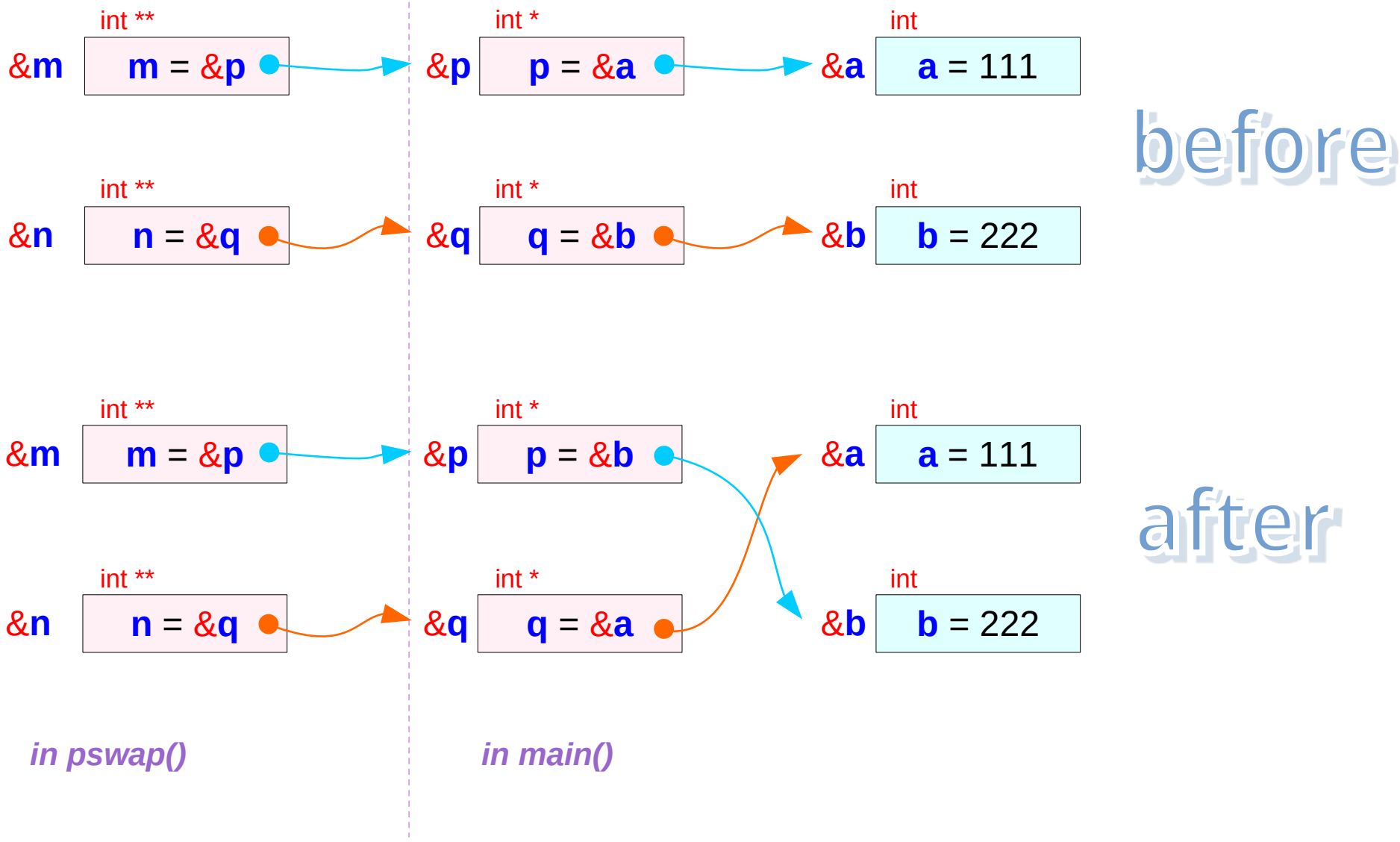
int **	m	int **	n
int *	*m	int *	*n
int	**m	int	**n

int *	tmp

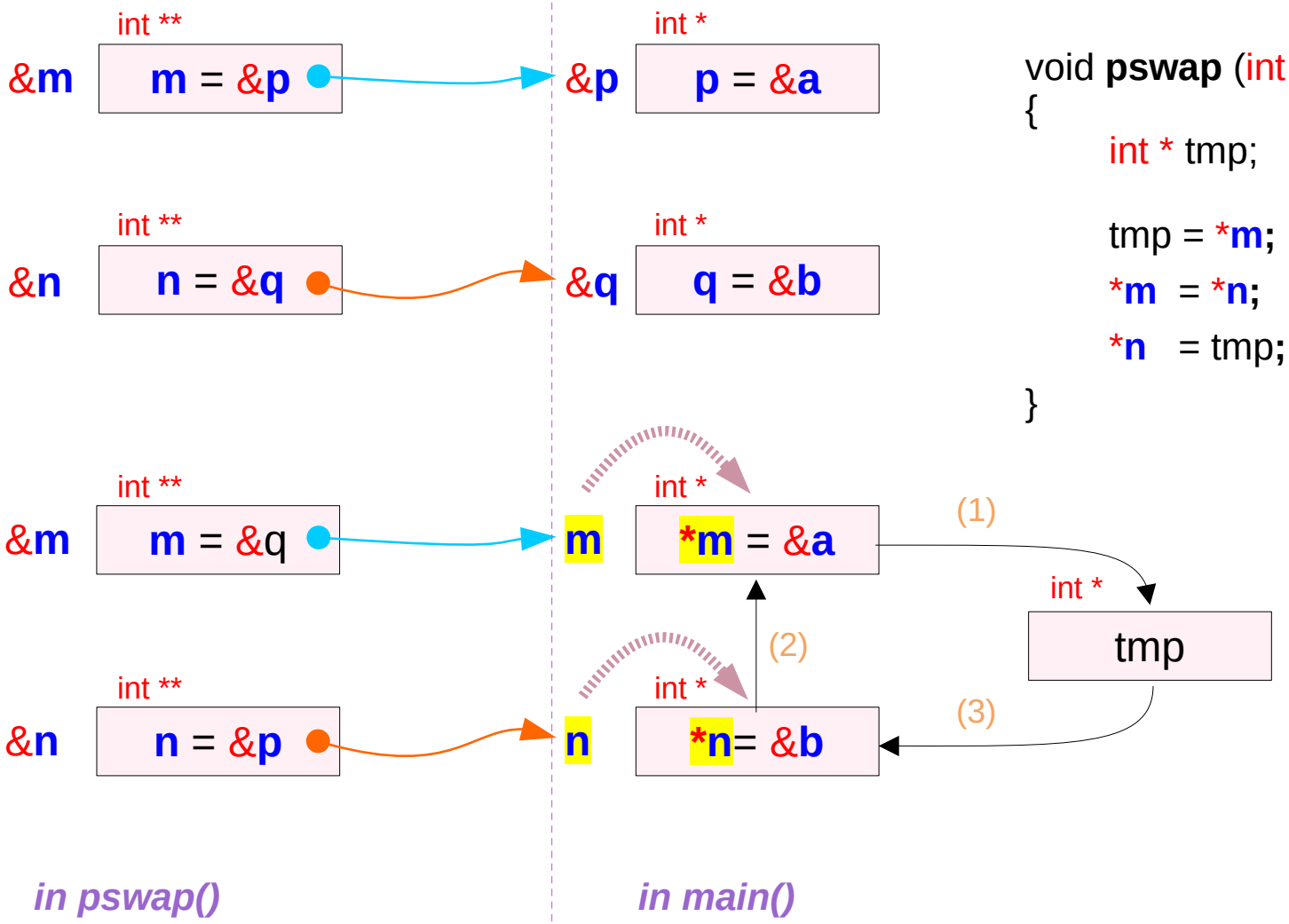
```
int    a = 111,  b = 222;
int  *p = &a, *q = &b;
       …
pswap ( &p, &q );
```

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)
}
```

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