Pointers (1A)

Young Won Lim
7/26/18

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

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

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

This document was produced by using LibreOffice.

Variables

```
int     a;
```

a can hold an *integer* value

address	data
&a	a

```
a = 100;
```

a holds the *integer* 100

address	data
&a	a ⬅ 100

Pointer Variables

```
int *        p;
```

p holds an *address*

p can hold the *address* of an int data

*p can hold an *integer* value

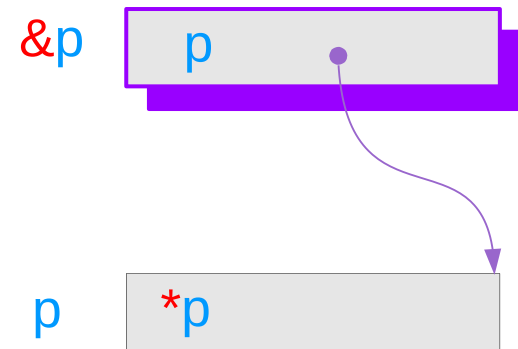
type ***variable***

int * p ;

pointer to int

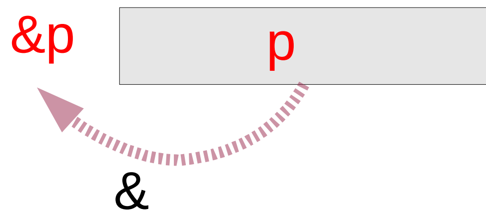
int * p ;

int

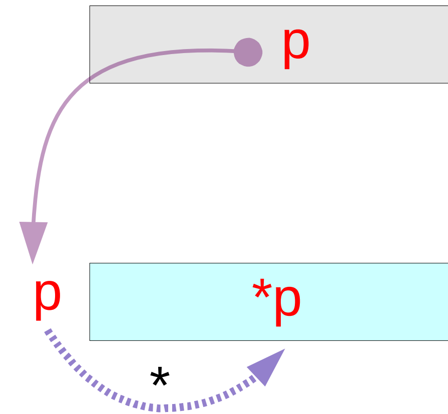


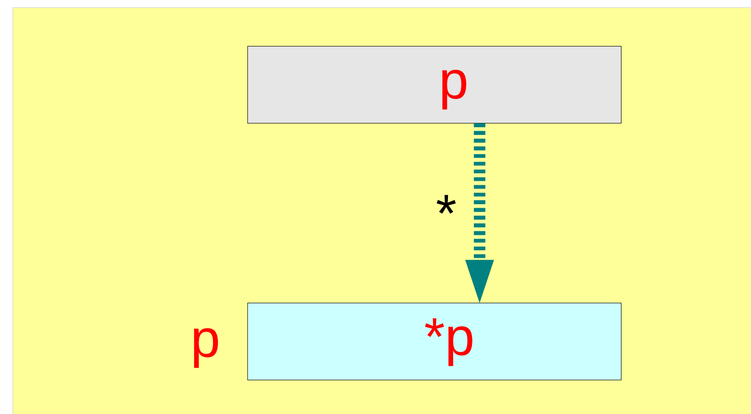
Dereferencing

The address of a variable :
Address of operator &



The content of a pointed location :
*Dereferencing operator **



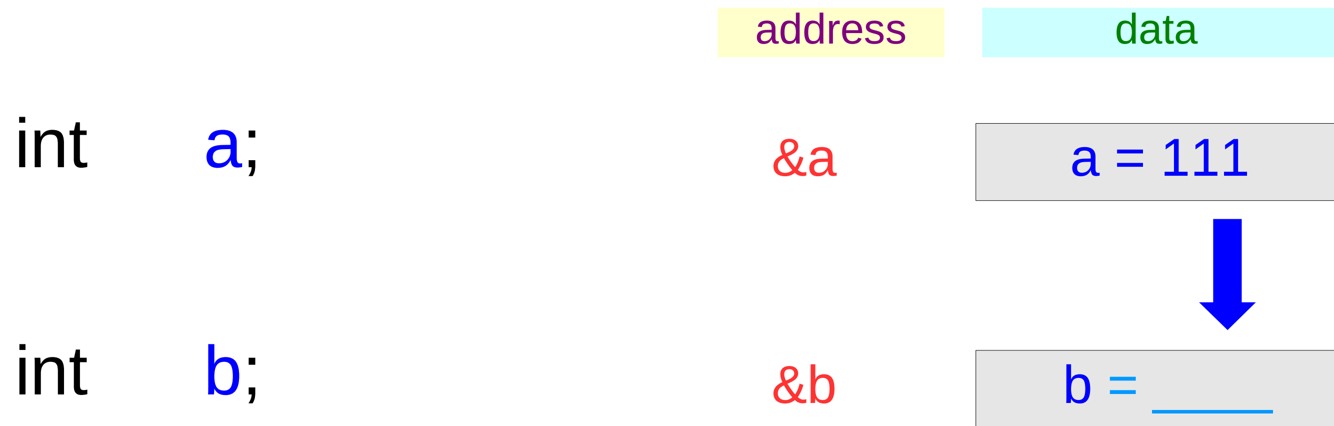


Variables and their addresses

```
int   a;

int * p;
```

address	data
&a	a
&p	p

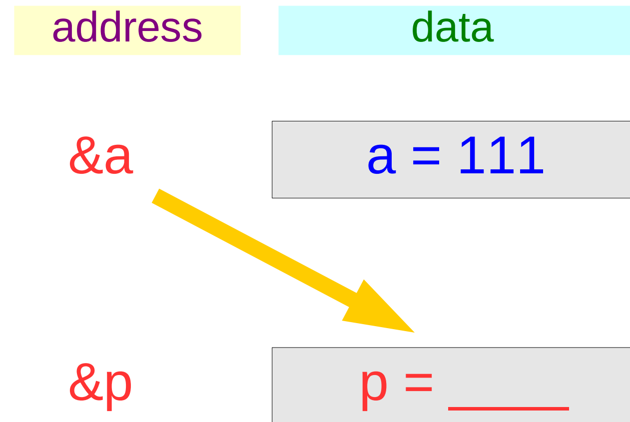
Assignment of a value



b = a;

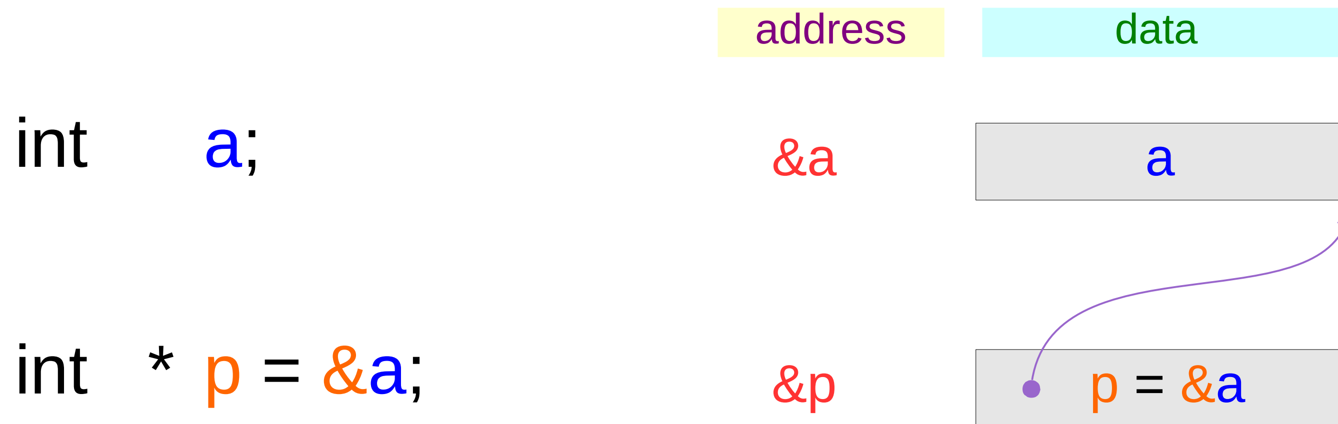
Assignment of an address

```
int     a;

int   * p;
```



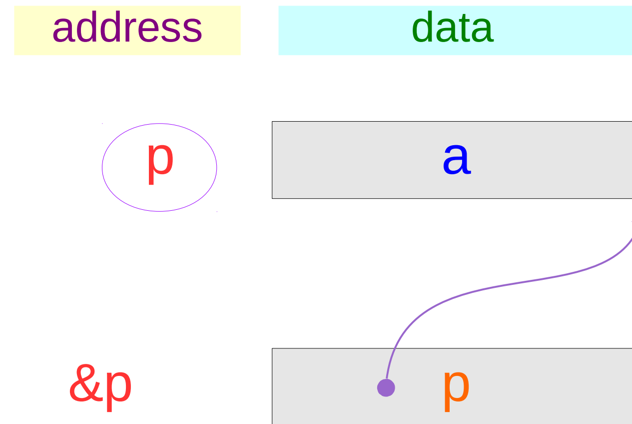

```
p = &a;
```

Variables with initializations



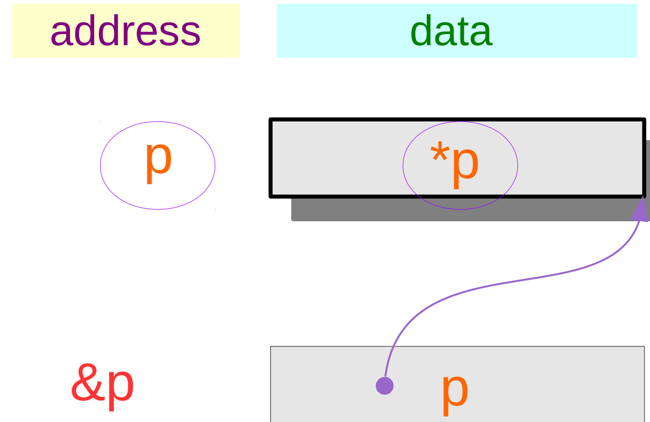
Pointed addresses : p

```
int     a;

int   * p = &a;
```



$p \equiv \&a$

Dereferenced Variable : *p



int a;

int * p = &a;

assignment

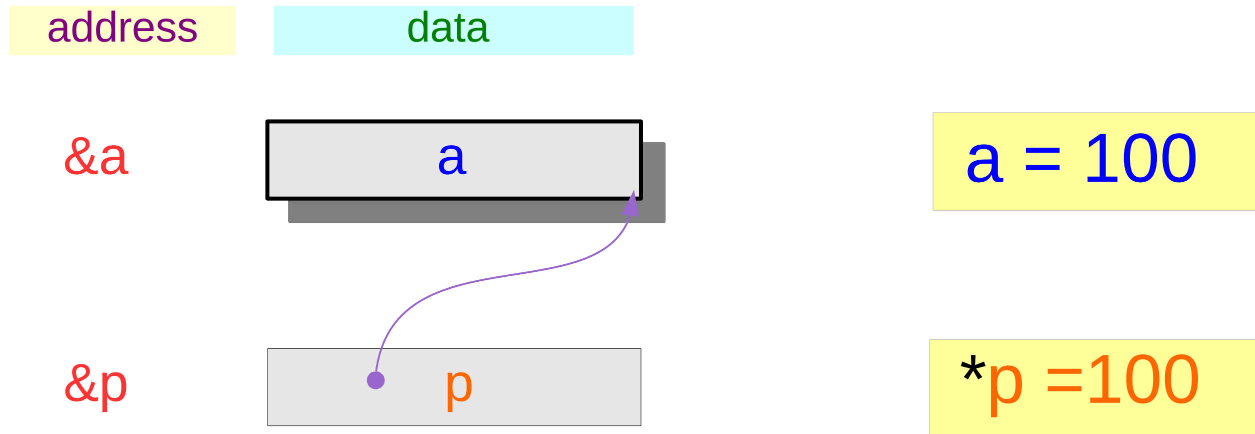
equivalence

p ≡ &a

*p ≡ *&a

*p ≡ a

Two way to access: a and *p



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

Double Pointers

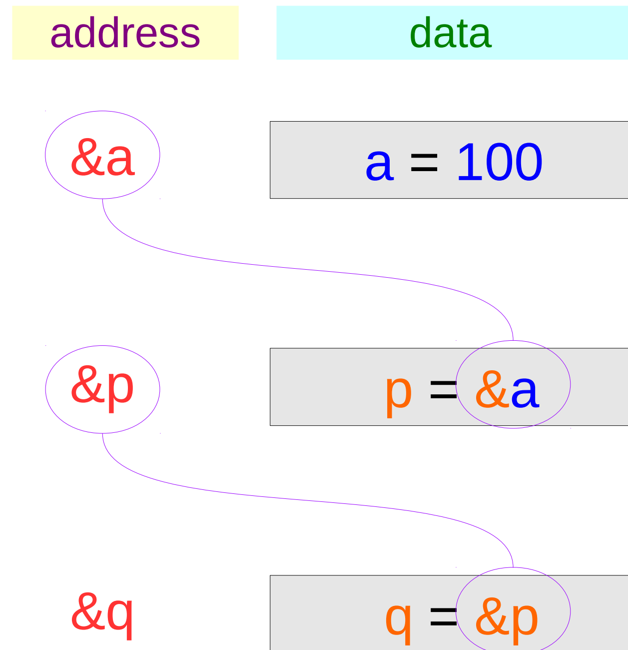
Variables and their addresses

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

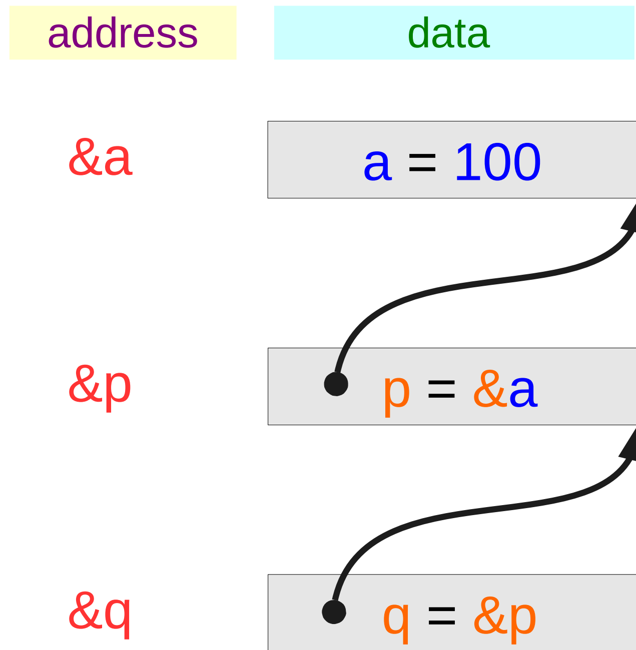
address	data
&a	a
&p	p
&q	q

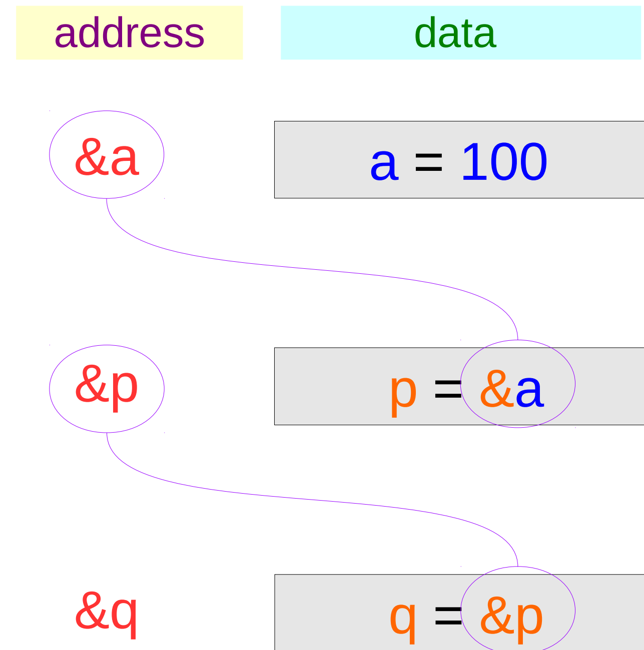
Initialization of Variables

```
int      a = 100;

int    * p = &a;

int   ** q = &p;
```



Traditional arrow notations

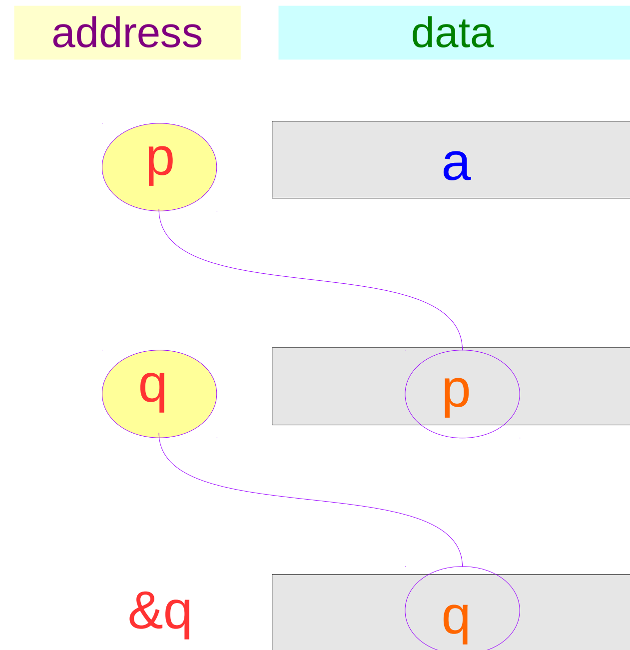




LSB, little endian

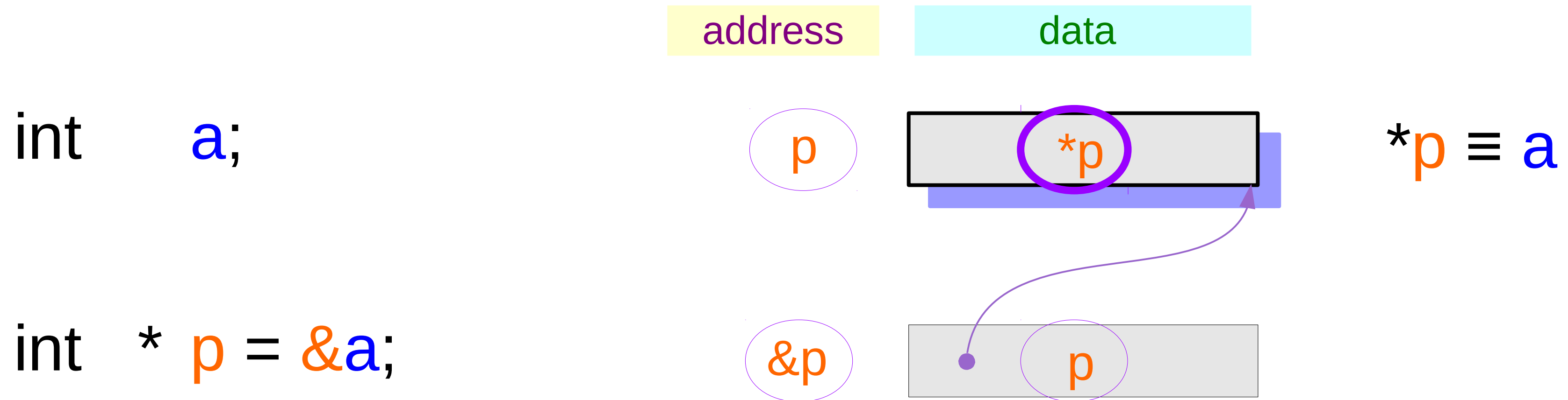
Pointed addresses : p, q

```
int     a;

int   * p = &a;

int  ** q = &p;
```



p = &a
q = &p

A dereferenced variable : *p



An aliased variable : *p

```
int    a;

int  * p = &a;
```

Address assignment

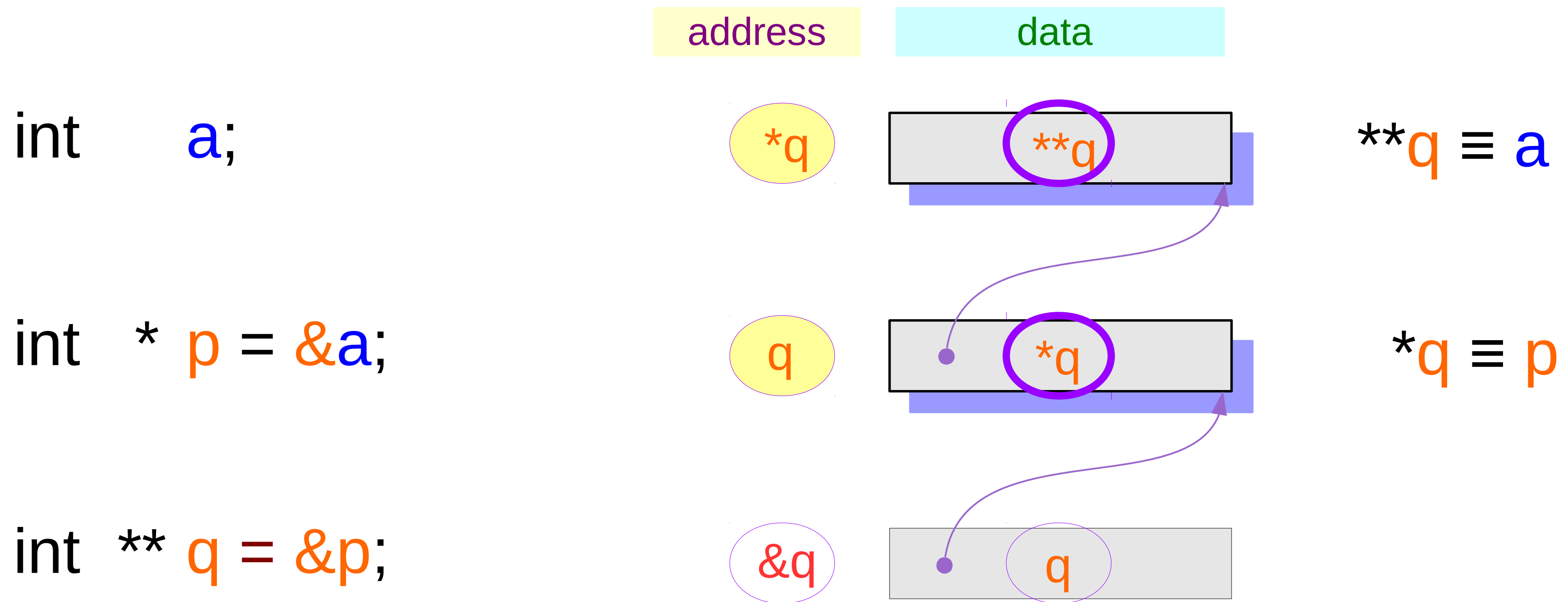
Variable aliasing

p = &a ➡ *p ≡ a

p	≡	&a
*(p)	≡	*(&a)
* p	≡	a

equivalent relations after address assignment

Dereferenced variables : *q, **q



Aliased variables : *q, **q

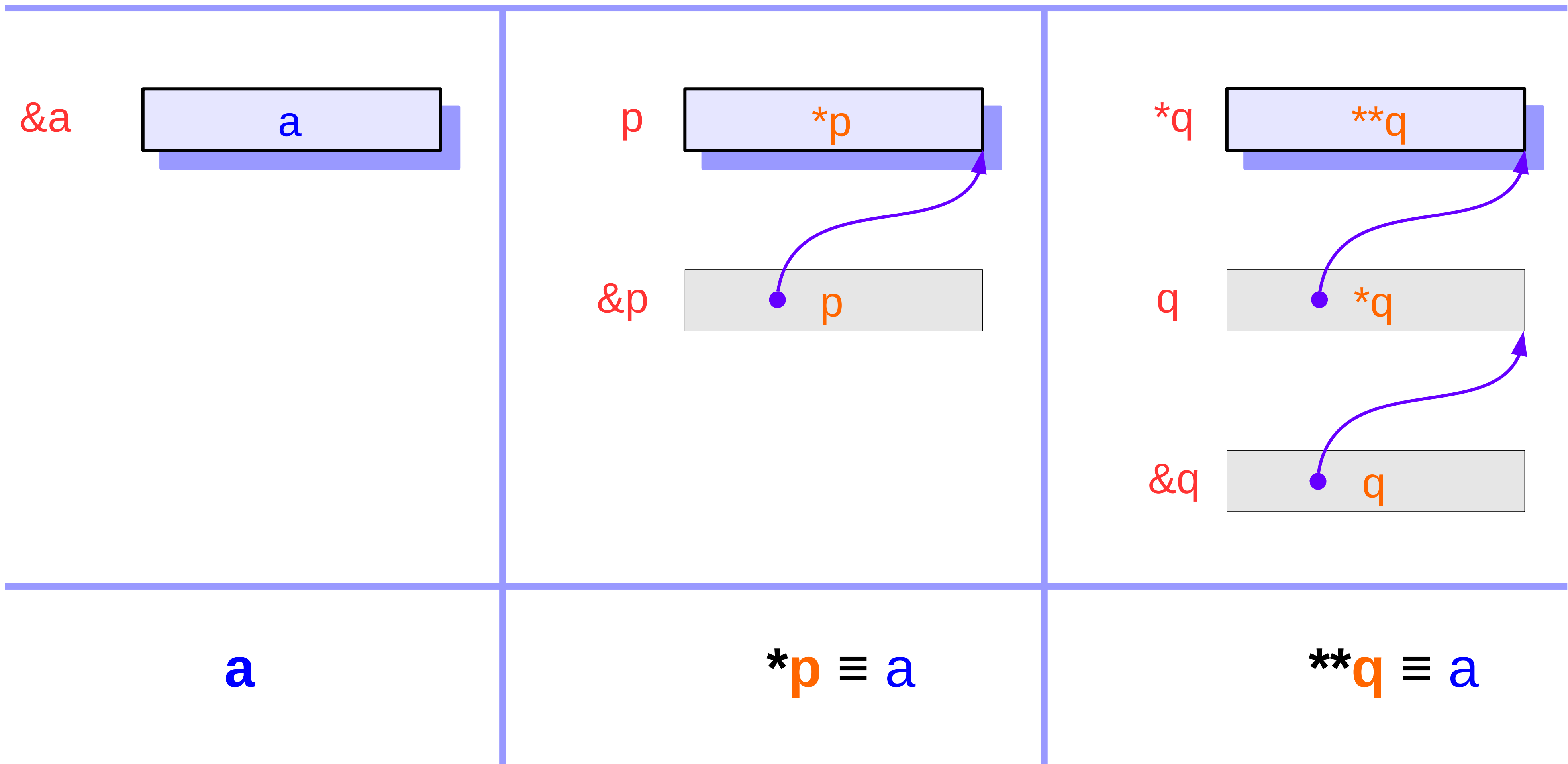
```
int    a;
int  * p = &a;
int ** q = &p;
```

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

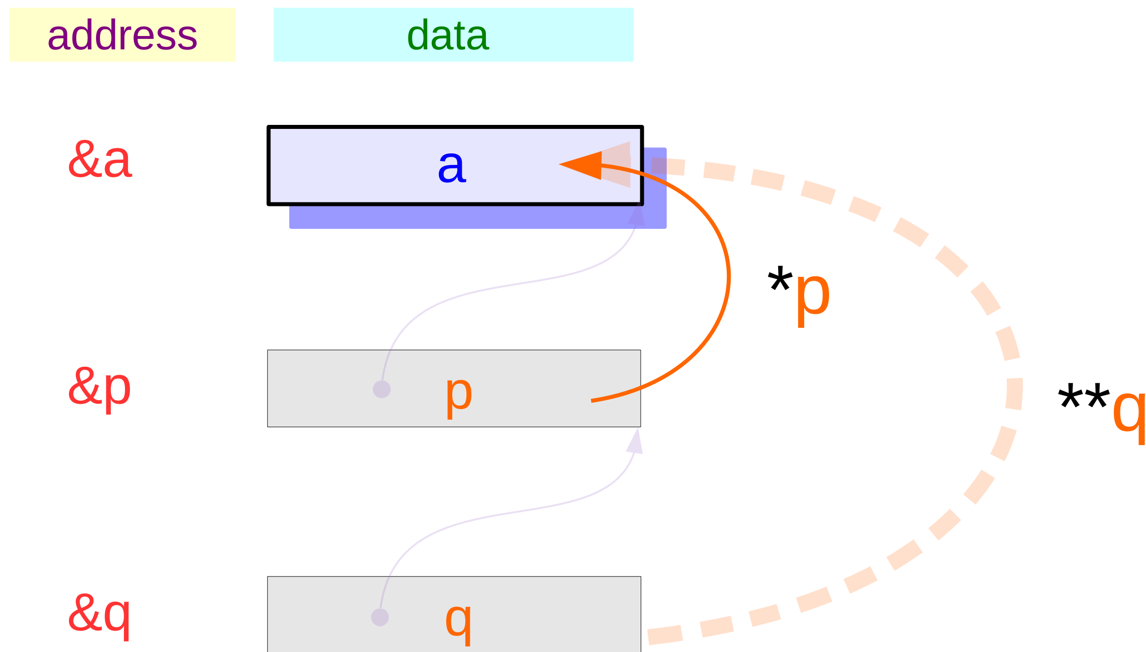
q ≡ &p
*(q) ≡ *(&p)
* q ≡ p
**q ≡ *p
**q ≡ a

equivalent relations after address assignment

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



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



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

Variable Definitions

```
int     a;
```

a can hold an *integer*

address	data
&a	a

```
a = 100;
```

a holds 100

address	data
&a	a ⬅ 100

Pointer Variable Definition

int *   p;

p can hold an ***address***

int * | p;

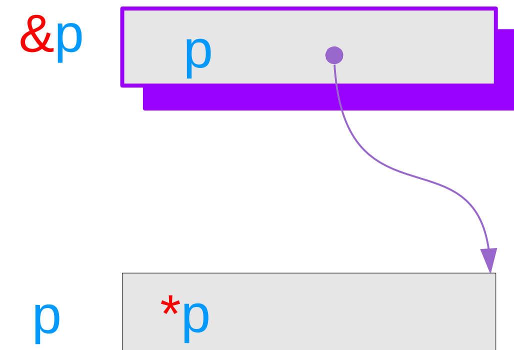
pointer to int

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

int | * p;

int

p** holds a int type ***data



Double Pointer Variable Definition

```
int **      q;
```

q holds an *address*

int **	q;

pointer to pointer to int

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

int *	*q;

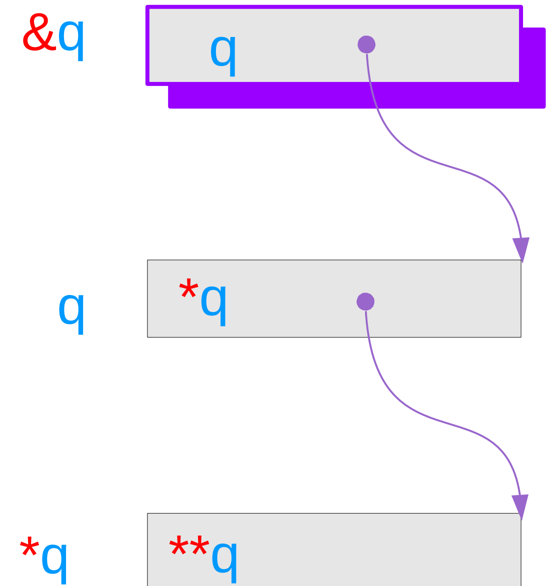
pointer to int

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

int	**q;

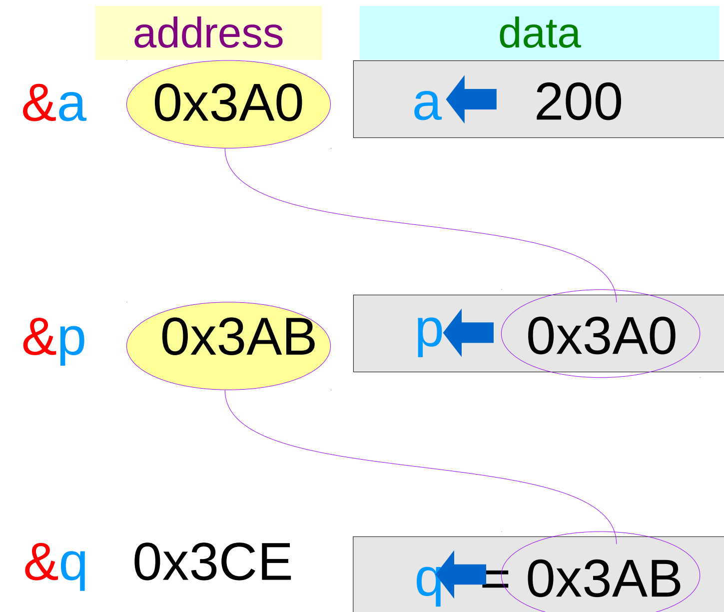
int

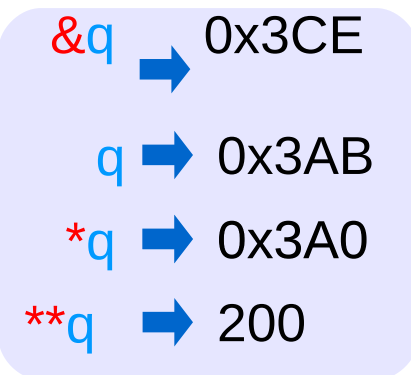
****q** holds a int type ***data***



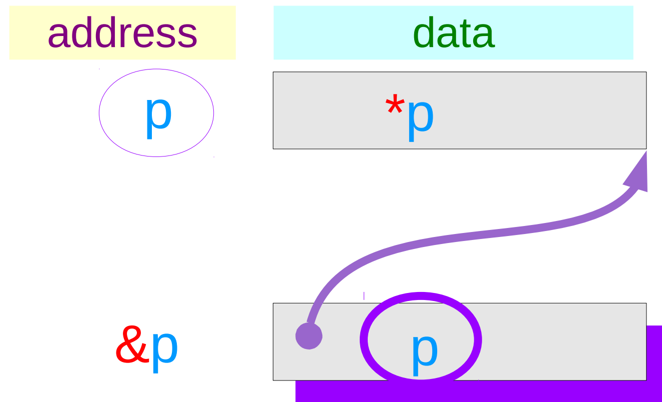
Pointer Variable Examples

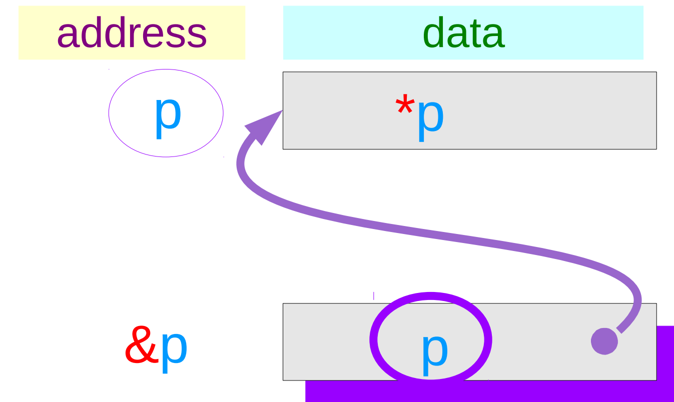
```
int     a = 200;

int *   p = & a;

int **  q = & p;
```





Arrow notations





using an arrow notation (I)

using an arrow notation (II)

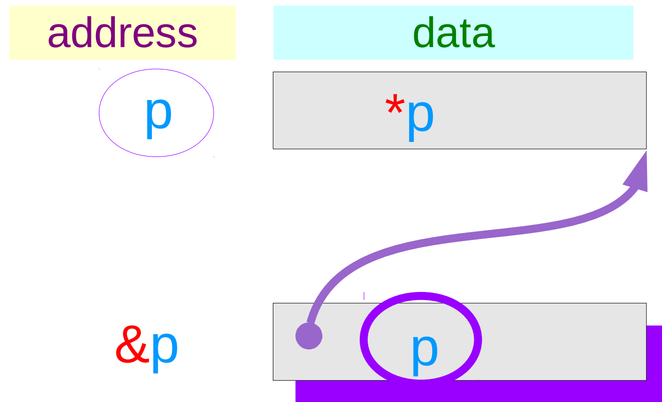
MSB LSB

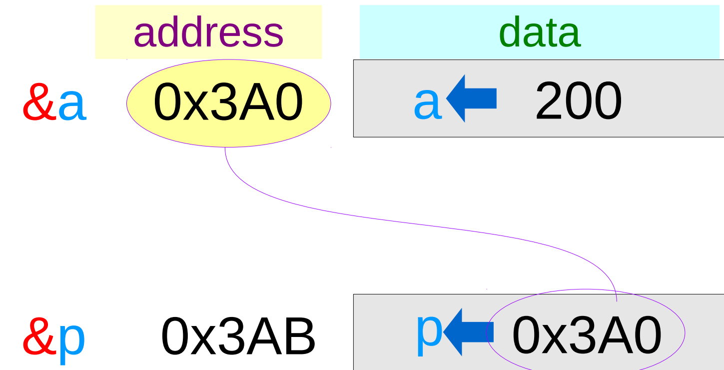
Simplified Abstract Drawing

Little Endian Assumed

Familiar, Well known

Pointer Variable **p** with an arrow notation





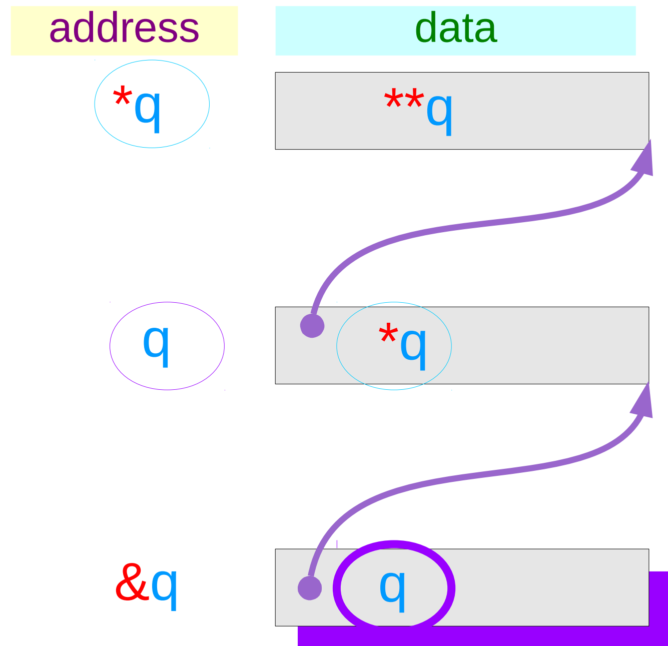
using an arrow notation

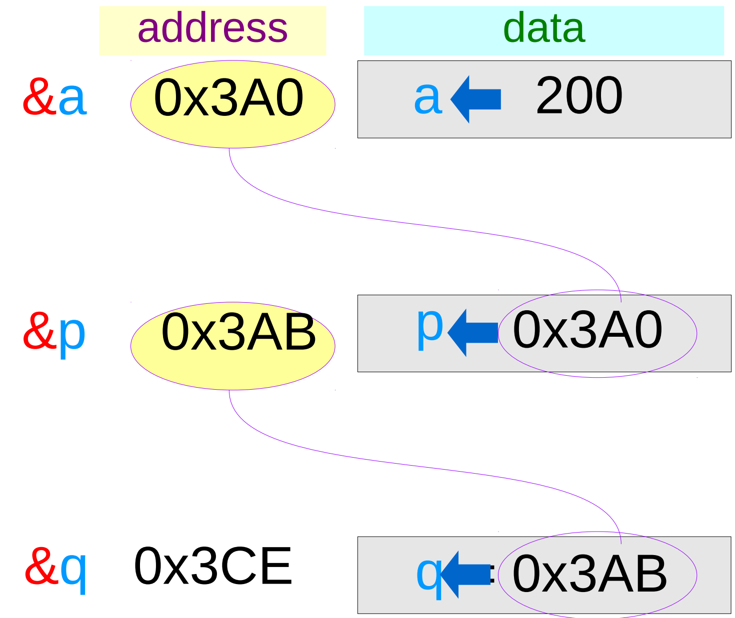
&p → 0x3AB

p → 0x3A0

*p → 200

Pointer Variable **q** with an arrow notation

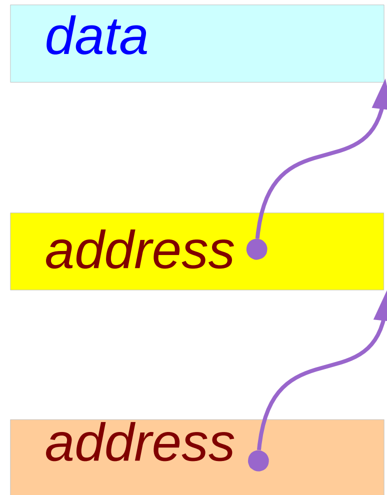


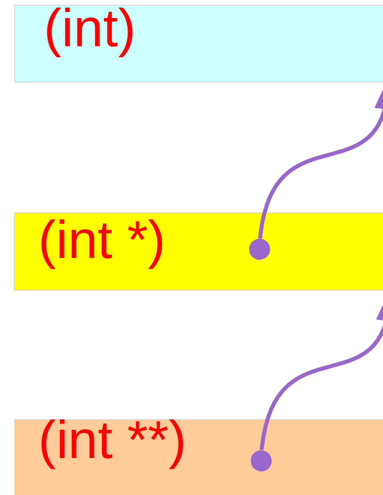


using an arrow notation

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

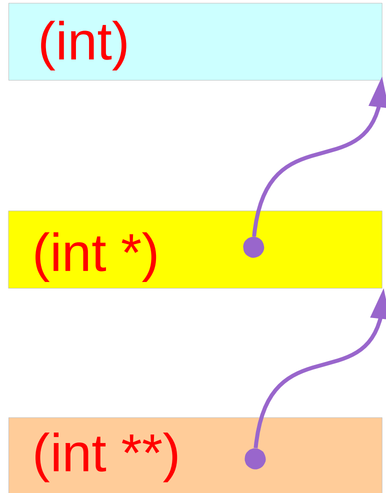
Pointers – a type view

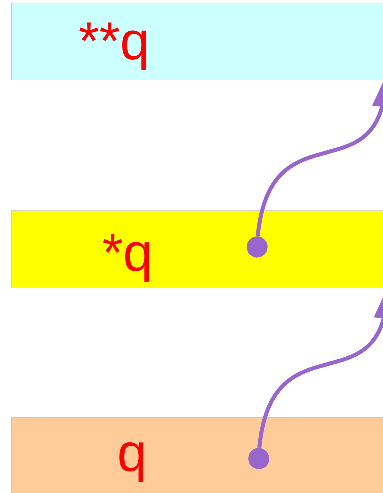


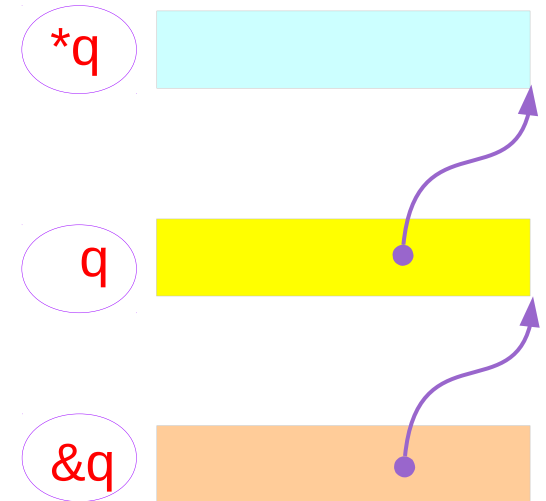


Types

Pointers – other view



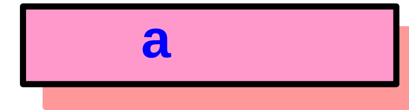


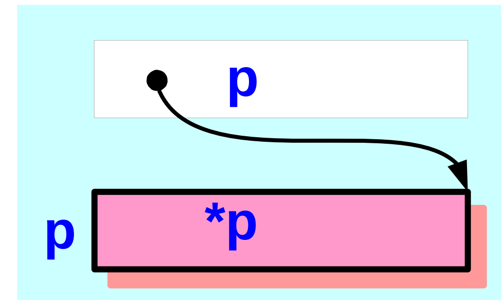


Types **Variables** **Addresses**

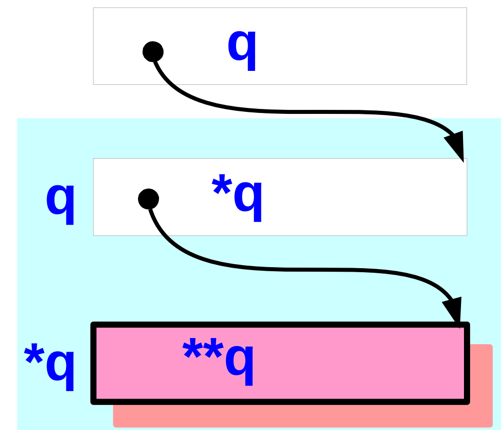
Single and double pointer examples (1)

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



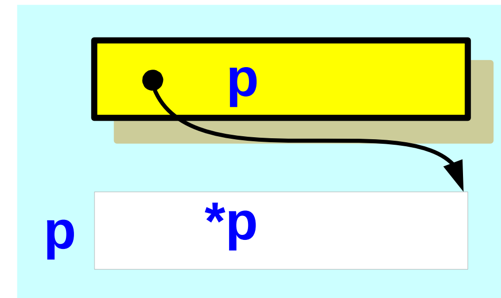


a, ***p**, and ****q**:
int variables

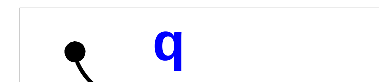


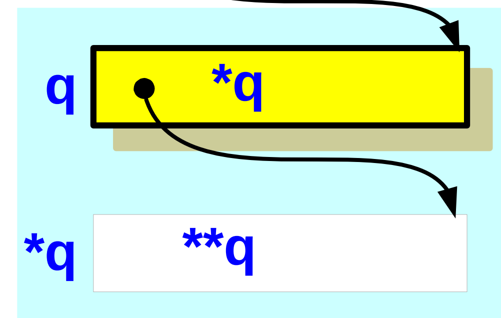
Single and double pointer examples (2)

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



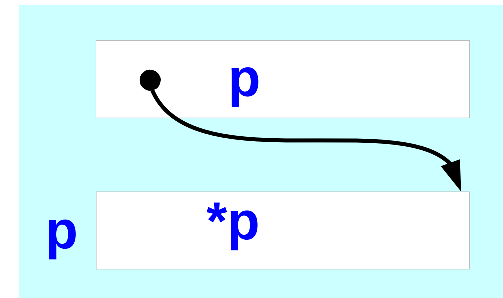
p and *__q__ __:__
int __pointer__ variables
(singlepointers)



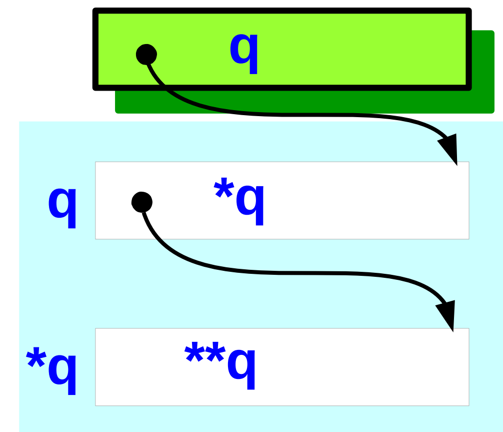


Single and double pointer examples (3)

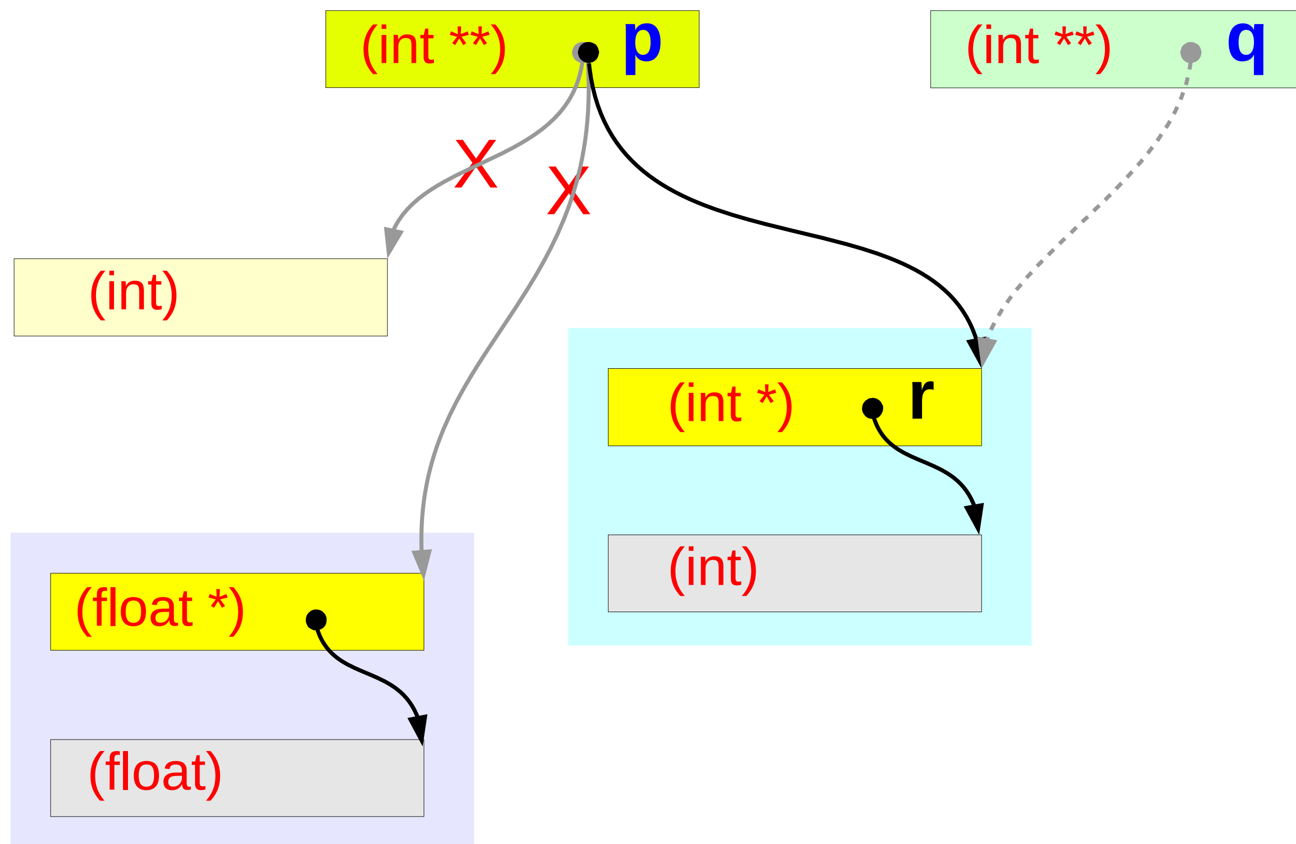
```
int      a ;
int *    p ;
int **   q ;
```



q :
double int pointer variables



Double pointer variable assignments



Pointed Addresses and Data

int a ;   &a   a =100

The variable a holds an integer data

int * p ;   &p   p → 200

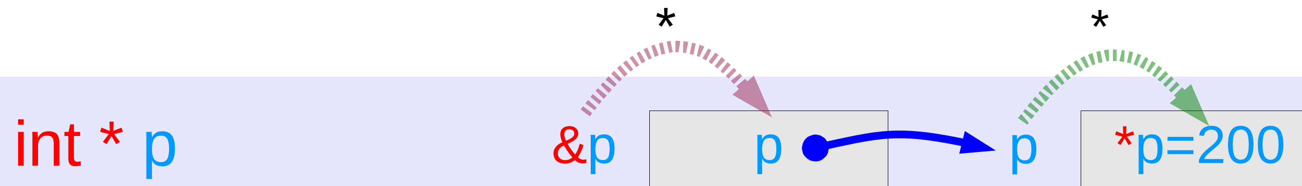
The **pointer** variable p holds an address,
at this address, an integer data is stored

int * * q ;   &q   q → *q → 30

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

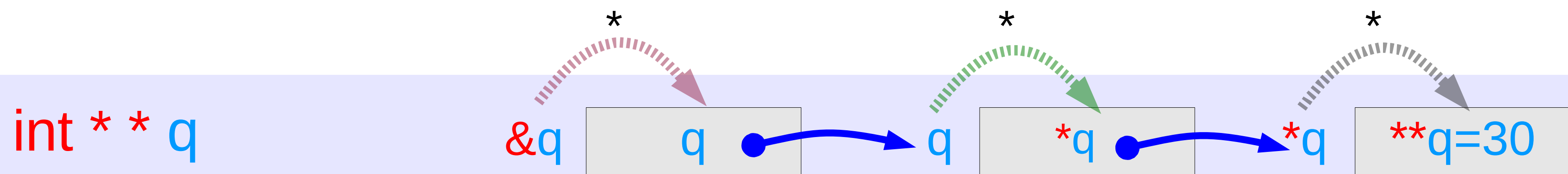
Dereferencing Operations

*(&a) = a



*(&p) = p

*(p) = *p



*(&q) = q

*(q) = *q

*(*q) = **q

Direct access to an integer **a**

```
int a ;
```

&a | a =100

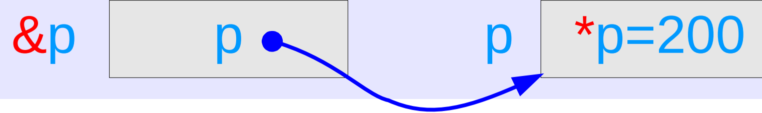
Direct Access

address	value	
&a	a	integer

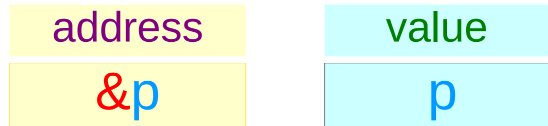
1 memory access

Indirect access *p to an integer a

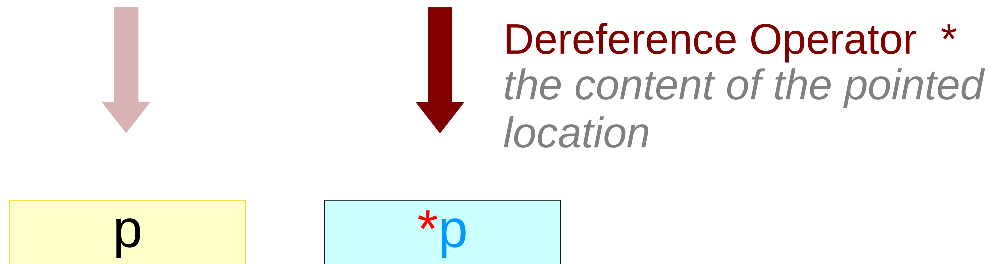
int * p ;



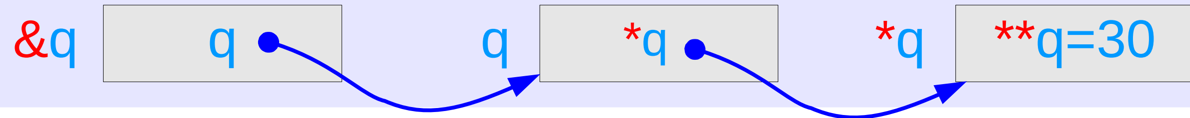
Indirect Access



2 memory accesses

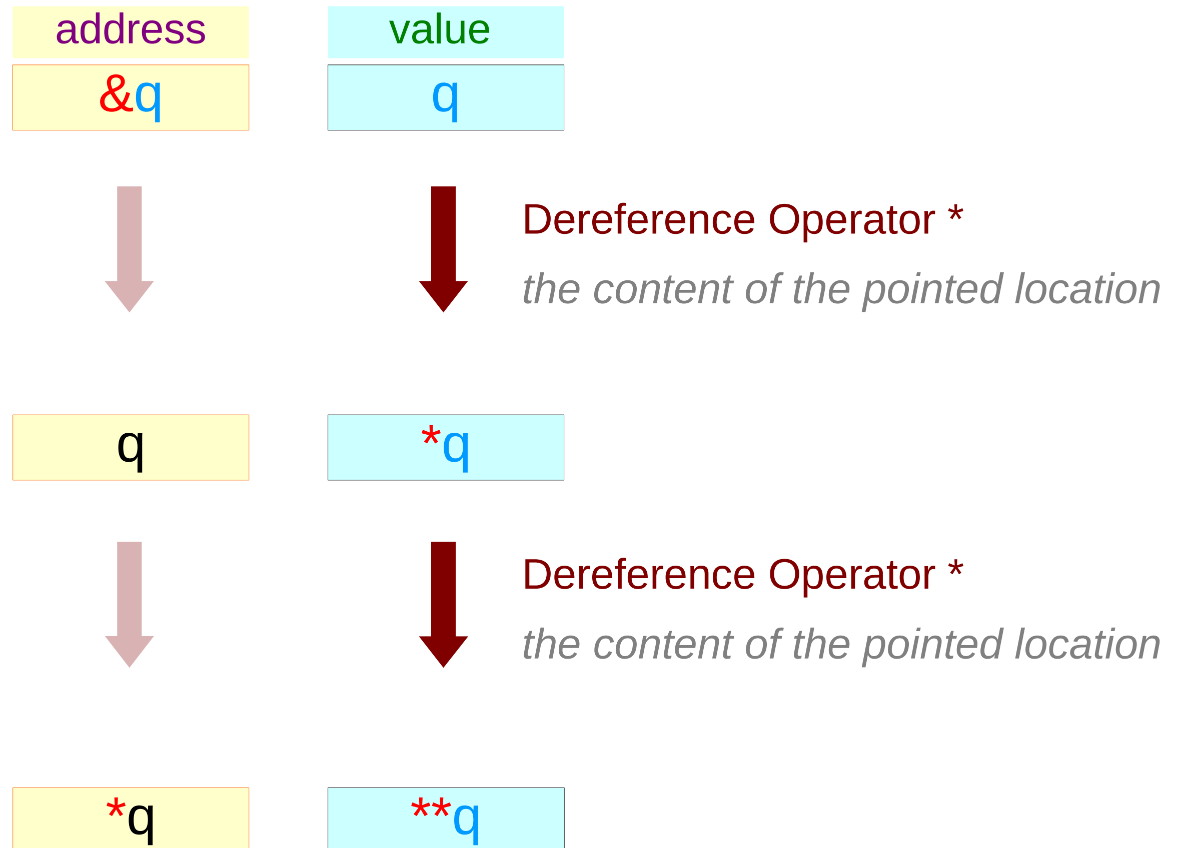


Double indirect access **q to an integer a



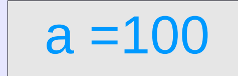
Double Indirect Access

3 memory accesses



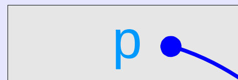
Values of variables

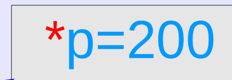
int a ;

&a 

address	value	
&a	a	integer

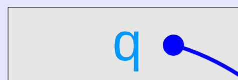
int * p ;

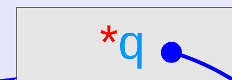
&p 

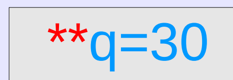
p 

address	value	
&p	p	address
p	*p	integer

int * * q ;

&q 

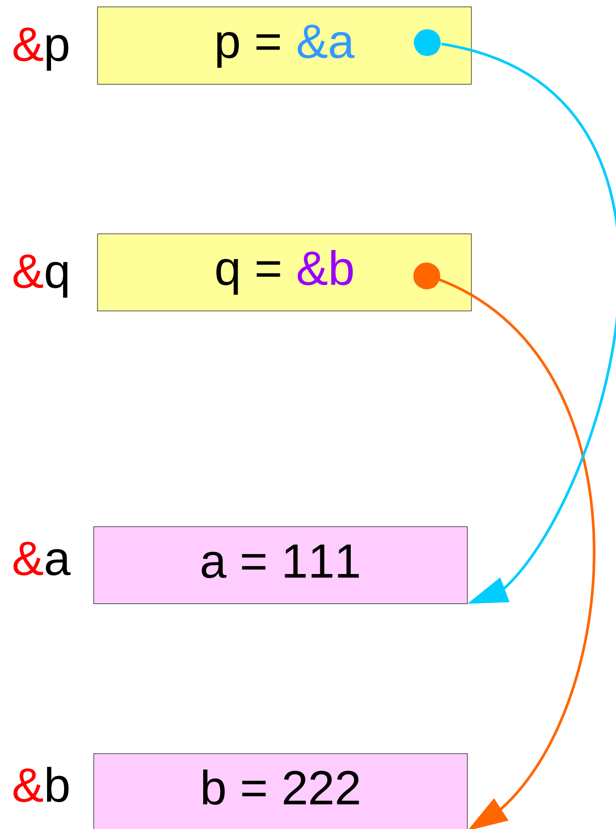
q 

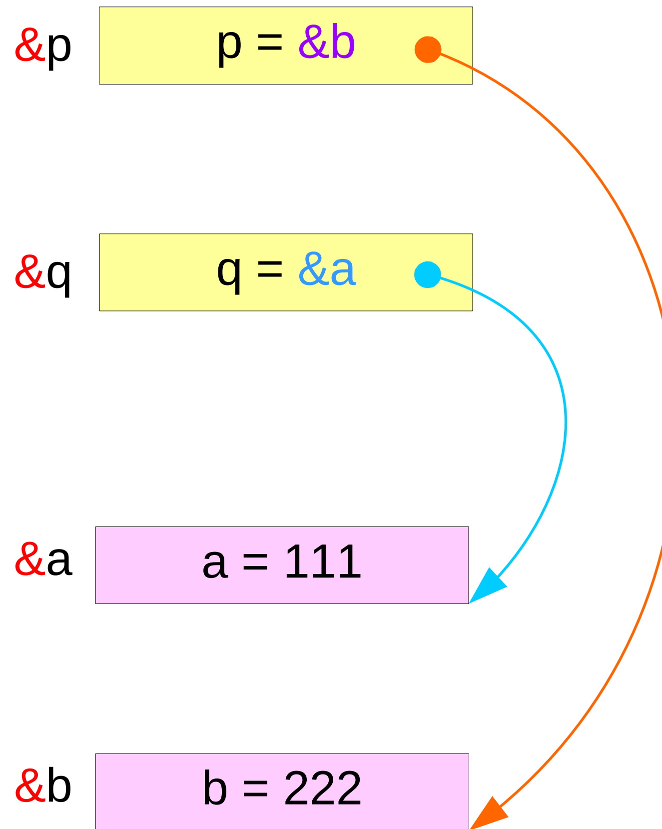
*q 

address	value	
&q	q	address
q	*q	address
*q	**q	integer

Swapping pointers

Swapping integer pointers





Swapping integer pointers

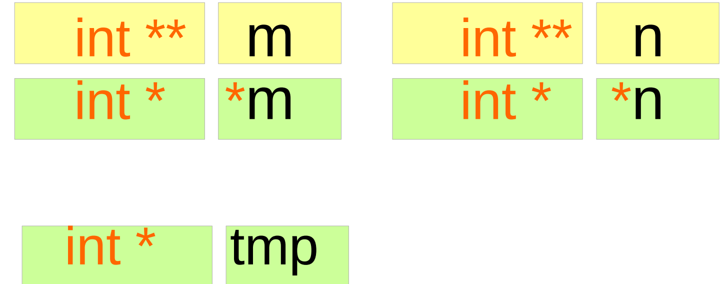
```
int              *p,      *q   ;

        swap_pointers(   &p,     &q );

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

function call

function prototype

Pass by integer pointer reference

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

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



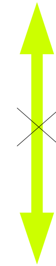

```
int   a,  b;
int  *p, *q;        p=&a, q=&b;
     …
swap_pointers( &p, &q );
```

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

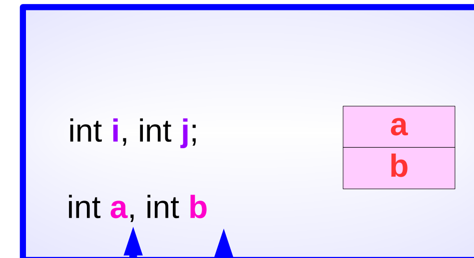
```
int main ()
{
  int x, int y;
  ...
  ...
  func1 ( 10, 20 );

  ...
  ...
}
```

x and y's
variable scope

Only **top** stack frame is <u>active</u>
and its variable can be <u>accessed</u>

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

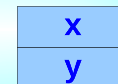
func1's
Stack
Frame



(10, 20)

main's
Stack
Frame

int x, int y;



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 <u>read</u> / <u>write</u> x and y
through their addresses

*a
*b

a=&x
b=&y

x and y's
variable scope

func1's Stack Frame

int i, int j;

int* a, int* b

a
b

(&x, &y)

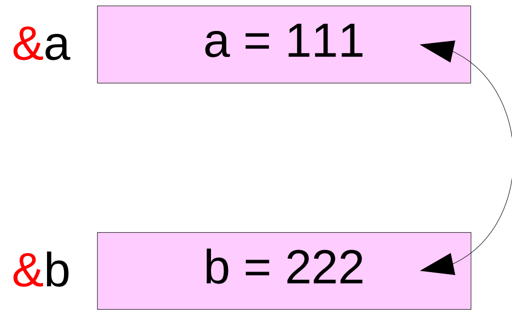
main's Stack Frame

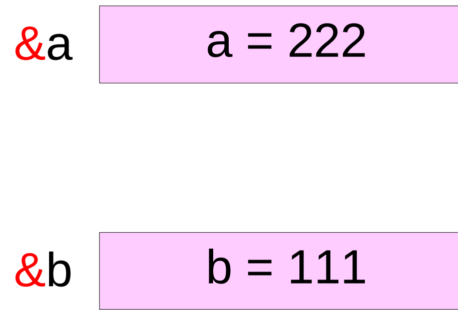
int x, int y;

x
y

*a
*b

Swapping integers

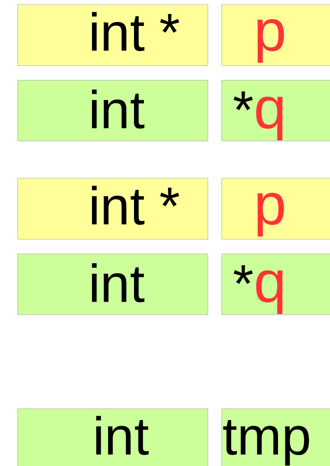





```
int a, b;

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;
}
```

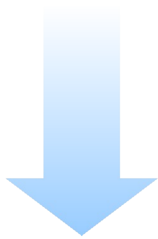



```
int a, b;
    …
swap( &a, &b );
```

Integer and Integer Pointer Types

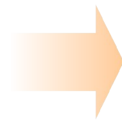
int	*m
int	*n

integer pointer declarations



a way of thinking

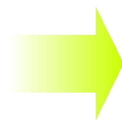
int *	m
int *	n



m
n

integer pointer variables — int *

int	*m
int	*n



*m
*n

treated as integer variables — int

variables ***types***

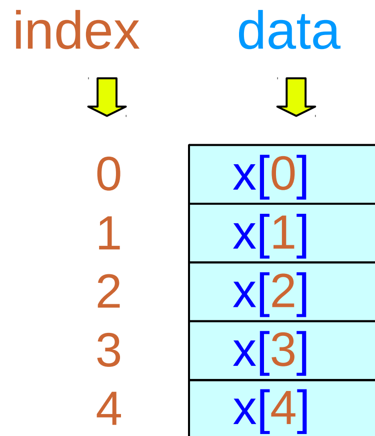
Arrays

Accessing array elements – using an address

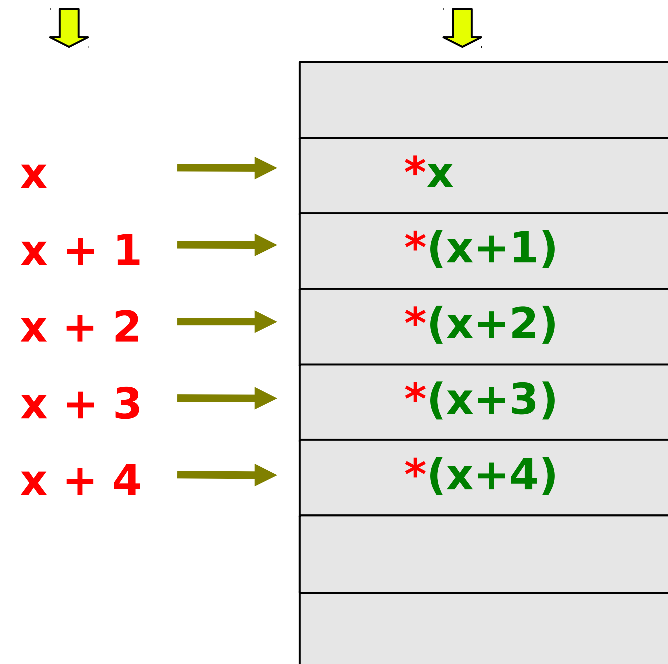
```
int        x[5];
```

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

5 int variables

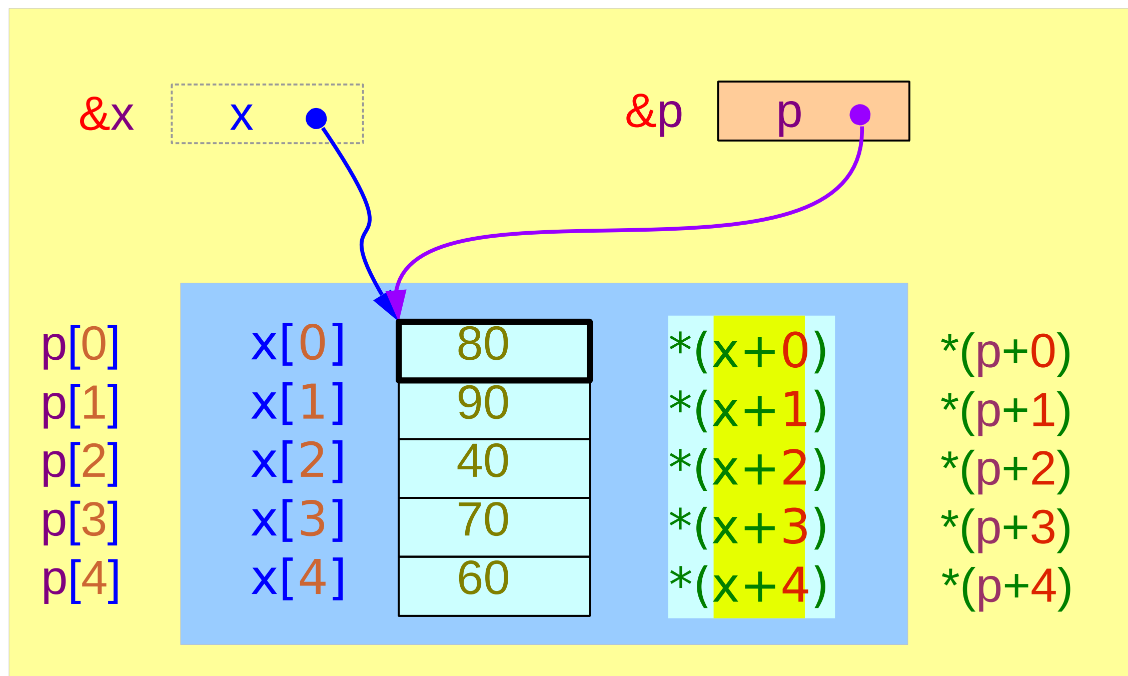


cannot change address x (constant)



Accessing an Array with a Pointer Variable

```
int x [5] = { 1, 2, 3, 4, 5 };

int  *p = x;
```



x is a constant symbol
cannot be changed

p is a variable
can point to other addresses

Pointer Type Cast

Changing the associated data type of an address

long a; &a address of a long value

int * p; address of an int value ← &a

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

char * r; address of a char value ← &a

Pointer Type Casting

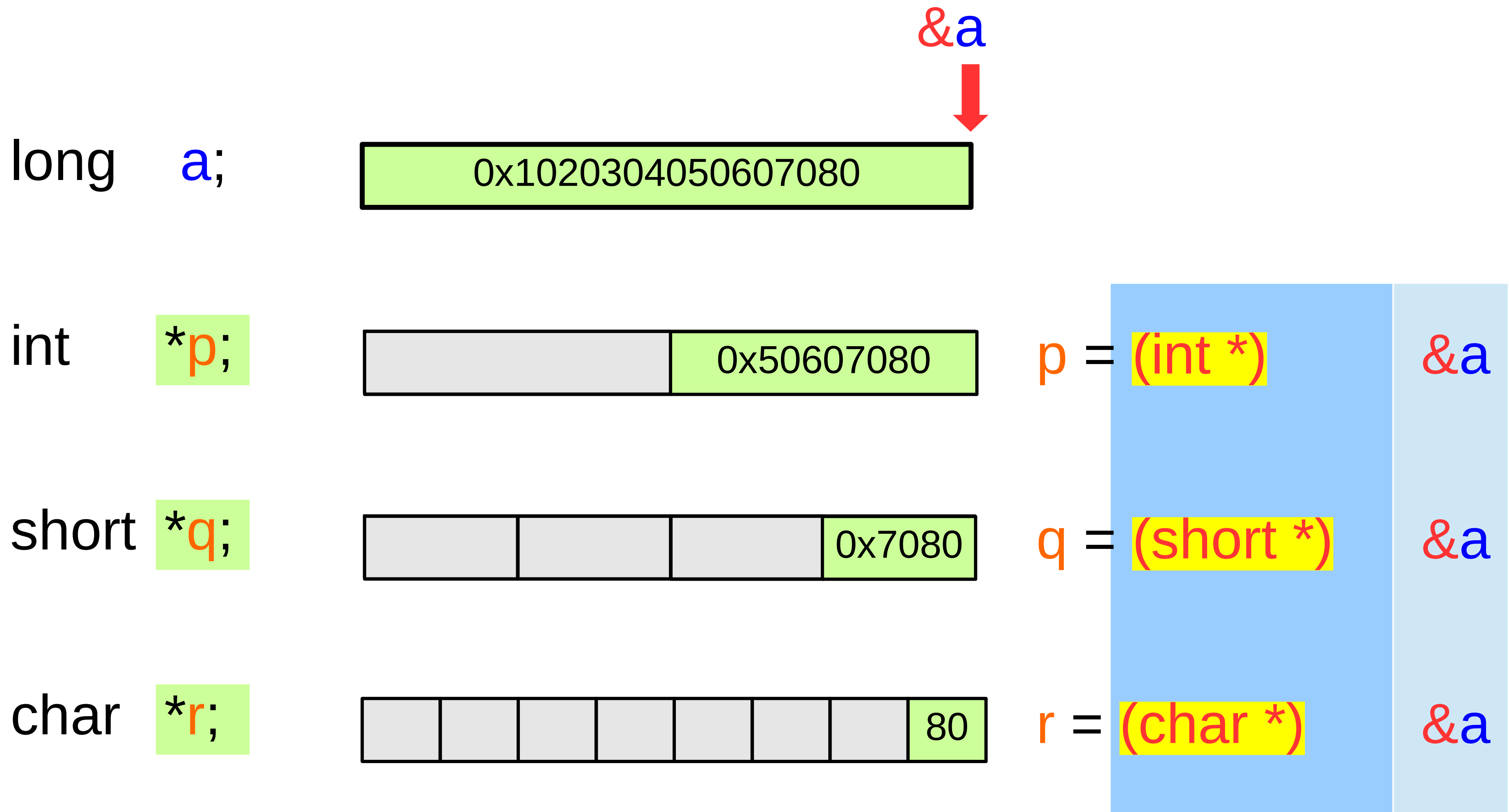
long a; &a

int * p; address of an int value ← p = (int *) &a

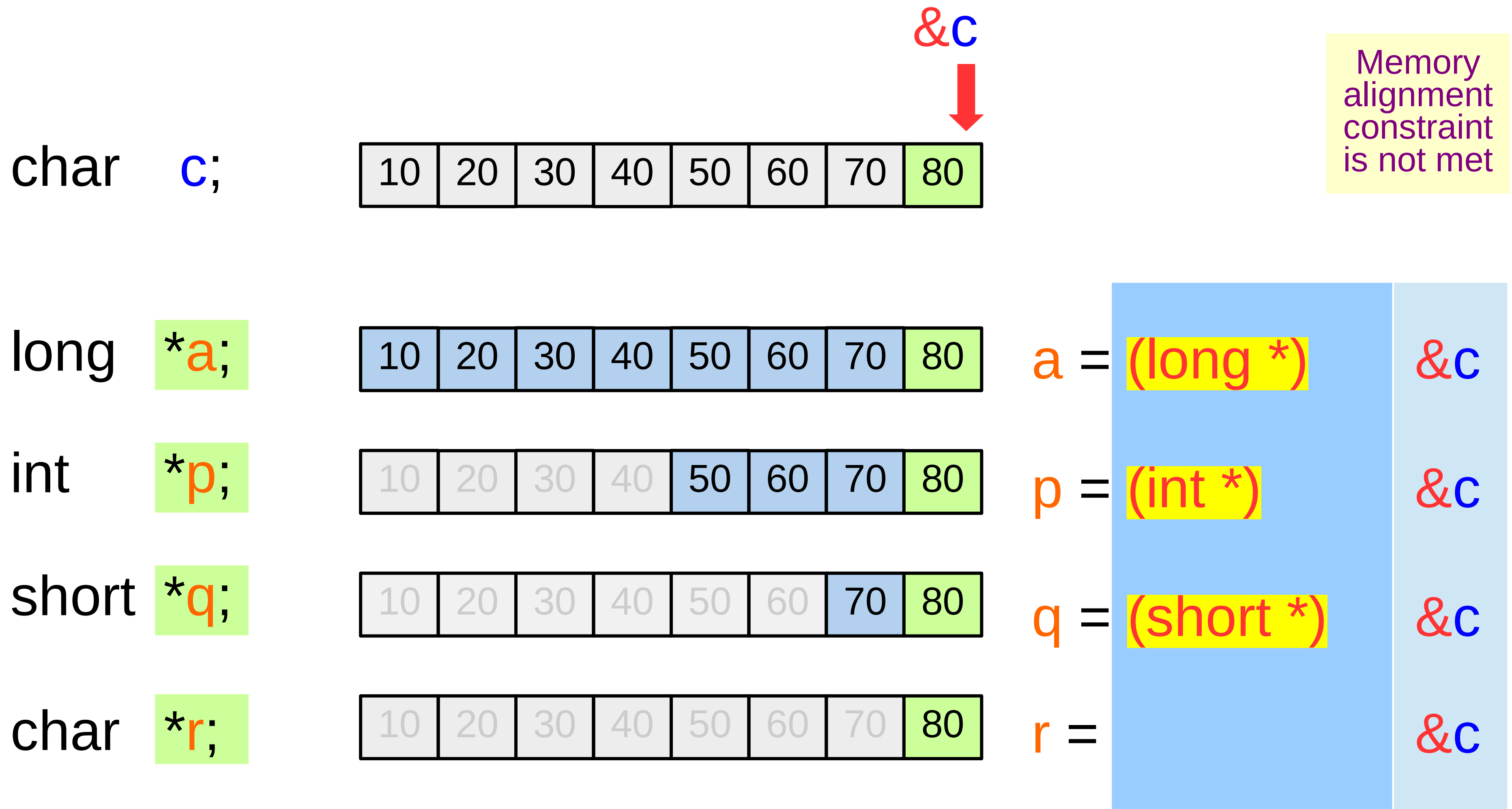
short * q; address of a short value ← q = (short *) &a

char * r; address of a char value ← r = (char *) &a

Re-interpretation of memory data – case I



Re-interpretation of memory data – case II



Depending on &C, the memory alignment constraint can be broken

const pointers

const type, const pointer type (1)

```
const int *p;

int * const q ;

const int * const r ;
```

constant integer value

constant integer pointer

constant integer value
constant integer pointer

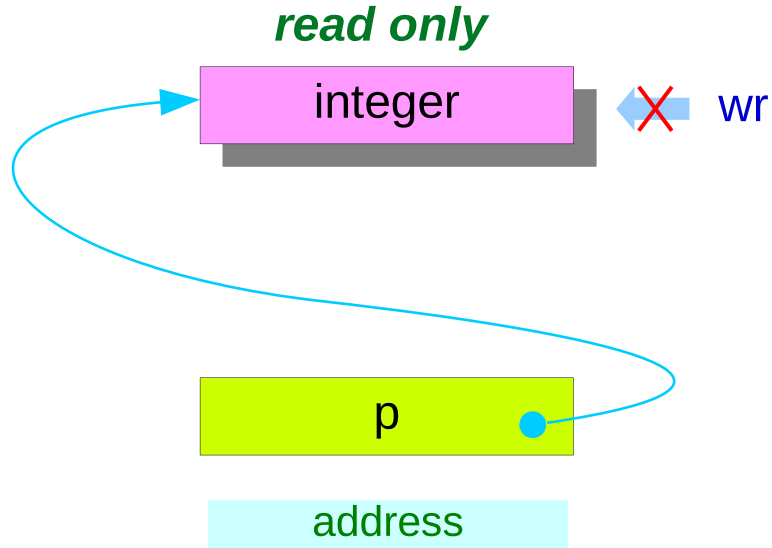
constant

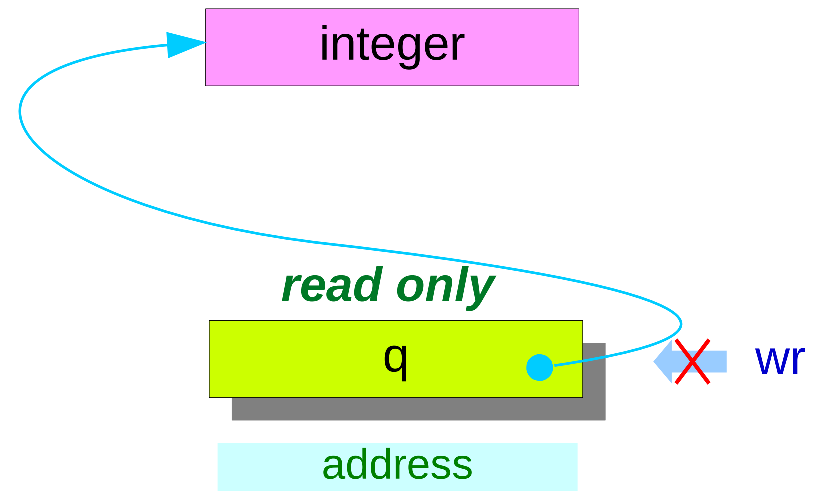
must not be changed
must not be updated
must not be written
must not be assigned

const type, const pointer type (2)

const int *p;

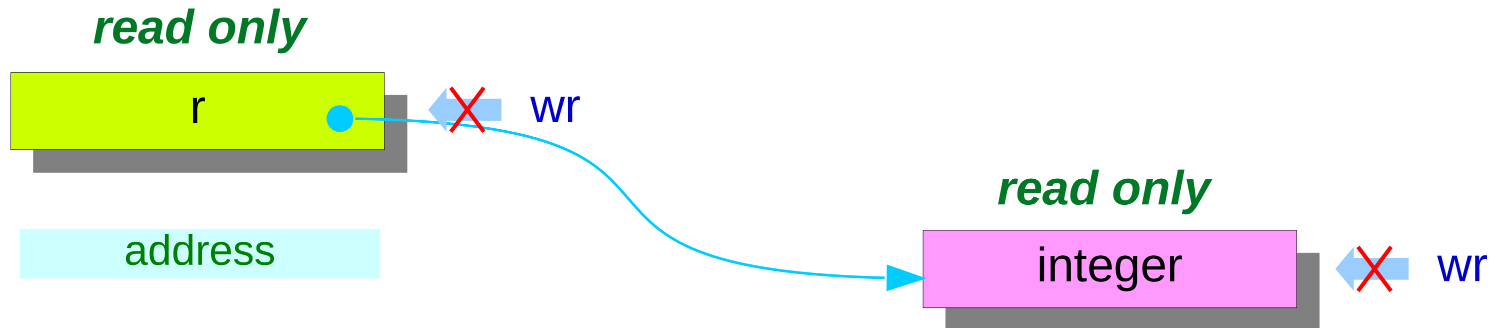
int * const q ;





const type, const pointer type (3)

const int *const r ;



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