

# Pointers (1A)

---

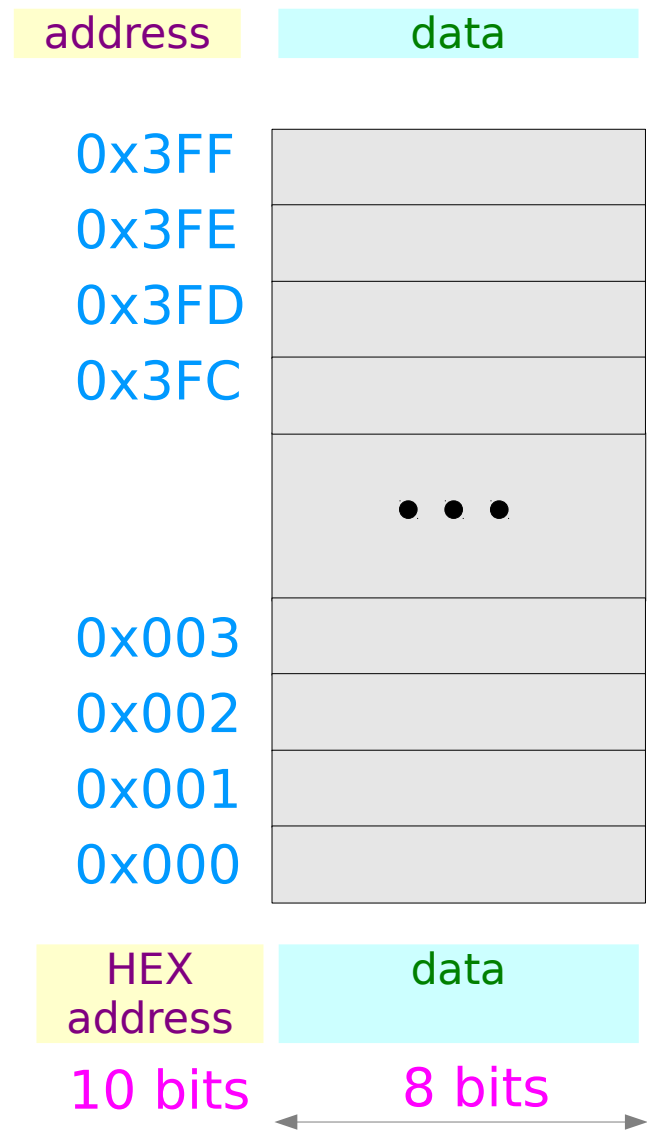
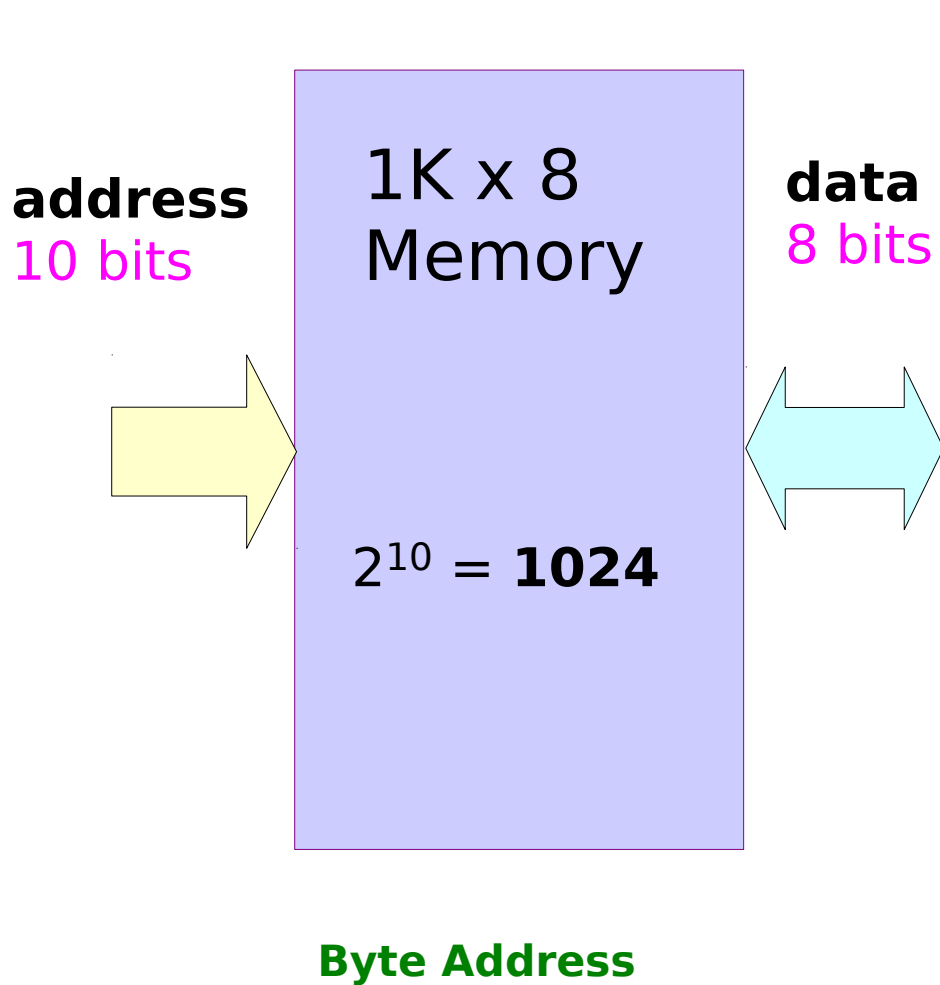
Copyright (c) 2010 - 2018 Young W. Lim.

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](mailto:youngwlim@hotmail.com).

This document was produced by using LibreOffice.

# Byte Address and Data in a Memory



# 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

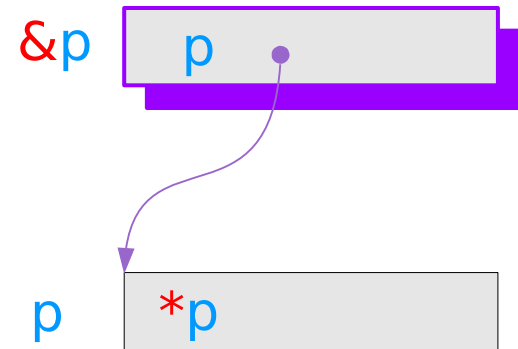
**type**            **variable**

```
int * p;
```

*pointer to int*

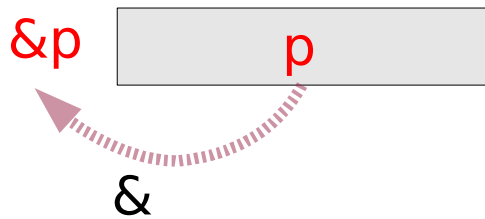
```
int * p;
```

*int*

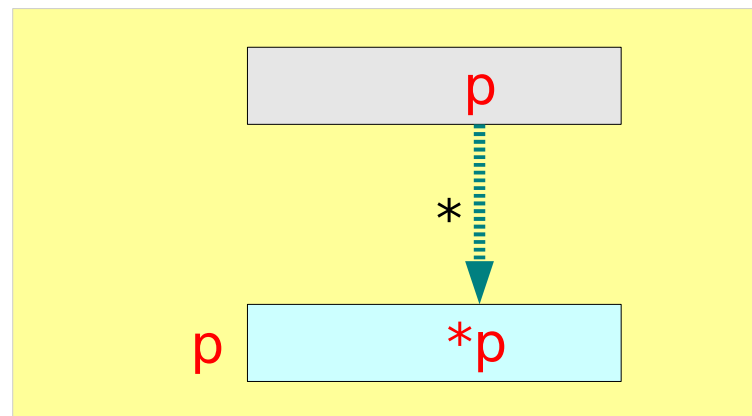
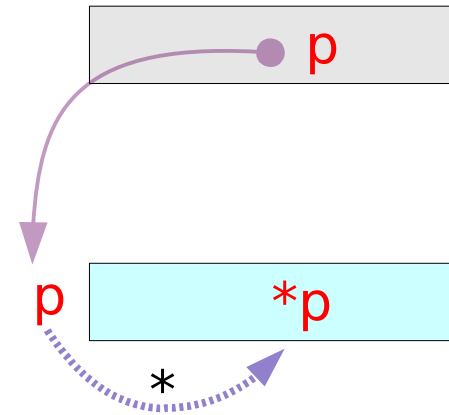


# Dereferencing

*The address of a variable :  
Address of operator &*



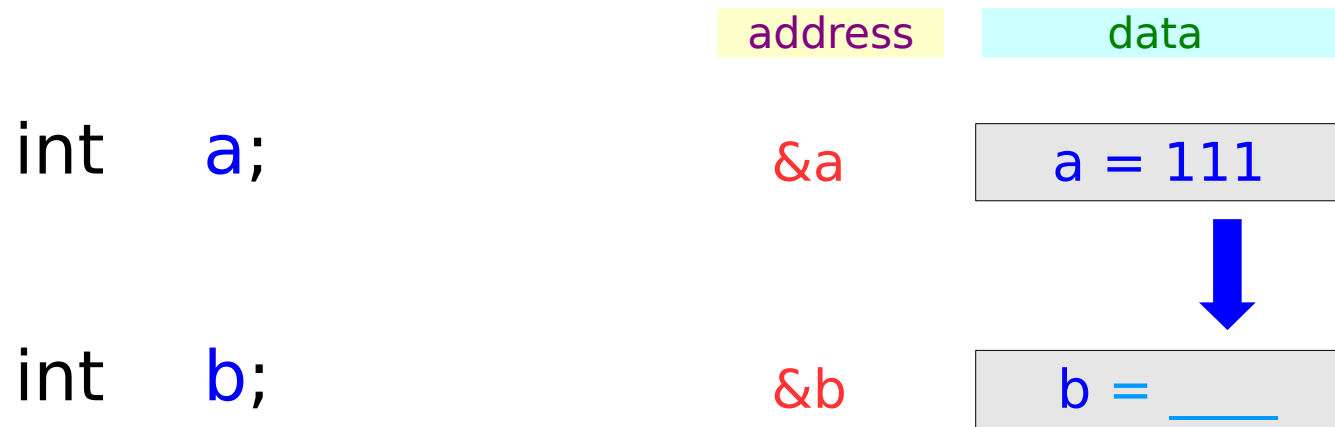
*The content of a pointed location :  
Dereferencing operator \**



# Variables and their addresses

	address	data
<code>int a;</code>	<code>&amp;a</code>	<code>a</code>
<code>int *p;</code>	<code>&amp;p</code>	<code>p</code>

# Assignment of a value



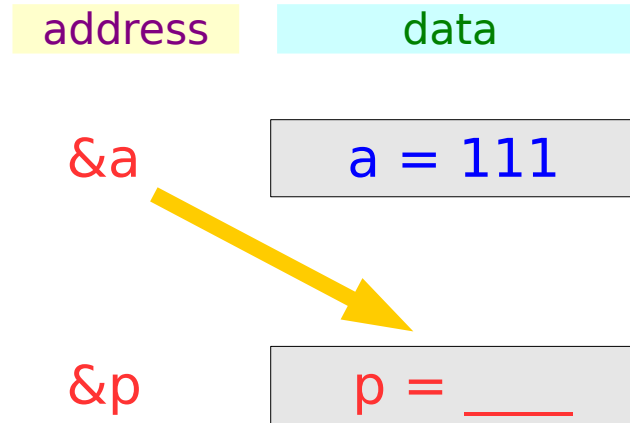
b = a;



# Assignment of an address

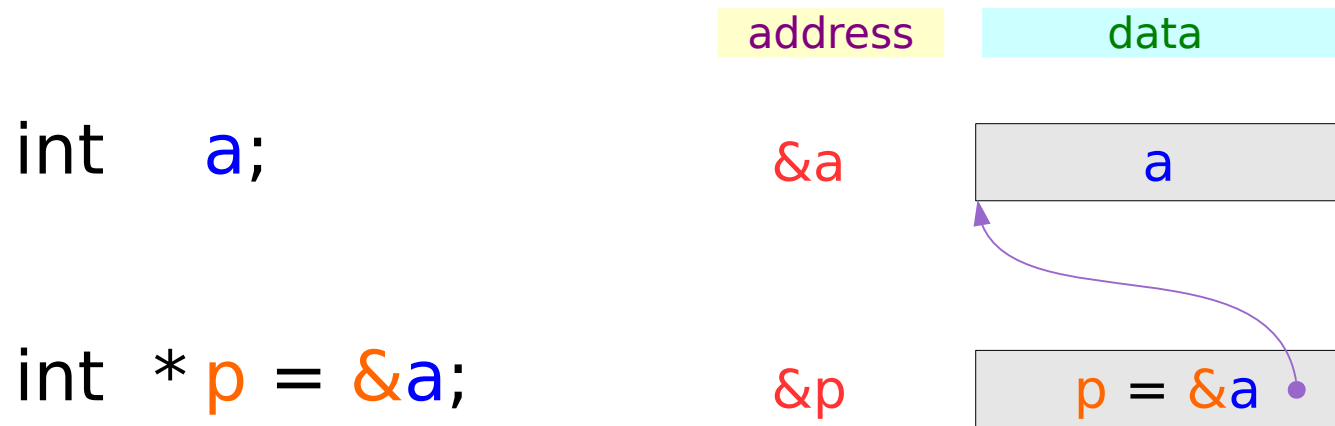
```
int a;
```

```
int *p;
```



```
p = &a;
```

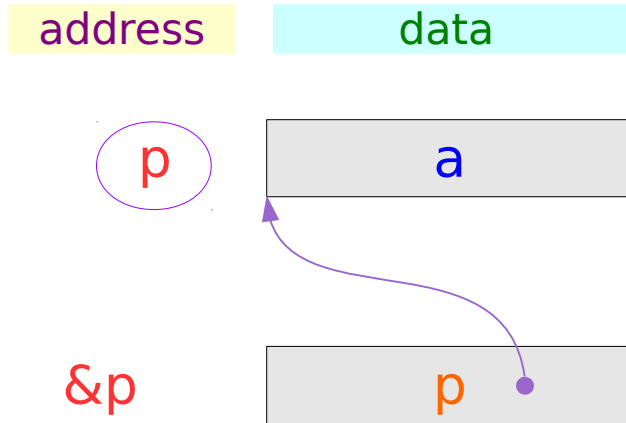
# Variables with initializations



# Pointed addresses : p

```
int a;
```

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

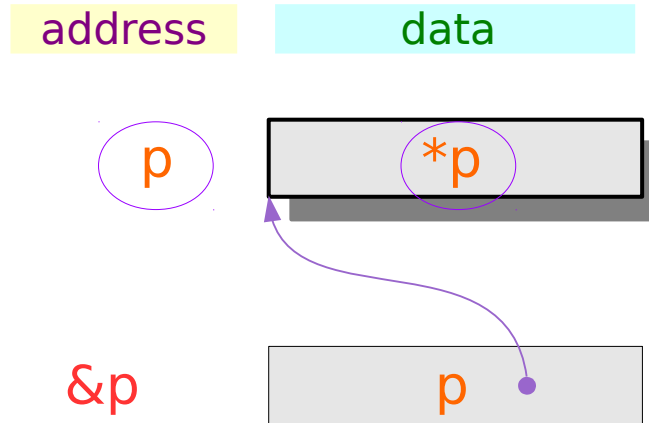


$p \equiv \&a$

# Dereferenced Variable : \*p

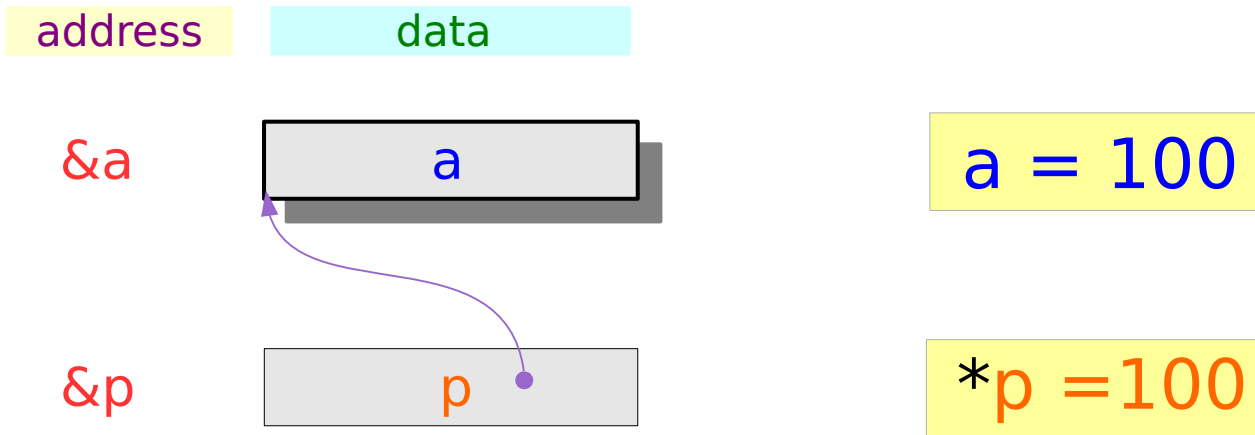
```
int a;
```

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



$p \equiv \&a$   
 $*p \equiv a$

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



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

- 
1. Pass by Reference
  2. Arrays

---

# 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

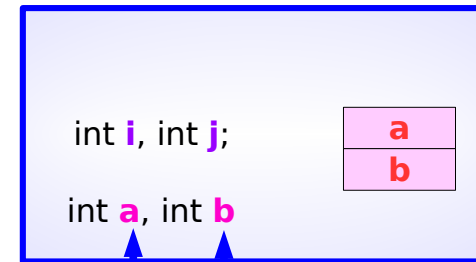
```
int main ()  
{  
    int x, int y;  
    ...  
    ...  
    func1 ( 10, 20 );  
    ...  
    ...  
}
```

**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

**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;  
    ...  
    ...  
    ...  
    ...  
}
```

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

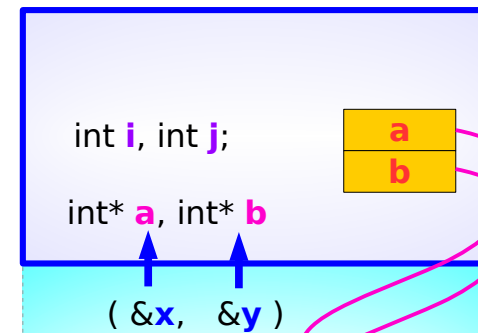
**a**=&**x**  
**b**=&**y**

**x** and **y**'s  
variable scope

```
int main ()  
{  
    int x, int y;  
    ...  
    ...  
    func1 ( &x, &y );  
    ...  
    ...  
}
```

\***a**  
\***b**

**func1**'s  
Stack  
Frame

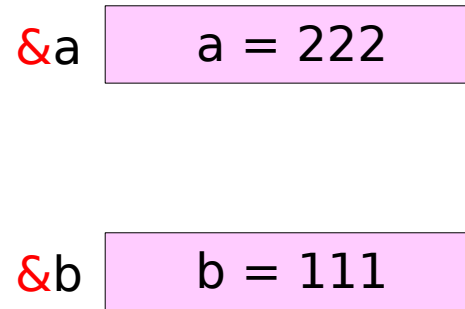
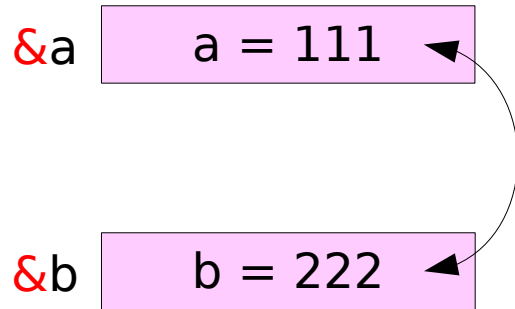


**main**'s  
Stack  
Frame

int **x**, int **y**;

\***a**  
\***b**

# Swapping integers



```
int a, b;
```

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

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

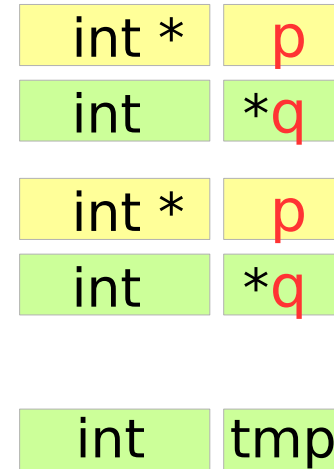
function call

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  
int *m  
int *n
```



```
m  
n  
*m  
*n
```

integer pointer variables

*treated as* integer variables

*variables*

```
int *
```

```
int
```

*types*

---

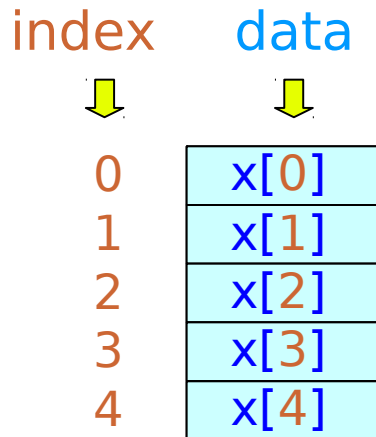
# Arrays

# Accessing array elements - using an address

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

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

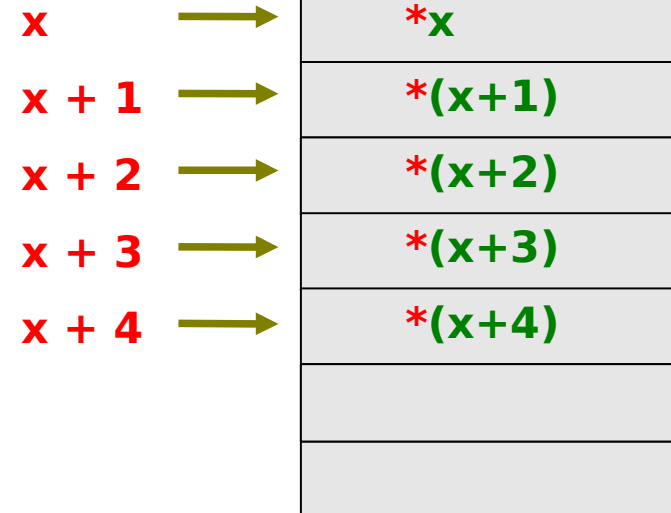
5 int variables



cannot change  
address x  
(constant)

address

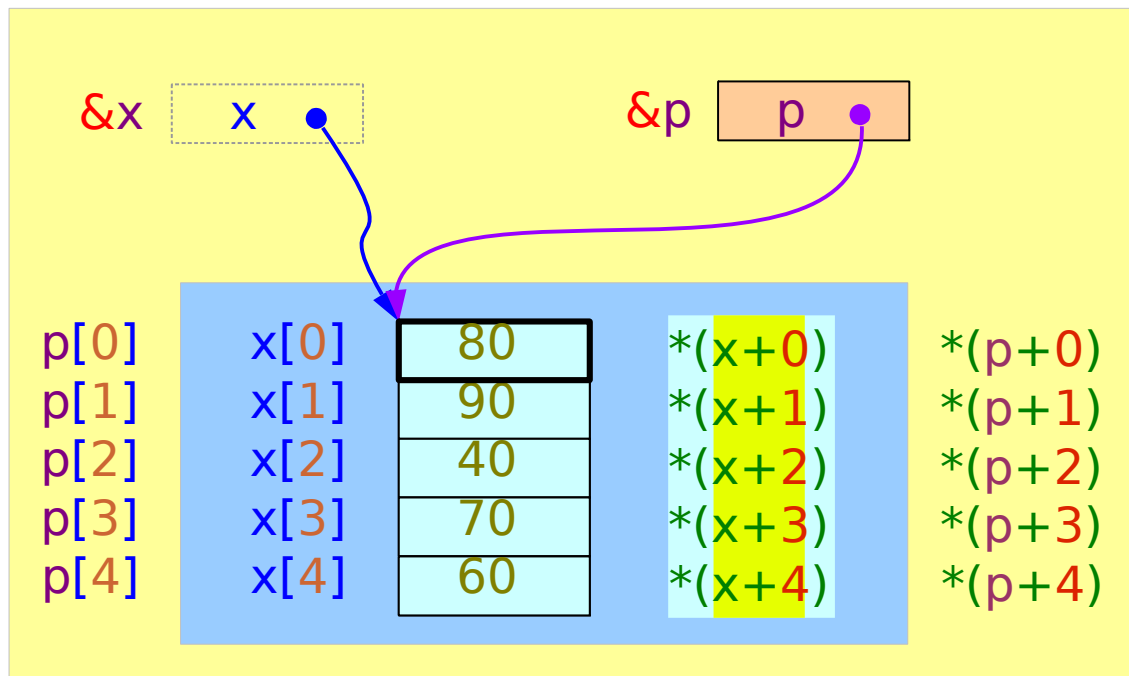
data



# 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

---

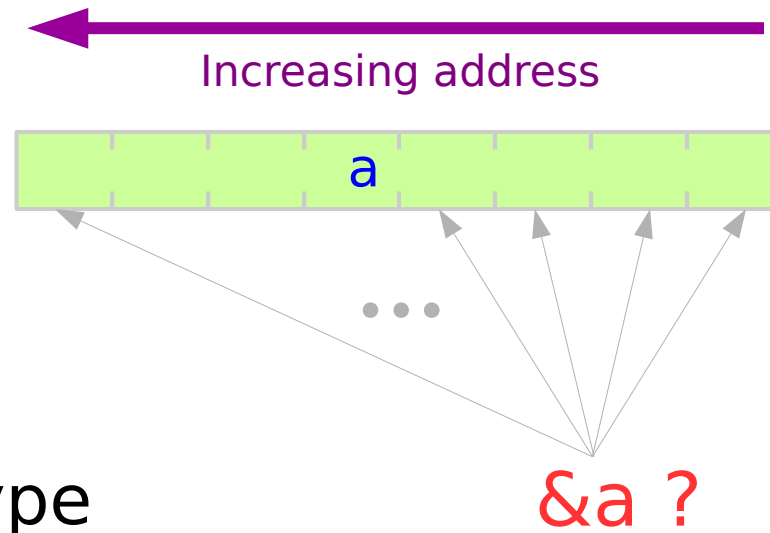
Byte Address  
Little Endian  
Big Endian



# Byte Addresses

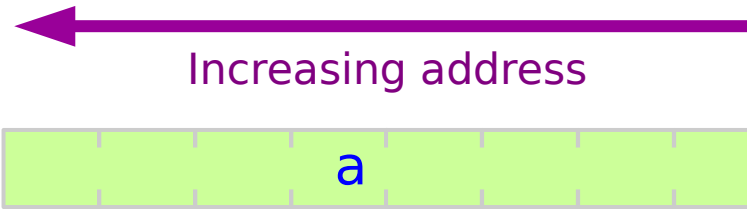
long a;

8-byte size data type



# Little / Big Endian Ordering of Bytes

long a;



MSByte Little Endian LSByte



$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$

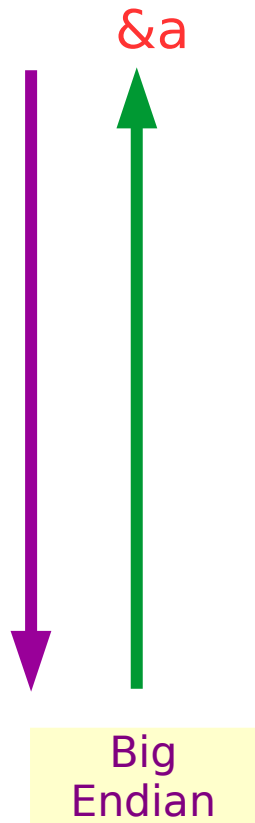
LSByte Big Endian MSByte



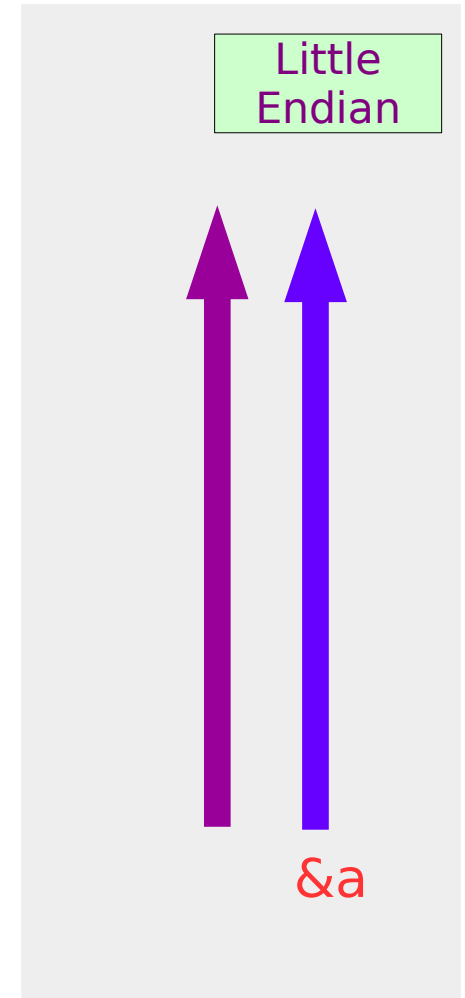
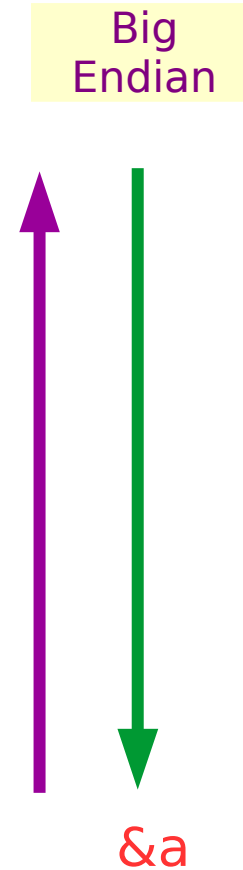
$a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7$

# Increasing address, Increasing byte weight

**downward**, increasing address



**upward**, increasing address

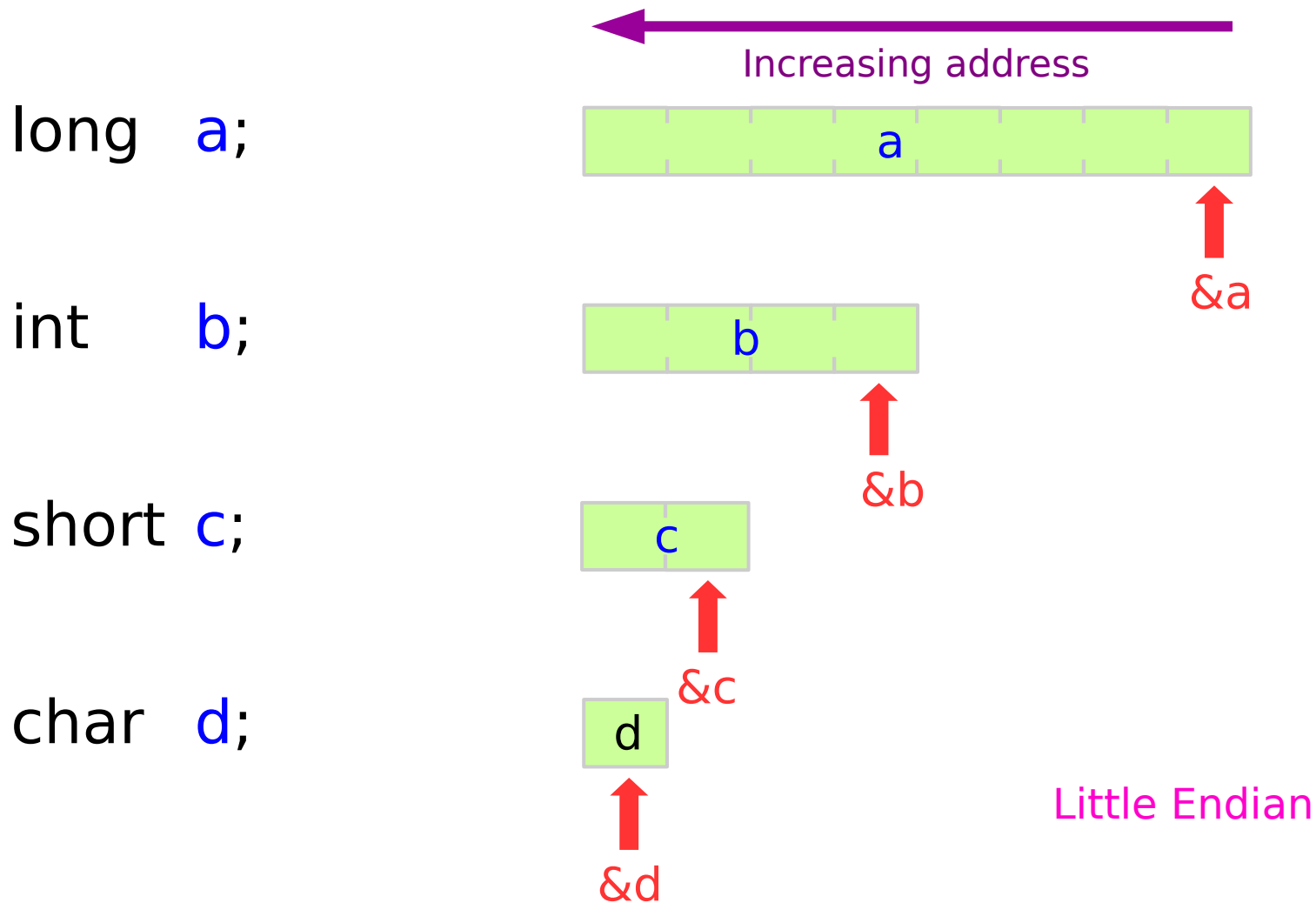


<https://stackoverflow.com/questions/15620673/which-bit-is-the-address-of-an-integer>

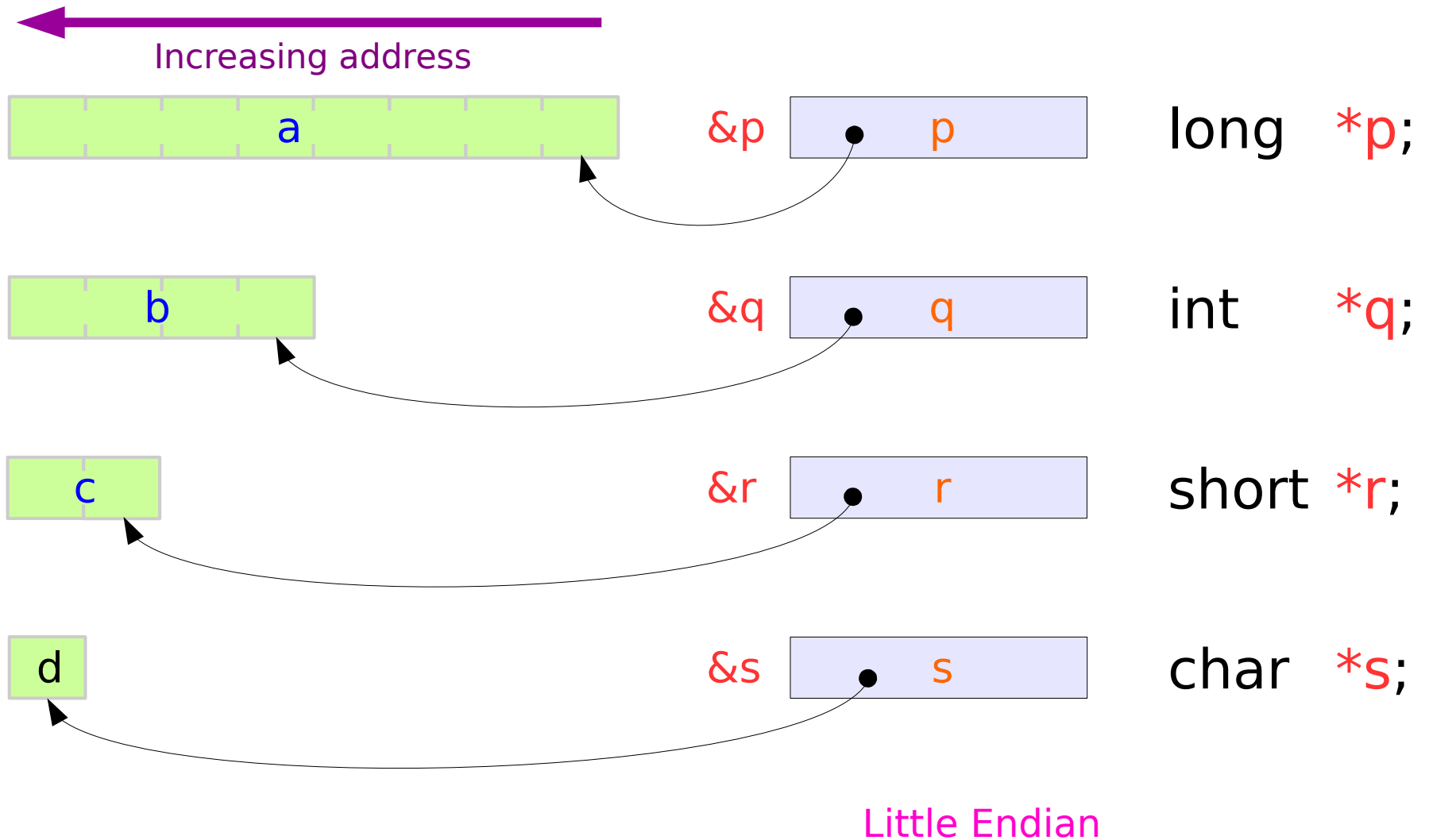
---

# Pointer Types

# Integer Type Variables and Their Addresses

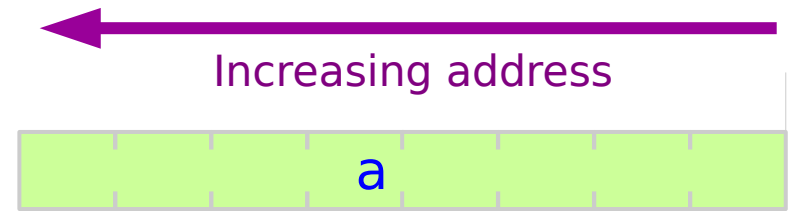


# Points to the LSByte



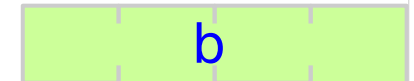
# Aligning variables of different sizes

`long a;`    `sizeof(long)` → 8 (bytes)



`&a` ↑

`int b;`    `sizeof(int)` → 4 (bytes)



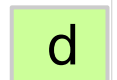
`&b` ↑

`short c;`    `sizeof(short)` → 2 (bytes)



`&c` ↑

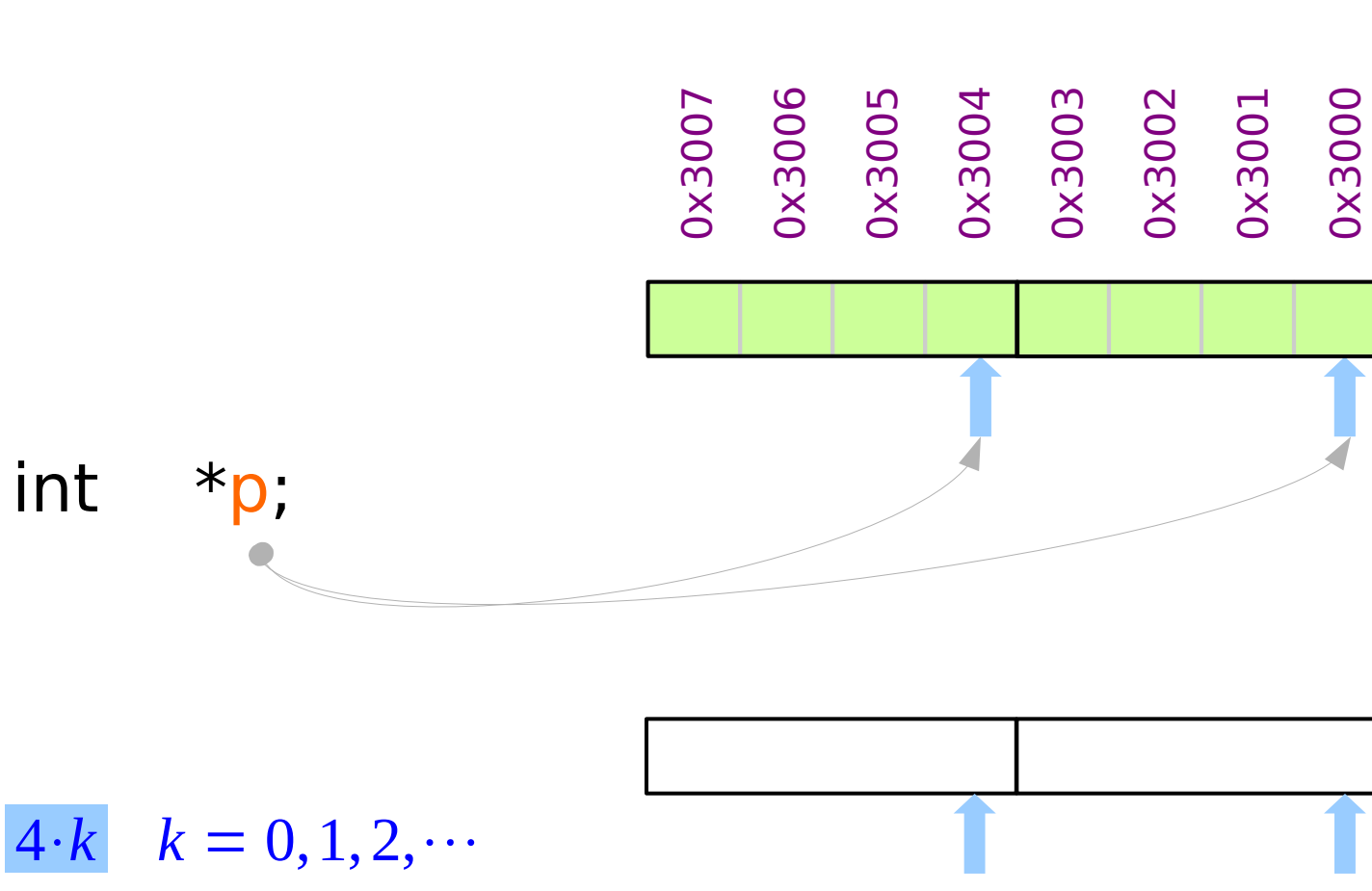
`char d;`    `sizeof(char)` → 1 (bytes)



`&d` ↑

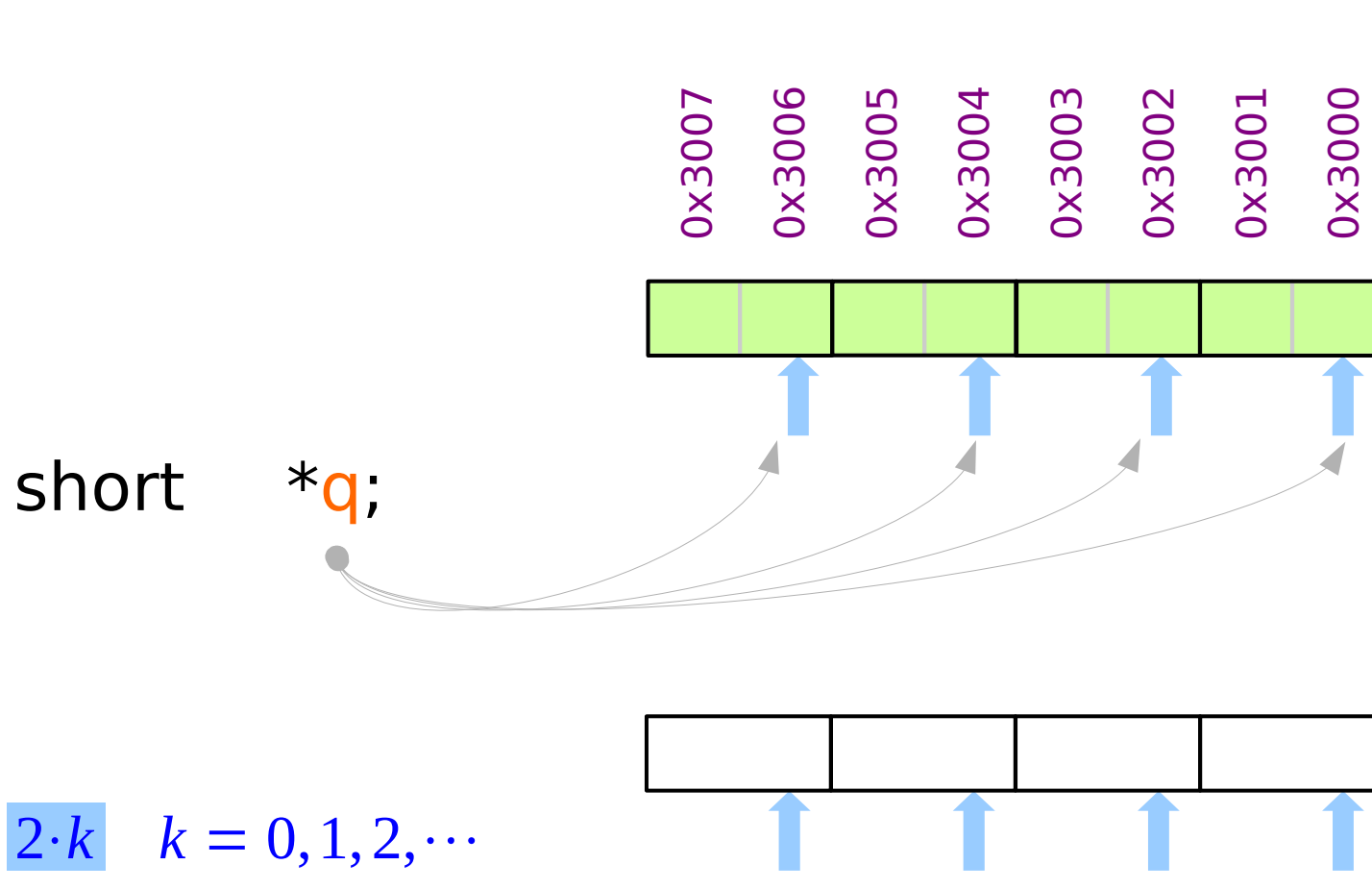
Memory Alignment  
in the Little Endian

# Possible addresses for **int** values

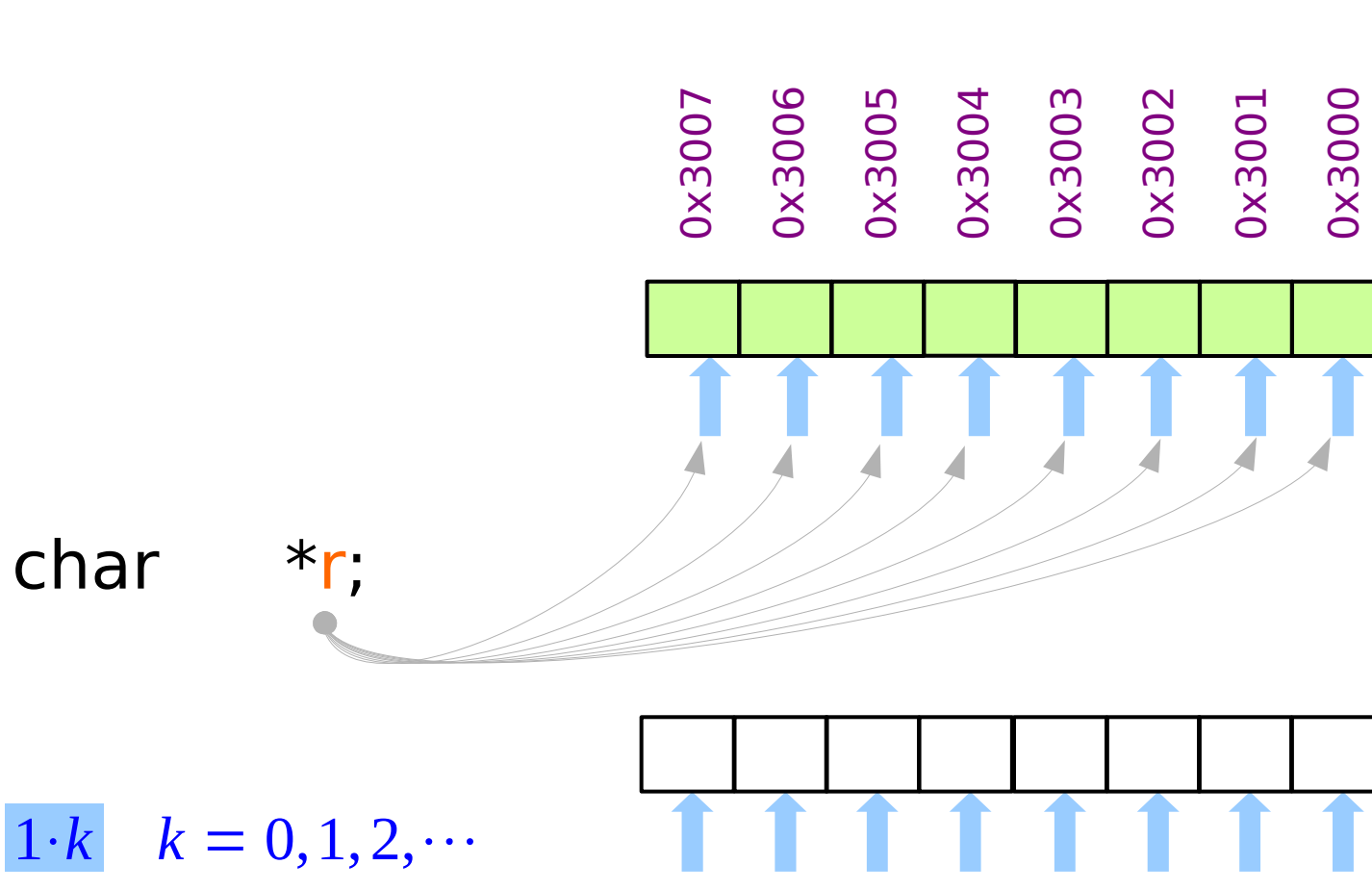




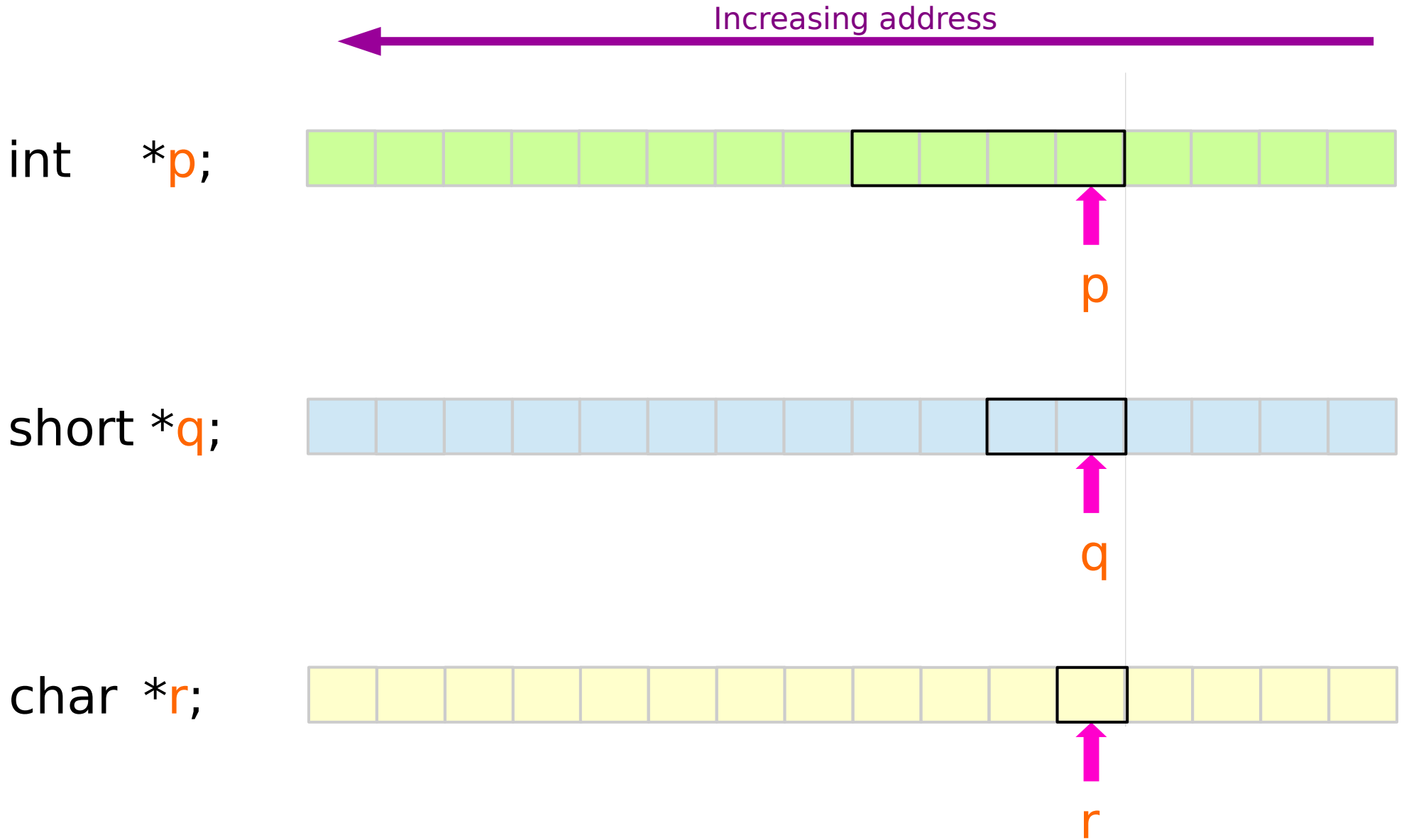
# Possible addresses for **short** values



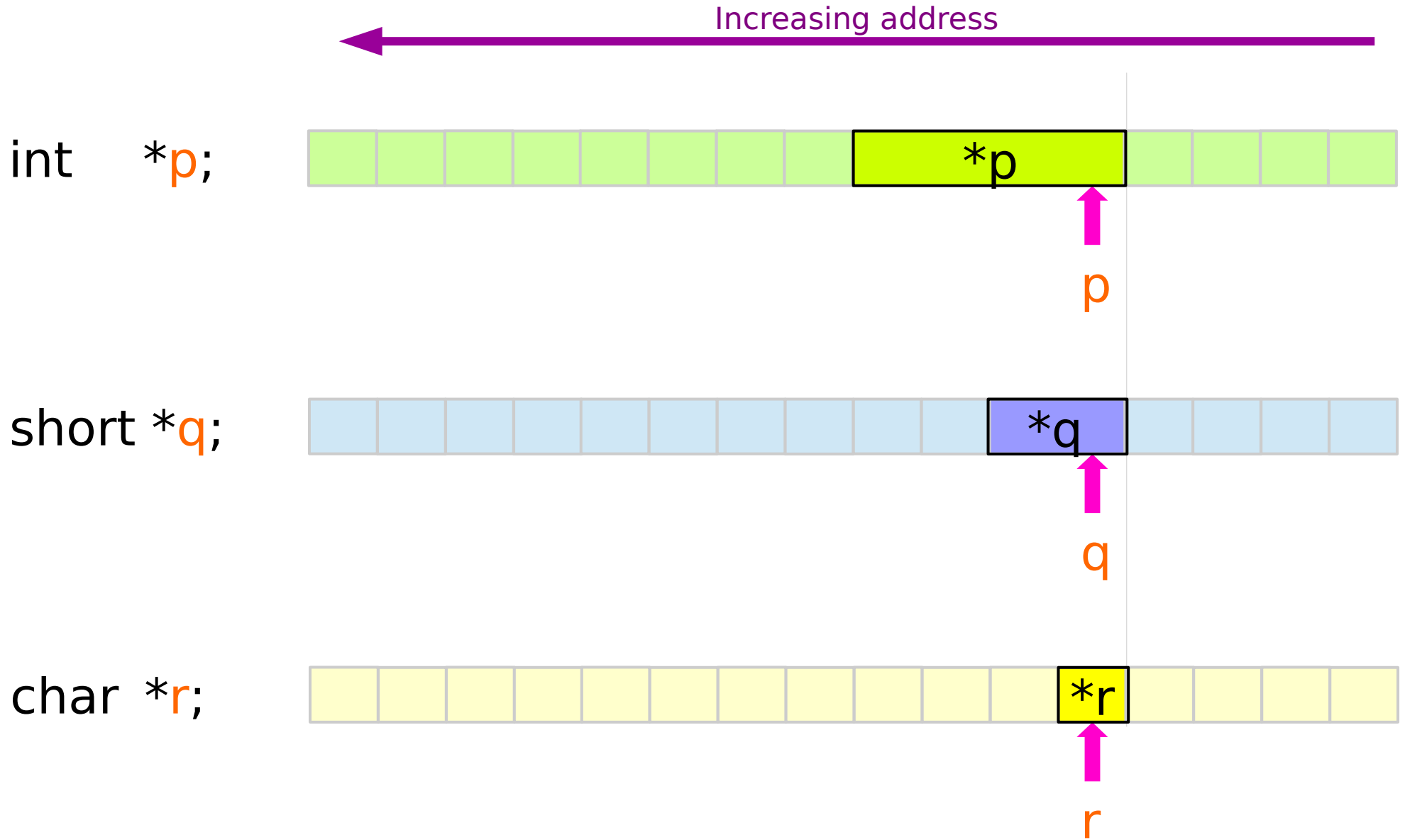
# Possible addresses for **char** values



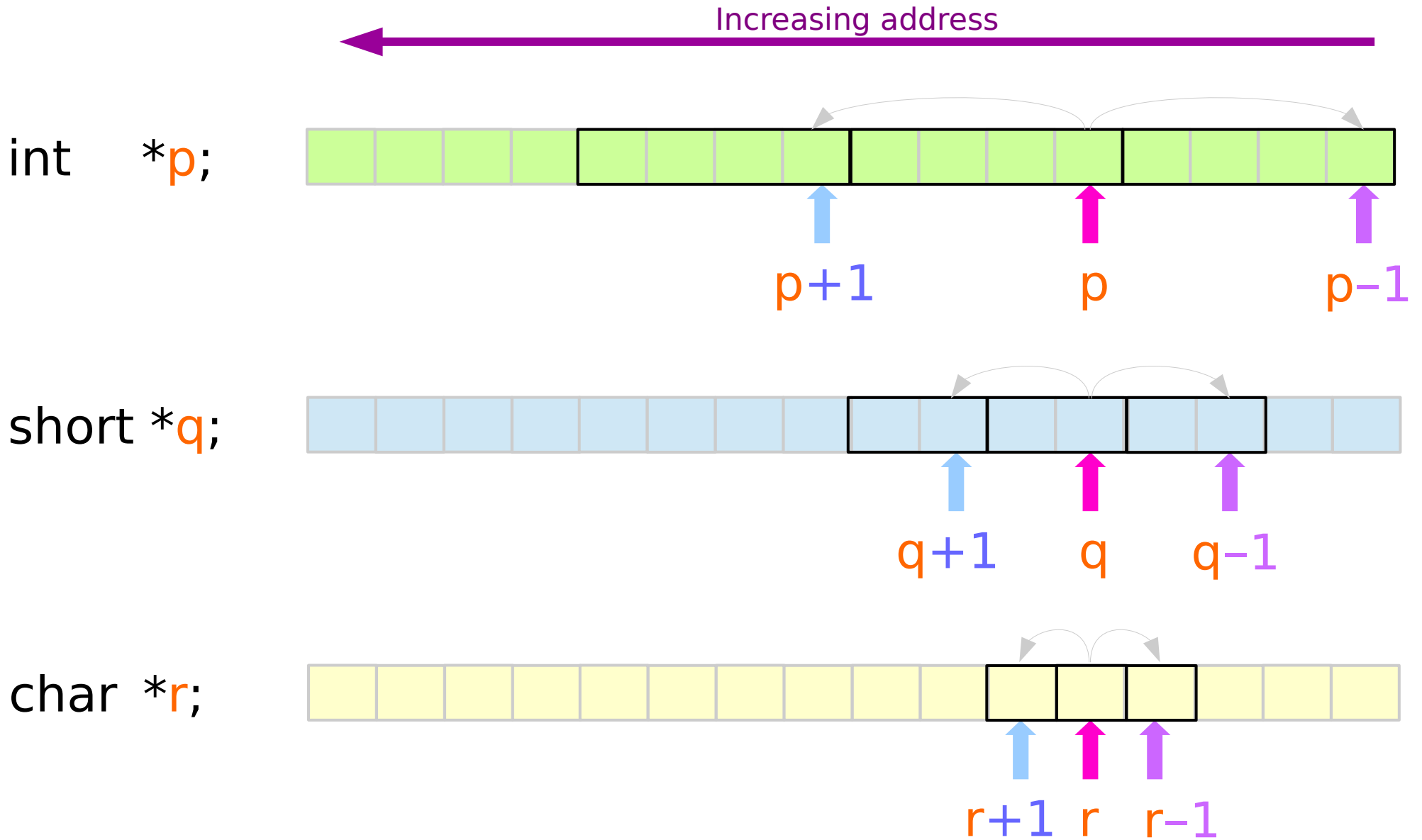
# Data size at the pointed addresses



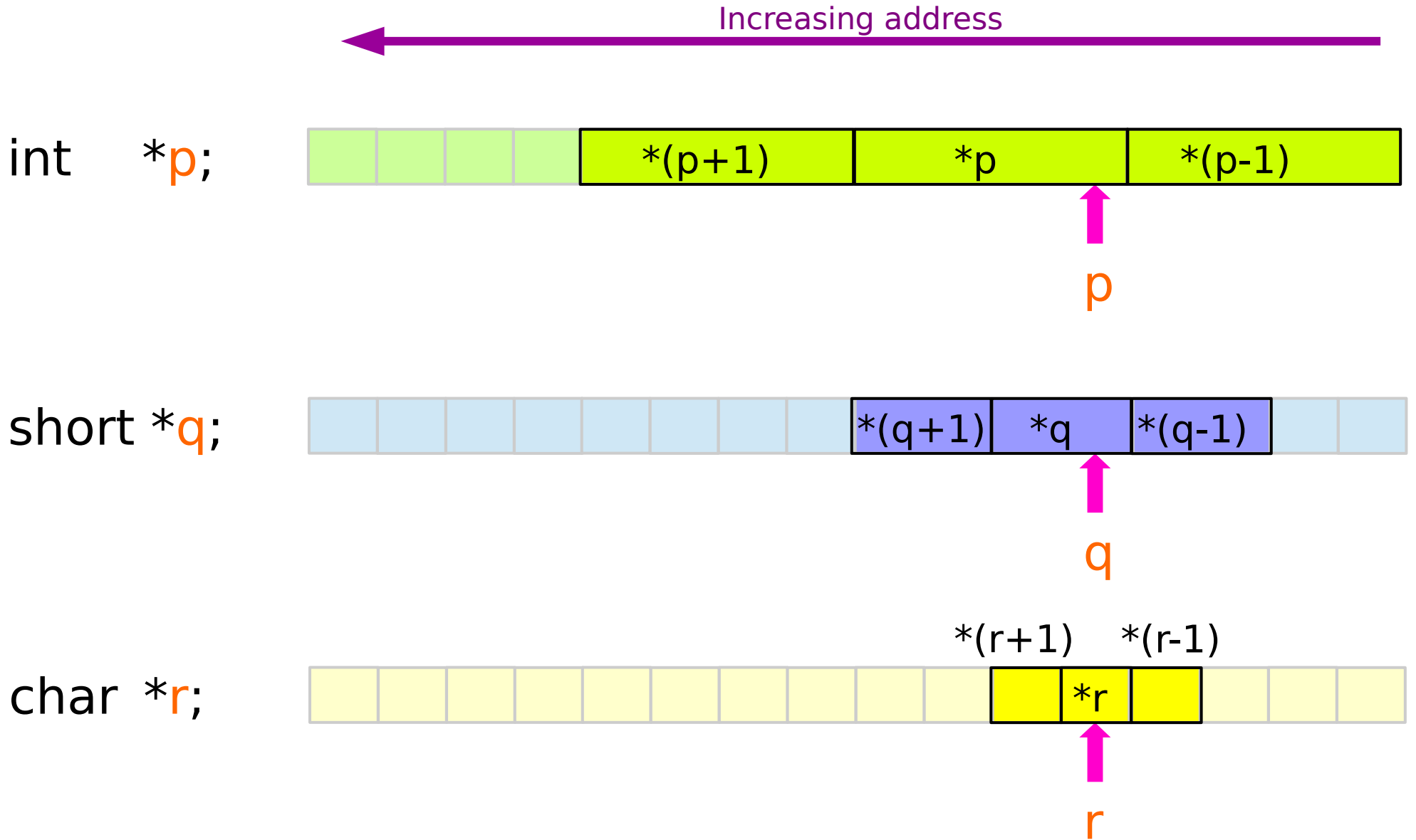
# Associated data at the pointed addresses



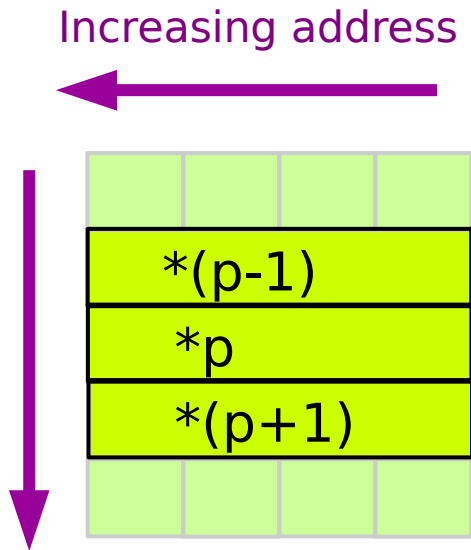
# Incrementing / decrementing pointers



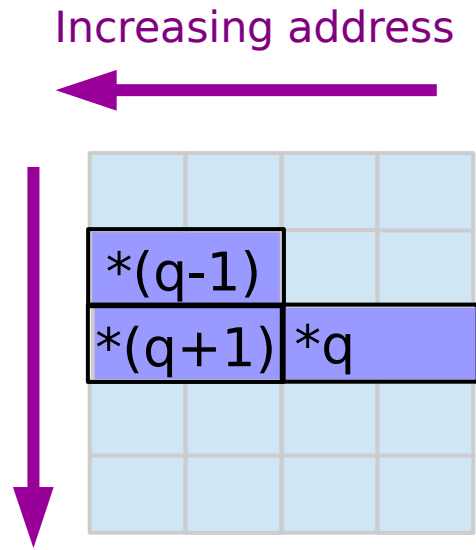
# Dereferencing the inc/decremented pointers



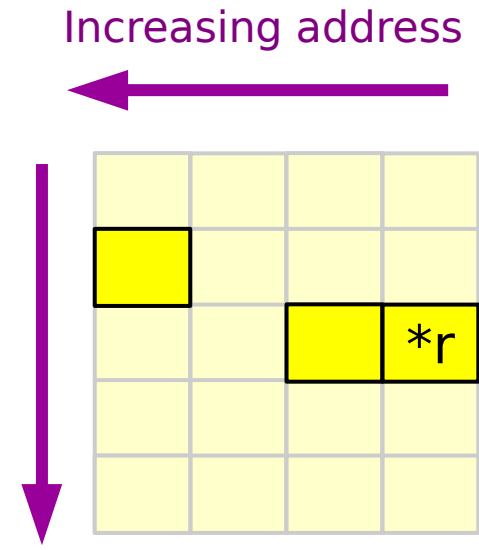
# Dereferencing the inc/decremented pointers



`int *p;`

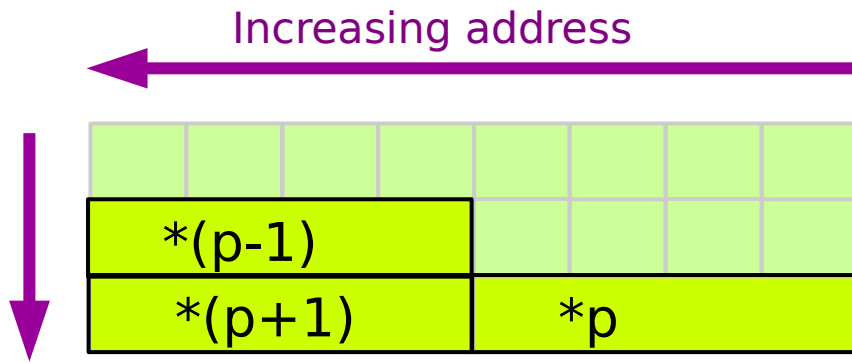


`short *q;`

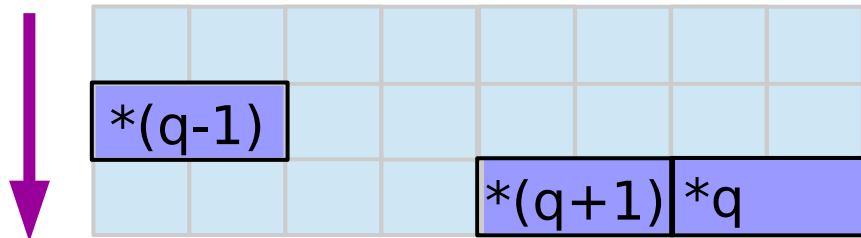


`char *r;`

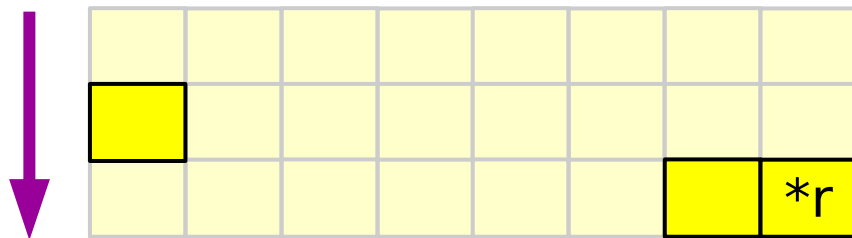
# Dereferencing the inc/decremented pointers



```
int *p;
```



```
short *q;
```



```
char *r;
```



---

# 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 ←

short\* q;    address of a short value ←

char \* r;    address of a char value ←



# Pointer Type Casting

long a;

address of a long value



&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

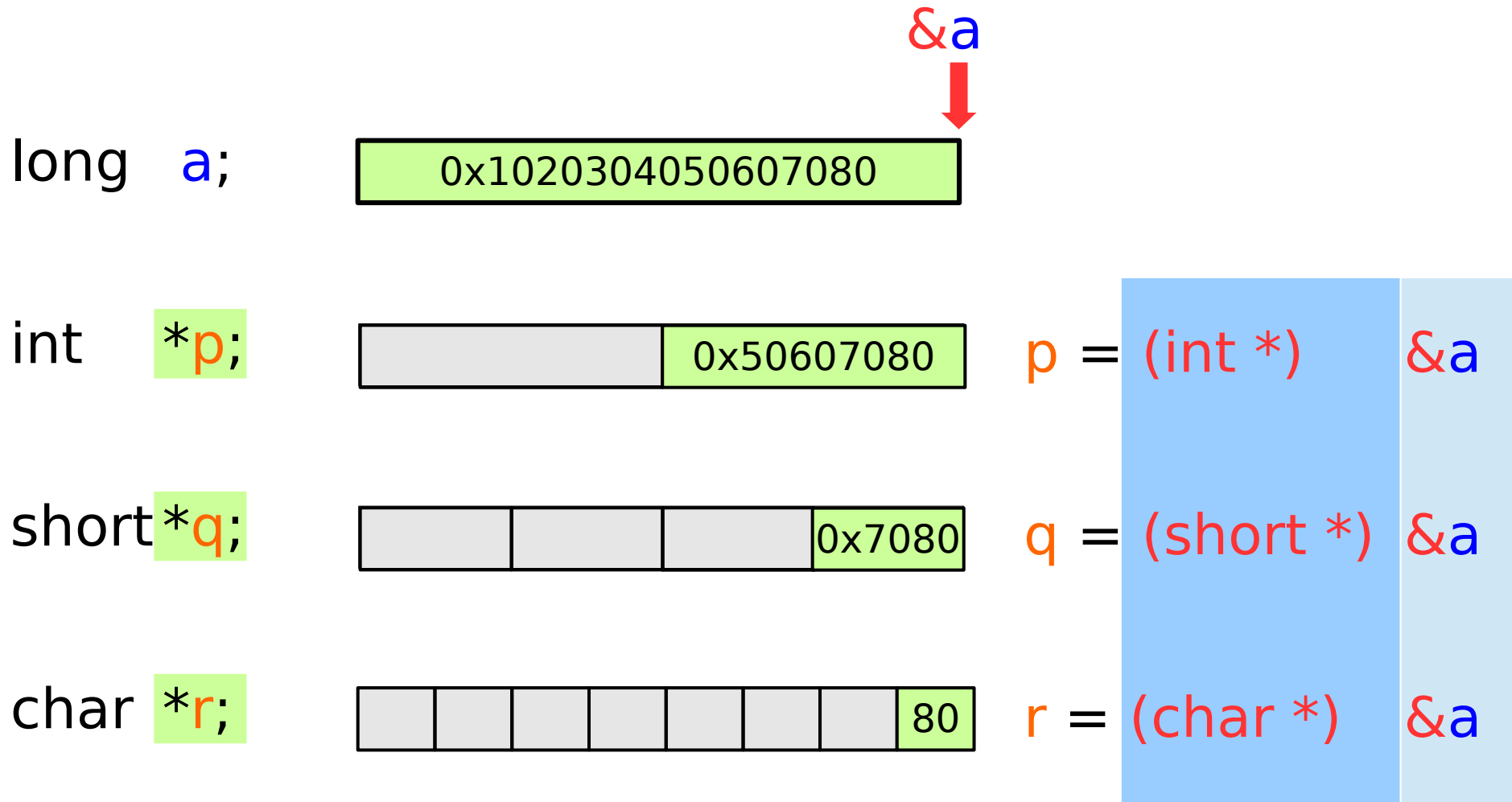


r =

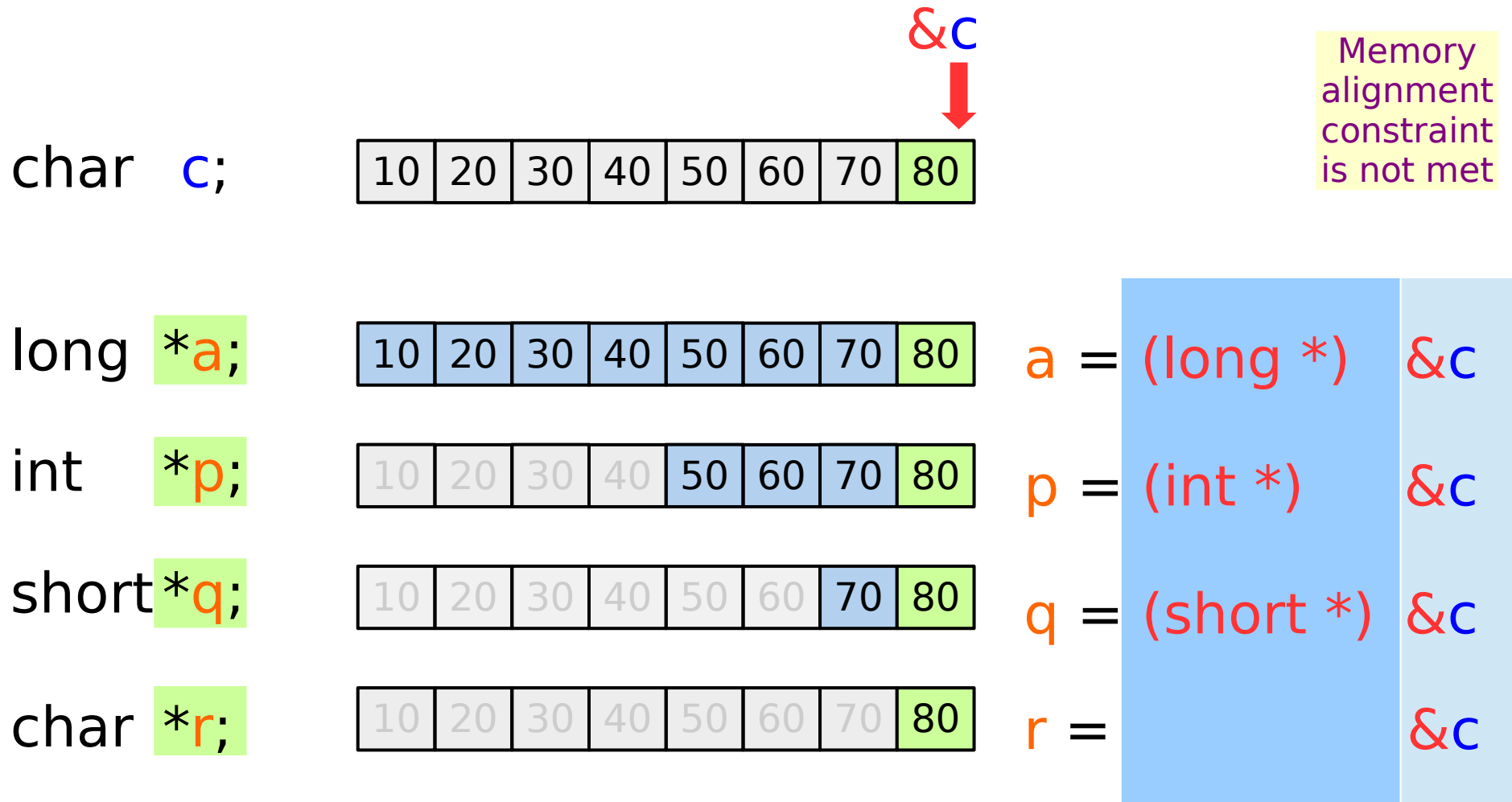
(char \*)

&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;
```

*read only integer value*

```
int * const q ;
```

*read only integer pointer*

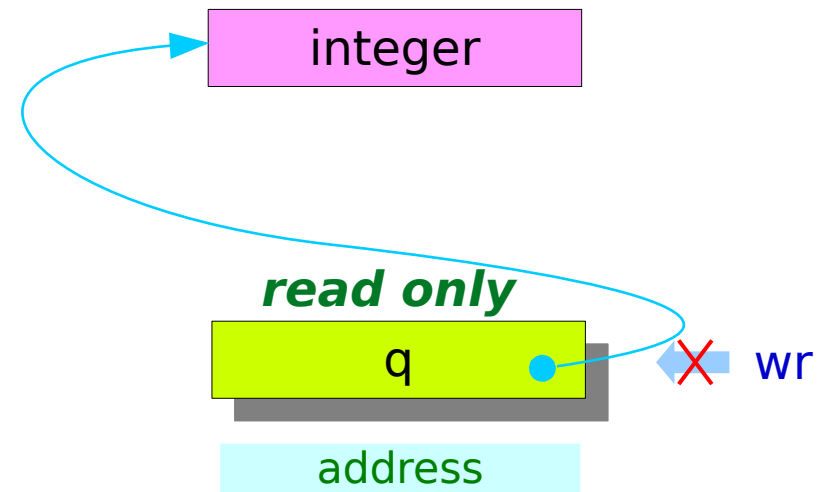
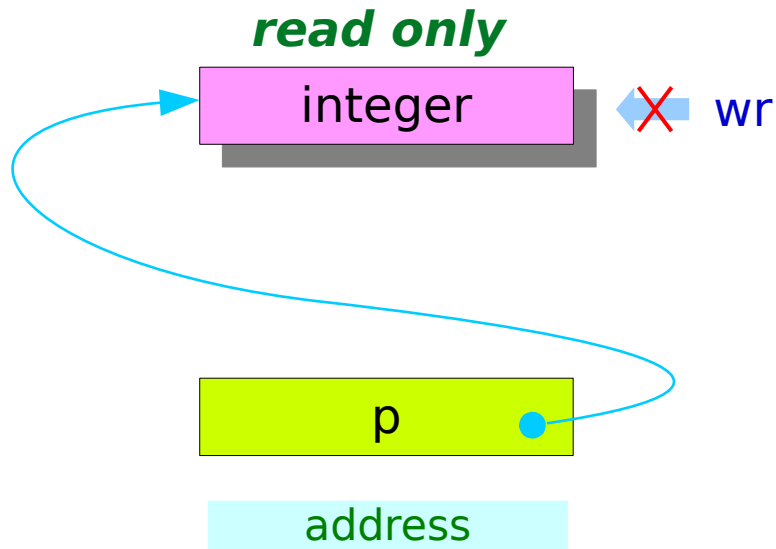
```
const int * const r ;
```

*read only integer value  
read only integer pointer*

# const type, const pointer type (2)

`const int * p;`

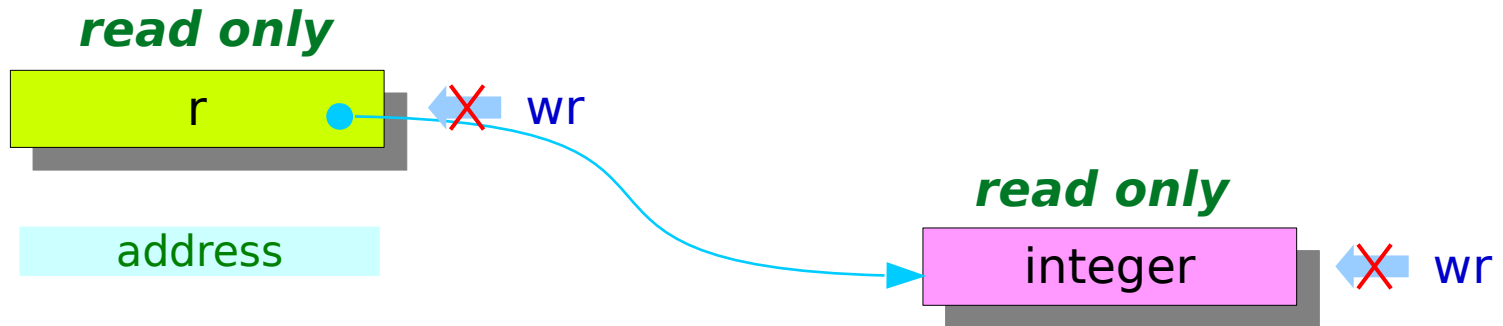
`int * const q;`





# 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