

# Pointers (1A)

---

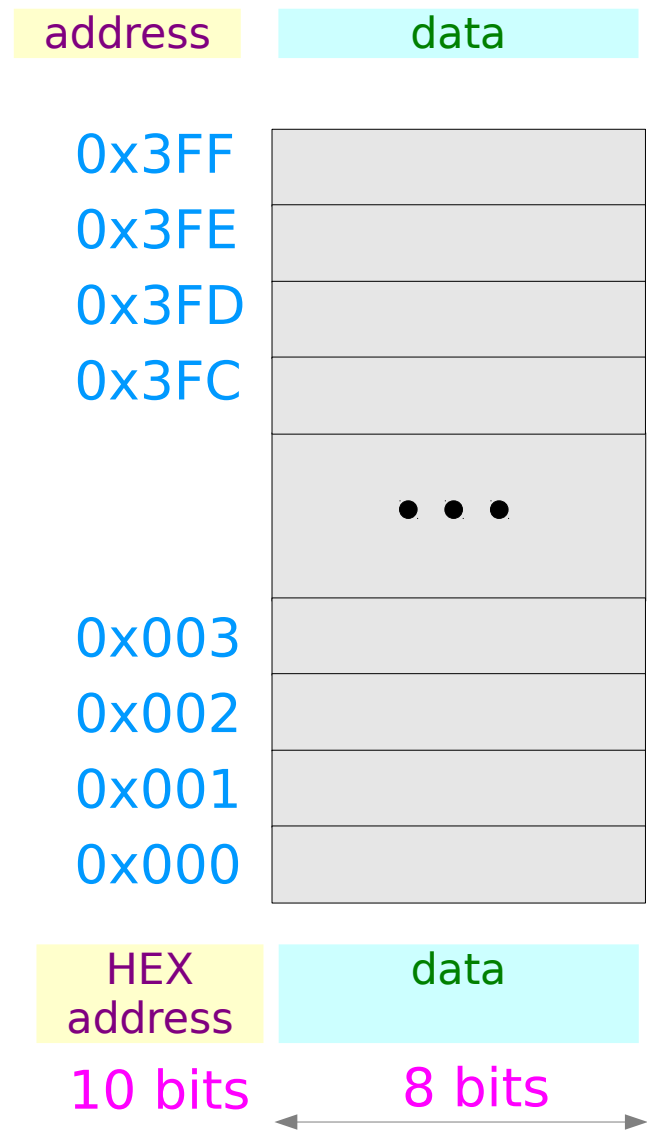
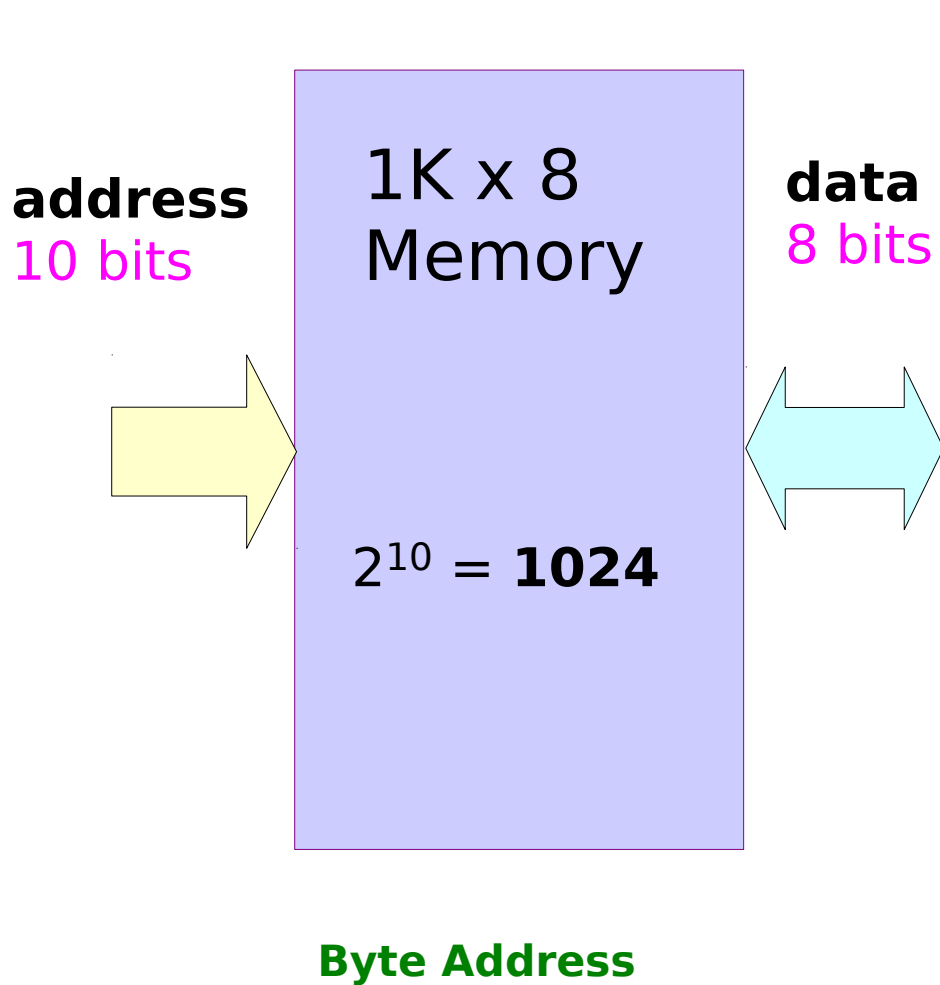
Copyright (c) 2010 - 2018 Young W. Lim.

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*

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

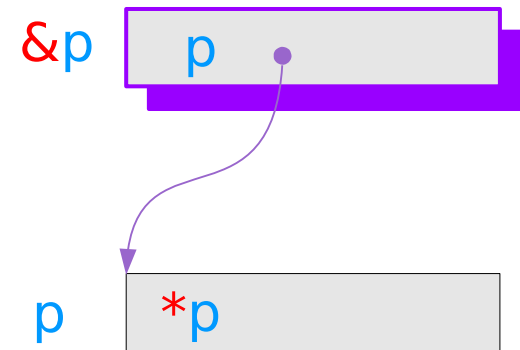
**type**      **variable**

```
int * p;
```

*pointer to int*

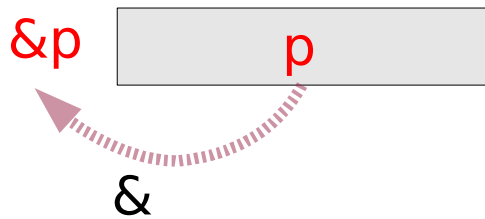
```
int * p;
```

*int*

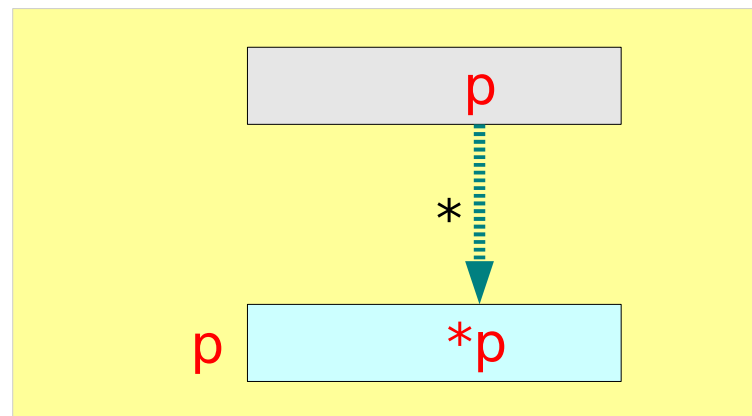
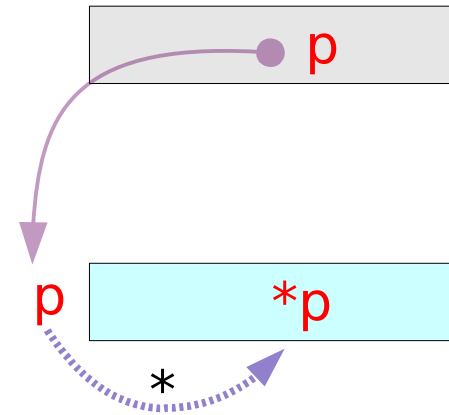


# Dereferencing

*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





# Assignment of an address

```
int a;
```

```
int *p;
```

address

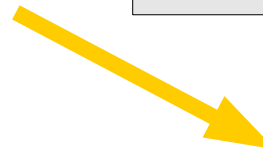
data

&a

a = 111

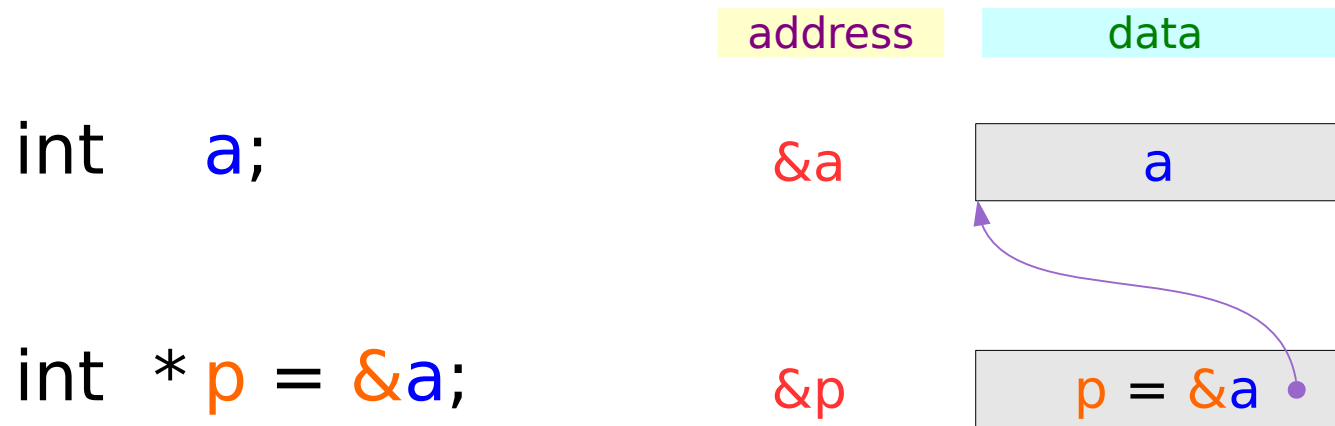
&p

p = \_\_\_\_\_



p = &a;

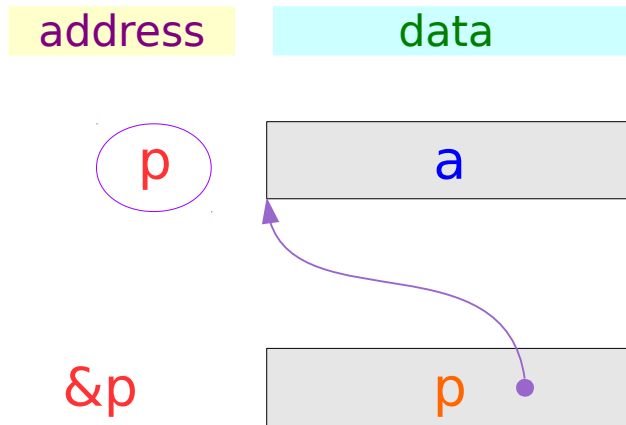
# Variables with initializations



# Pointed addresses : p

```
int a;
```

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

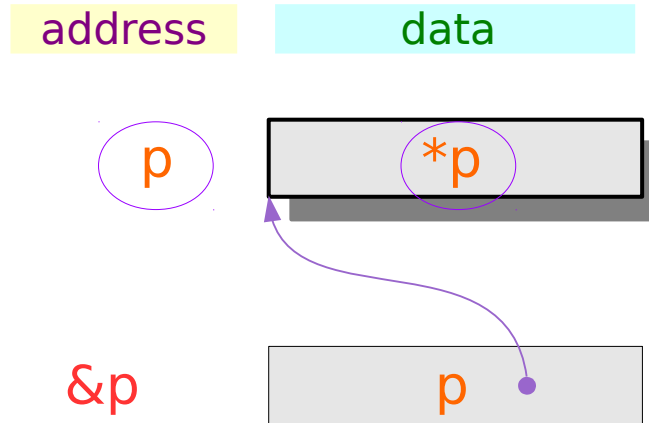


$p \equiv \&a$

# Dereferenced Variable : \*p

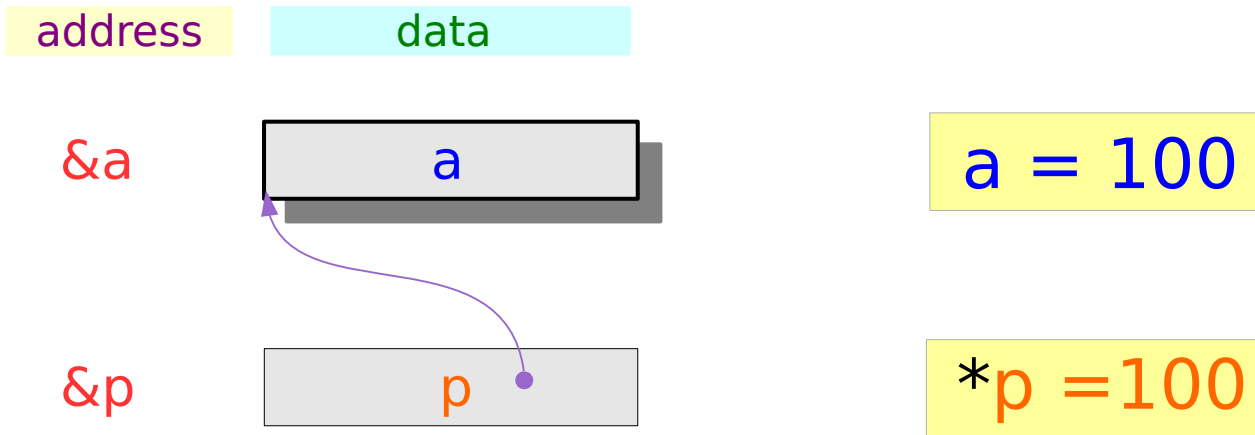
```
int a;
```

```
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

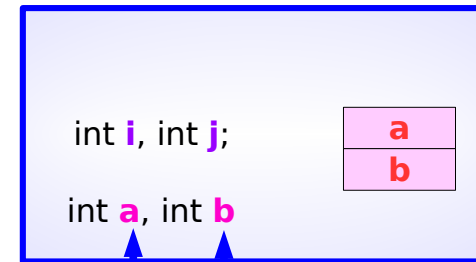
```
int main ()  
{  
    int x, int y;  
    ...  
    ...  
    func1 ( 10, 20 );  
    ...  
    ...  
}
```

**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

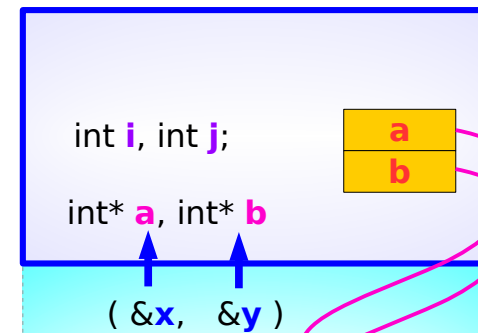
**a**=&**x**  
**b**=&**y**

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

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

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

**func1**'s  
Stack  
Frame

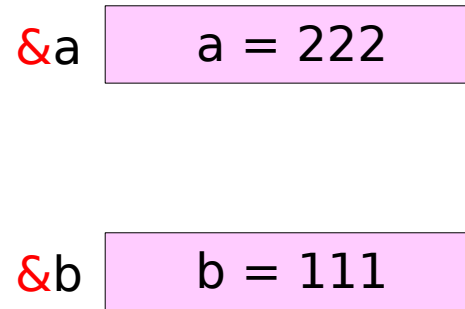
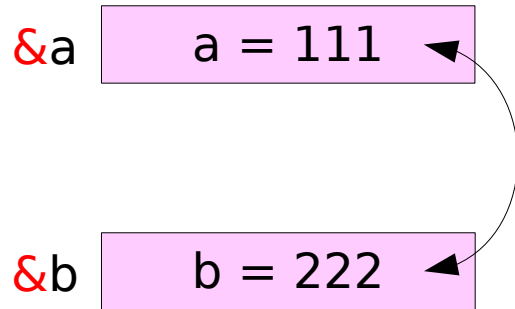


**main**'s  
Stack  
Frame

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

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

# Swapping integers



```
int a, b;
```

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

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

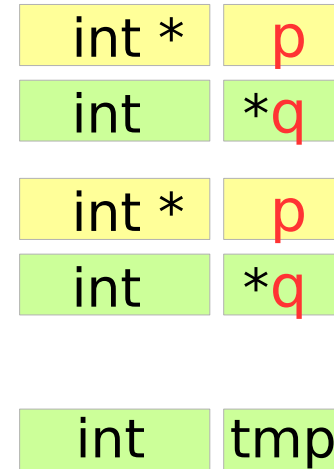
function call

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

```
int *
```

```
int
```

---

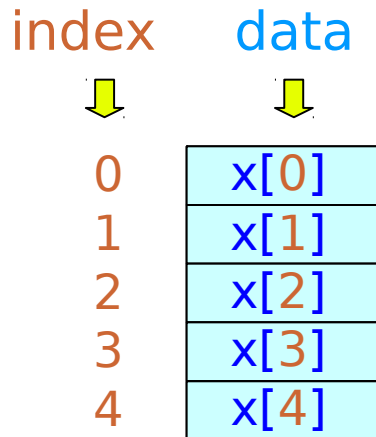
# Arrays

# Accessing array elements - using an address

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

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

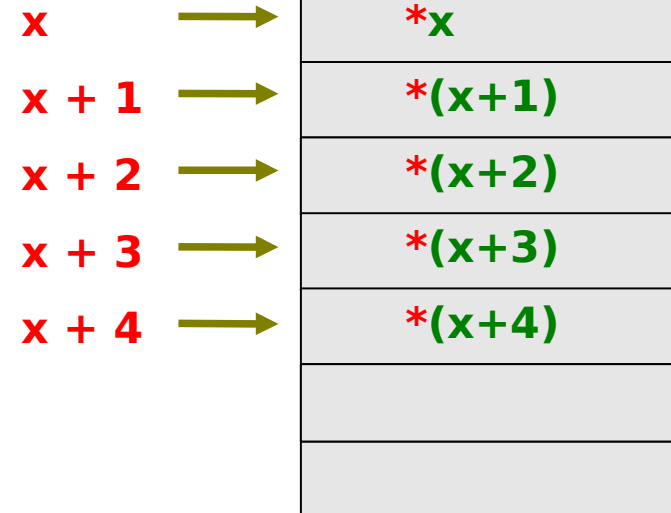
5 int variables



cannot change  
address x  
(constant)

address

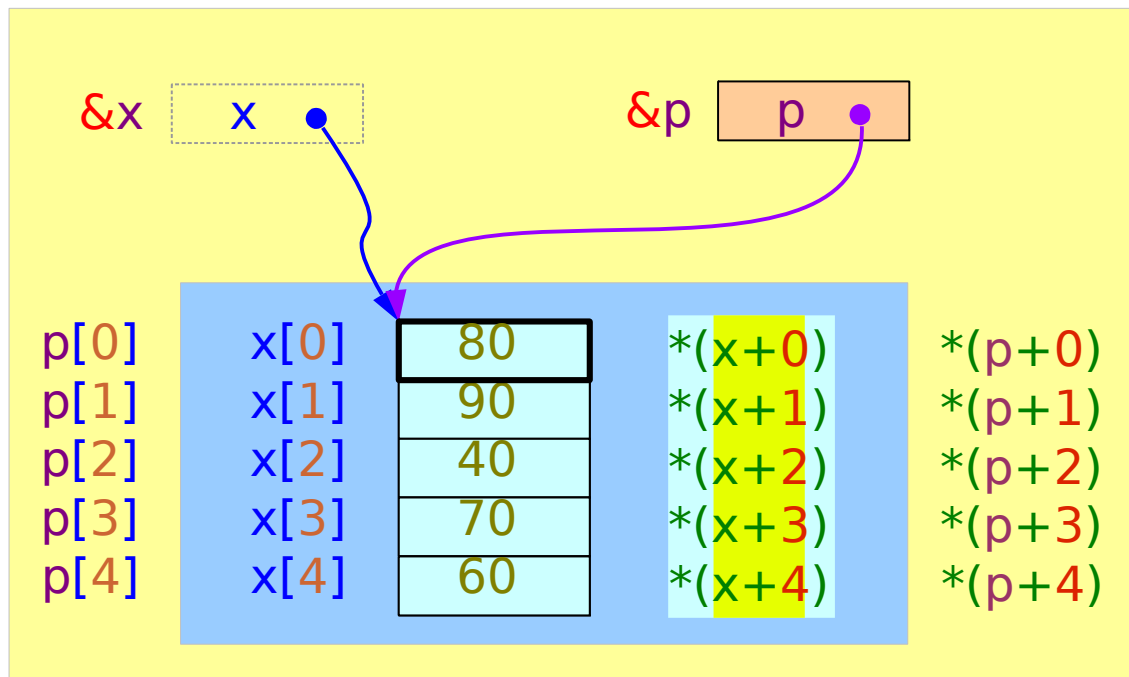
data



# 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

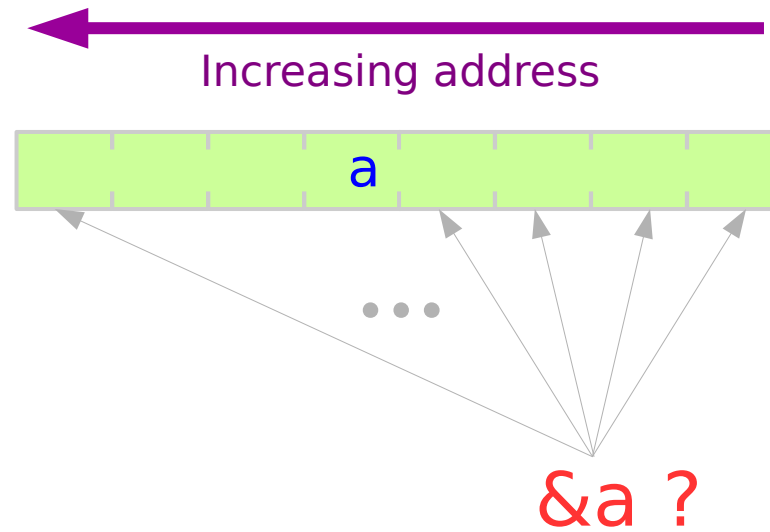
---

Byte Address  
Little Endian  
Big Endian



# Byte Address

```
long a;
```



# Numbers in Positional Notation

```
long a = 0x1020304050607080;
```

8 (bytes)

$a_7$   $a_6$   $a_5$   $a_4$   $a_3$   $a_2$   $a_1$   $a_0$

Most Significant Byte  $a_7 = 0x10 \dots 16^7$  the highest weight

$a_6 = 0x20 \dots 16^6$

$a_5 = 0x30 \dots 16^5$

$a_4 = 0x40 \dots 16^4$

$a_3 = 0x50 \dots 16^3$

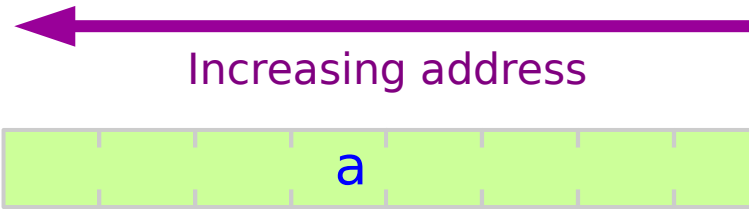
$a_2 = 0x60 \dots 16^2$

$a_1 = 0x70 \dots 16^1$

Least Significant Byte  $a_0 = 0x80 \dots 16^0$  the lowest weight

# Little / Big Endian

long a;



MSByte Little Endian LSByte



$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$

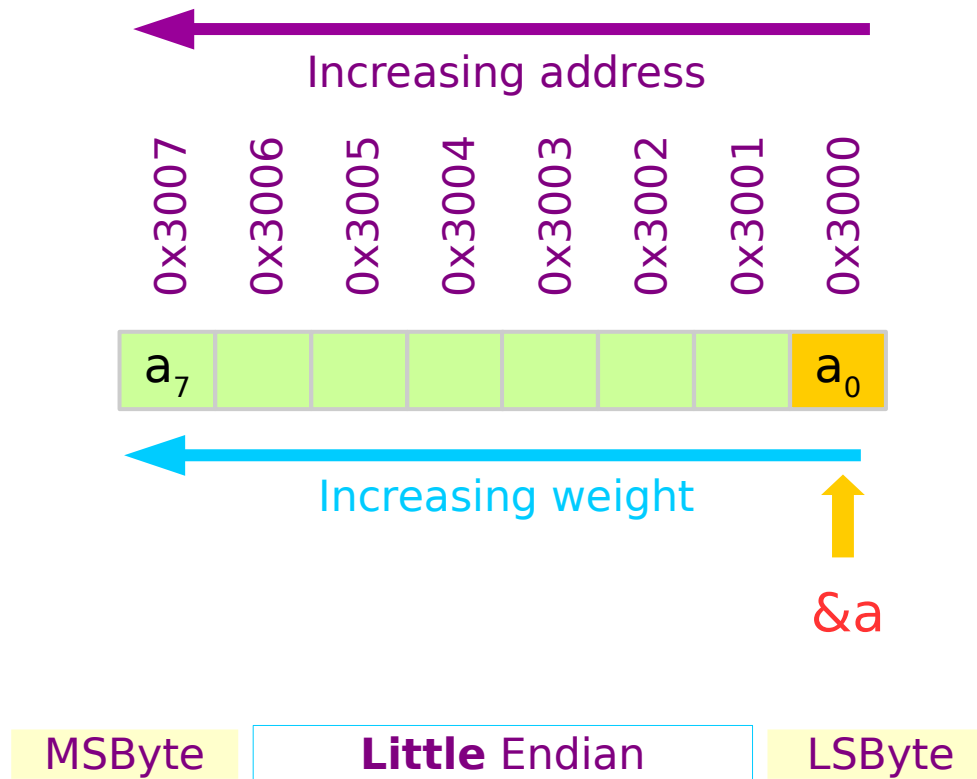
LSByte Big Endian MSByte



$a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7$

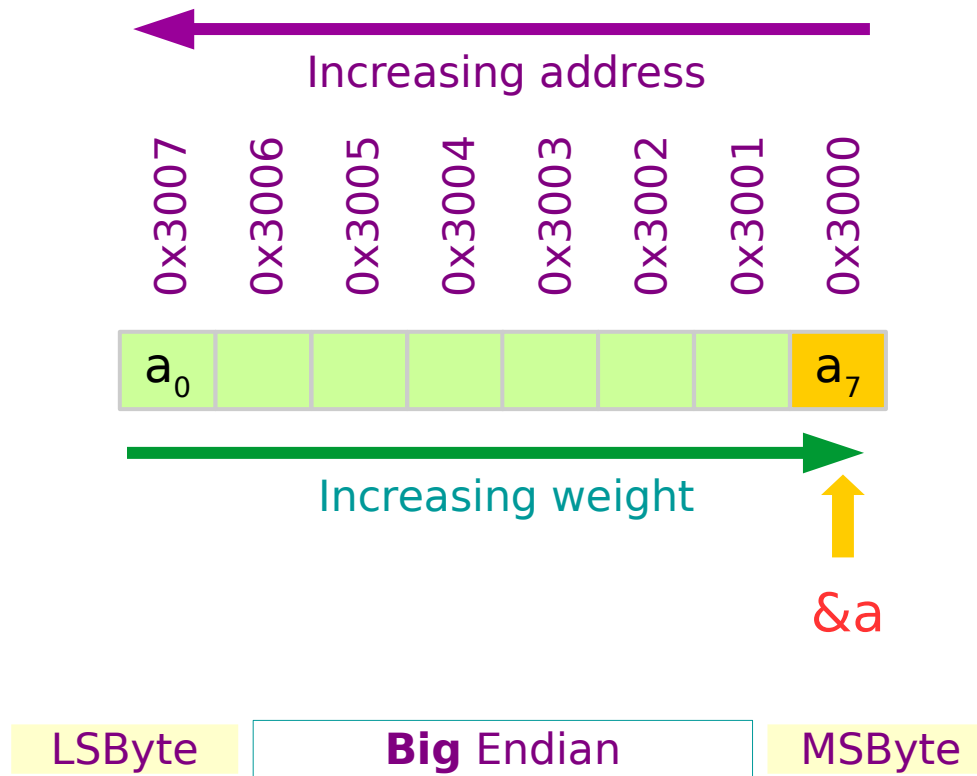
# Little Endian Byte Address Example

long a;



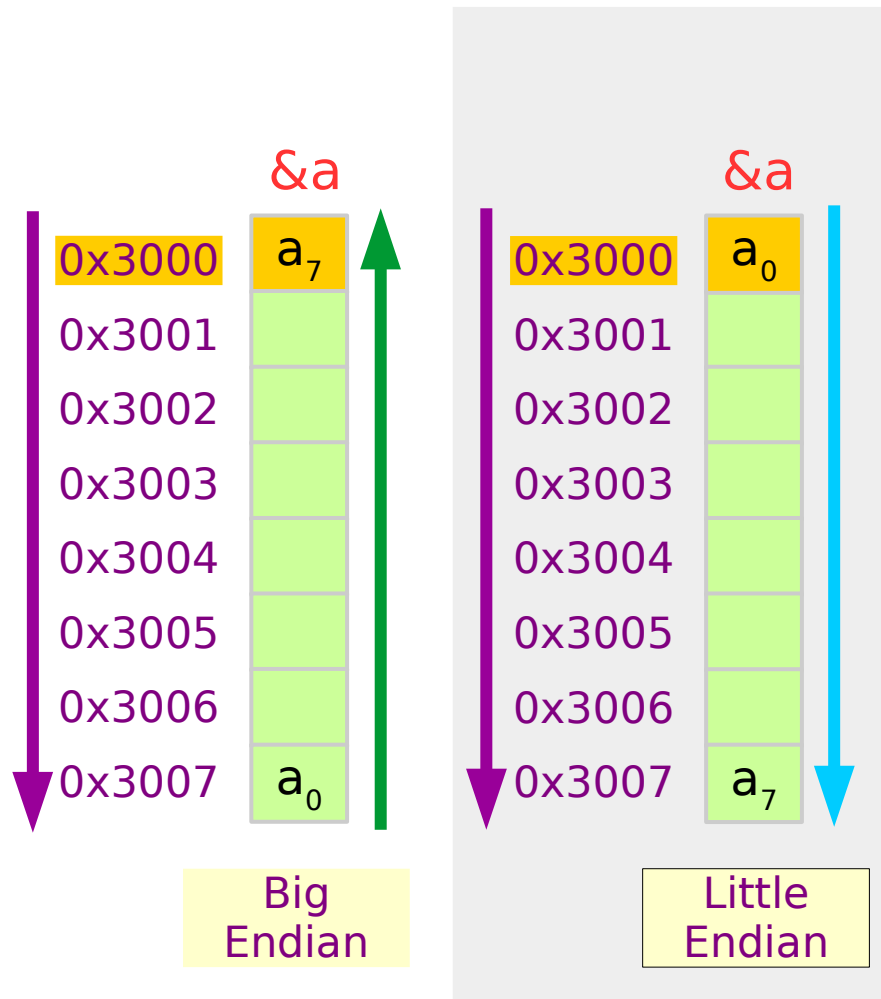
# Big Endian Byte Address Example

long a;

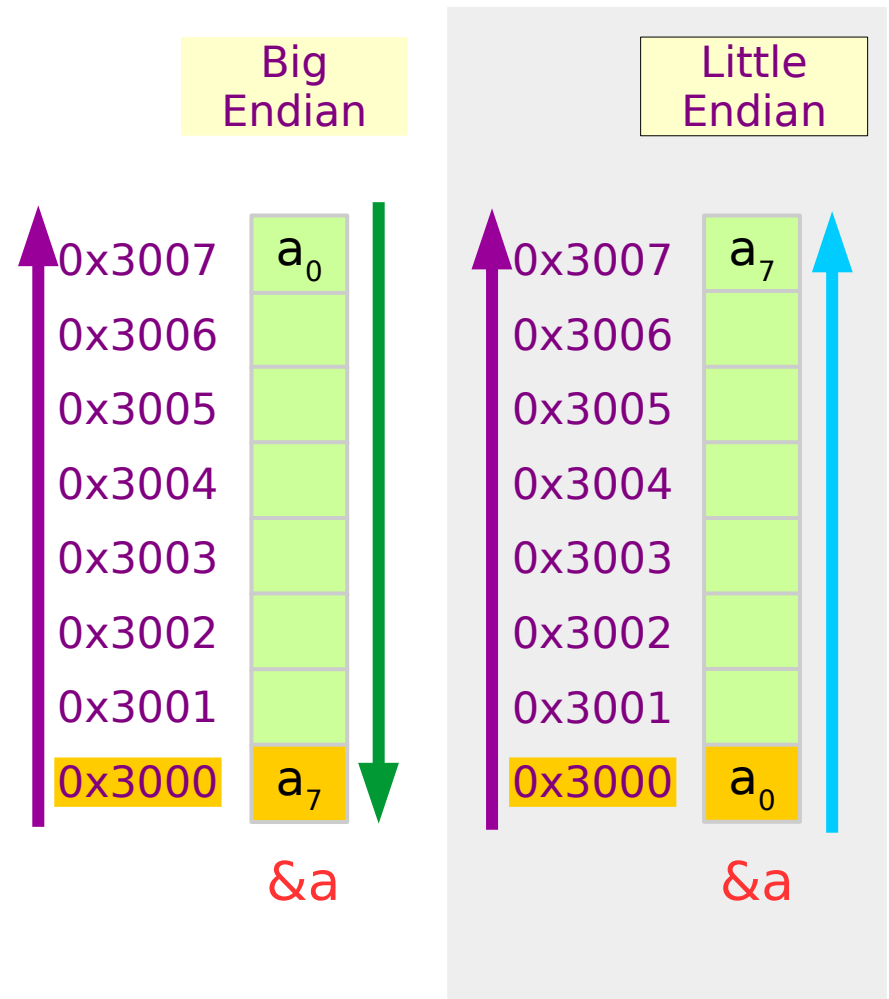


# Representations of Endianness

**downward**, increasing address



**upward**, increasing address



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

# Little / Big Endian Processors

| Processor                    | Endianness          |
|------------------------------|---------------------|
| <b>Motorola 68000</b>        | Big Endian          |
| <b>PowerPC (PPC)</b>         | Big Endian          |
| Sun Sparc                    | Big Endian          |
| IBM S/390                    | Big Endian          |
| <b>Intel x86 (32 bit)</b>    | Little Endian       |
| <b>Intel x86_64 (64 bit)</b> | Little Endian       |
| Dec VAX                      | Little Endian       |
| Alpha                        | (Big/Little) Endian |
| <b>ARM</b>                   | (Big/Little) Endian |
| IA-64 (64 bit)               | (Big/Little) Endian |
| MIPS                         | (Big/Little) Endian |

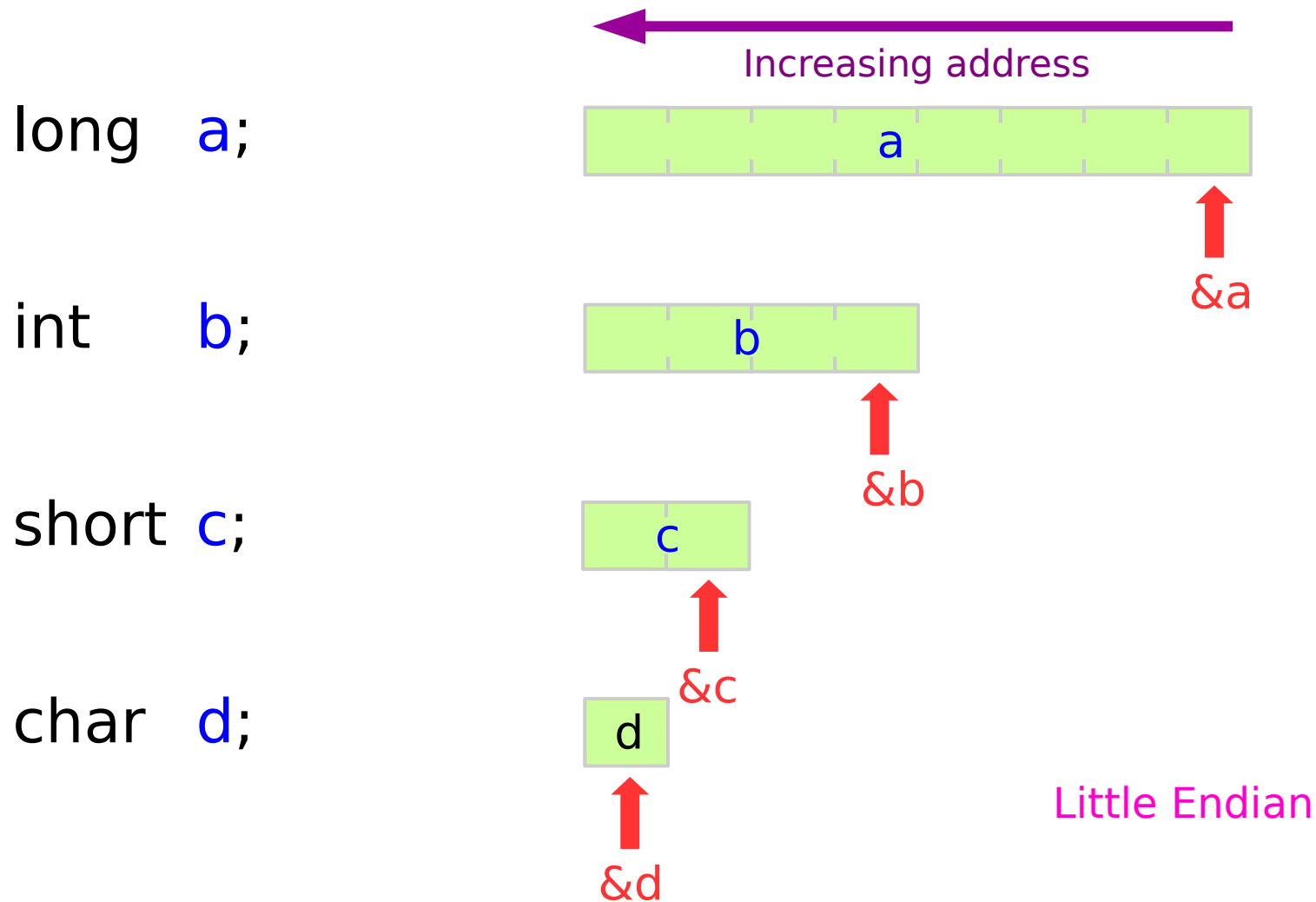
<http://www.yolinux.com/TUTORIALS/Endian-Byte-Order.html>

---

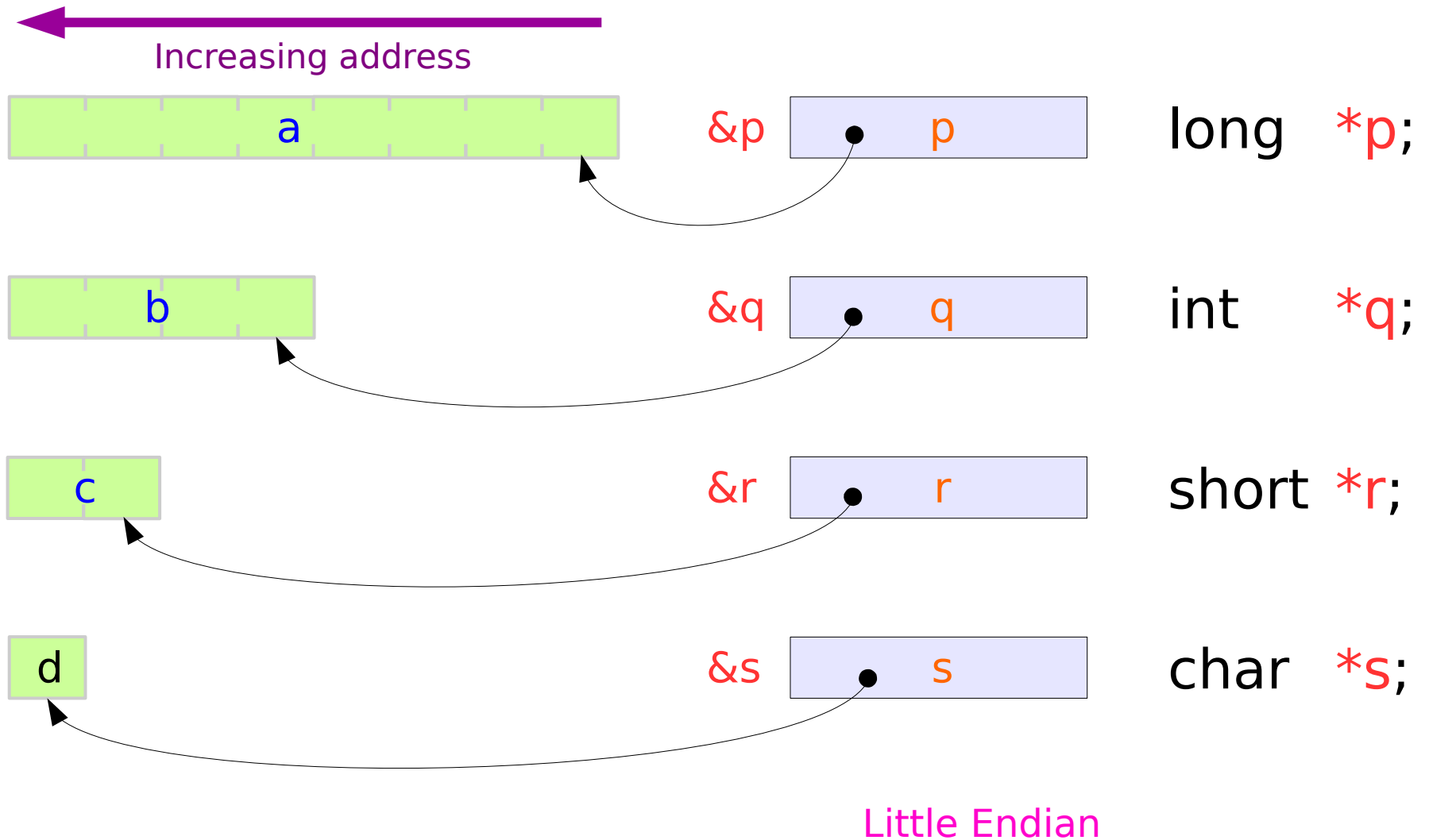
# Pointer Types



# Integer Type Variables and Their Addresses



# Points to the LSByte



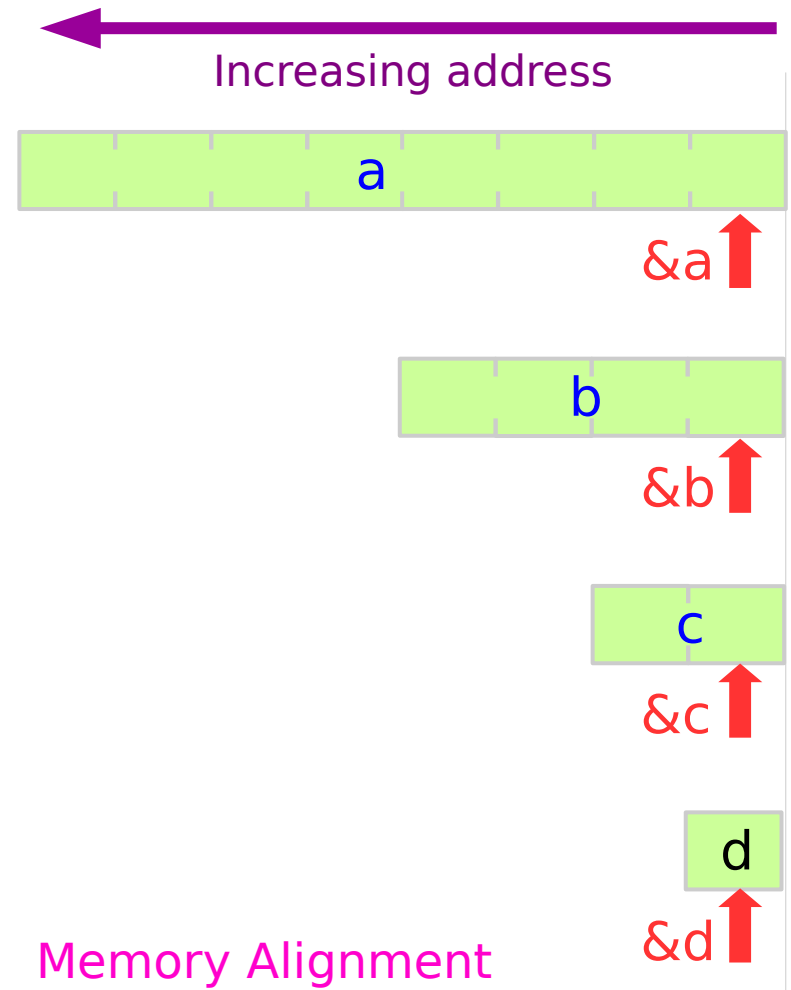
# Sizes of Integer Types

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

`int b;`    `sizeof(int)` → 4 (bytes)

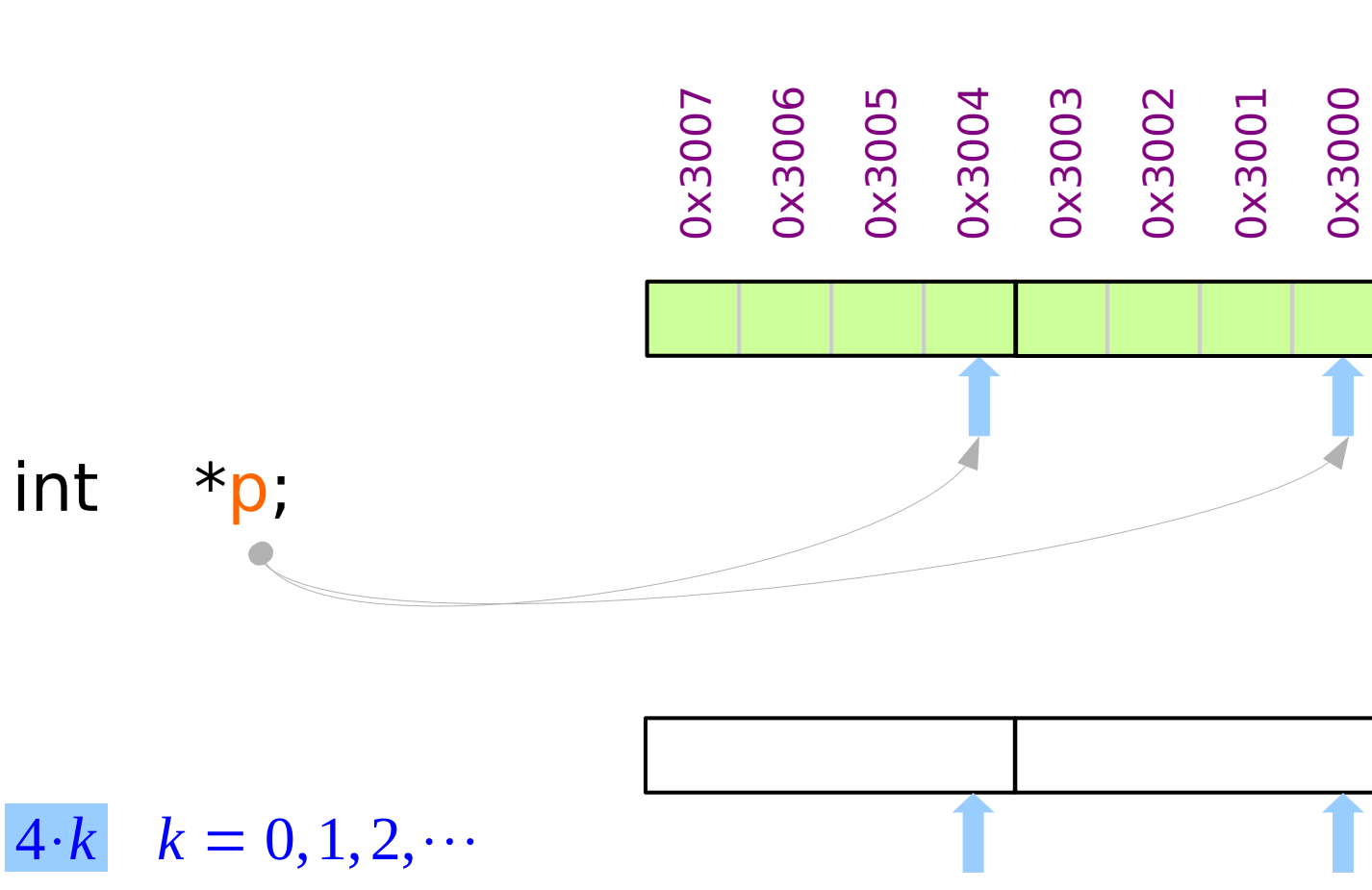
`short c;`    `sizeof(short)` → 2 (bytes)

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

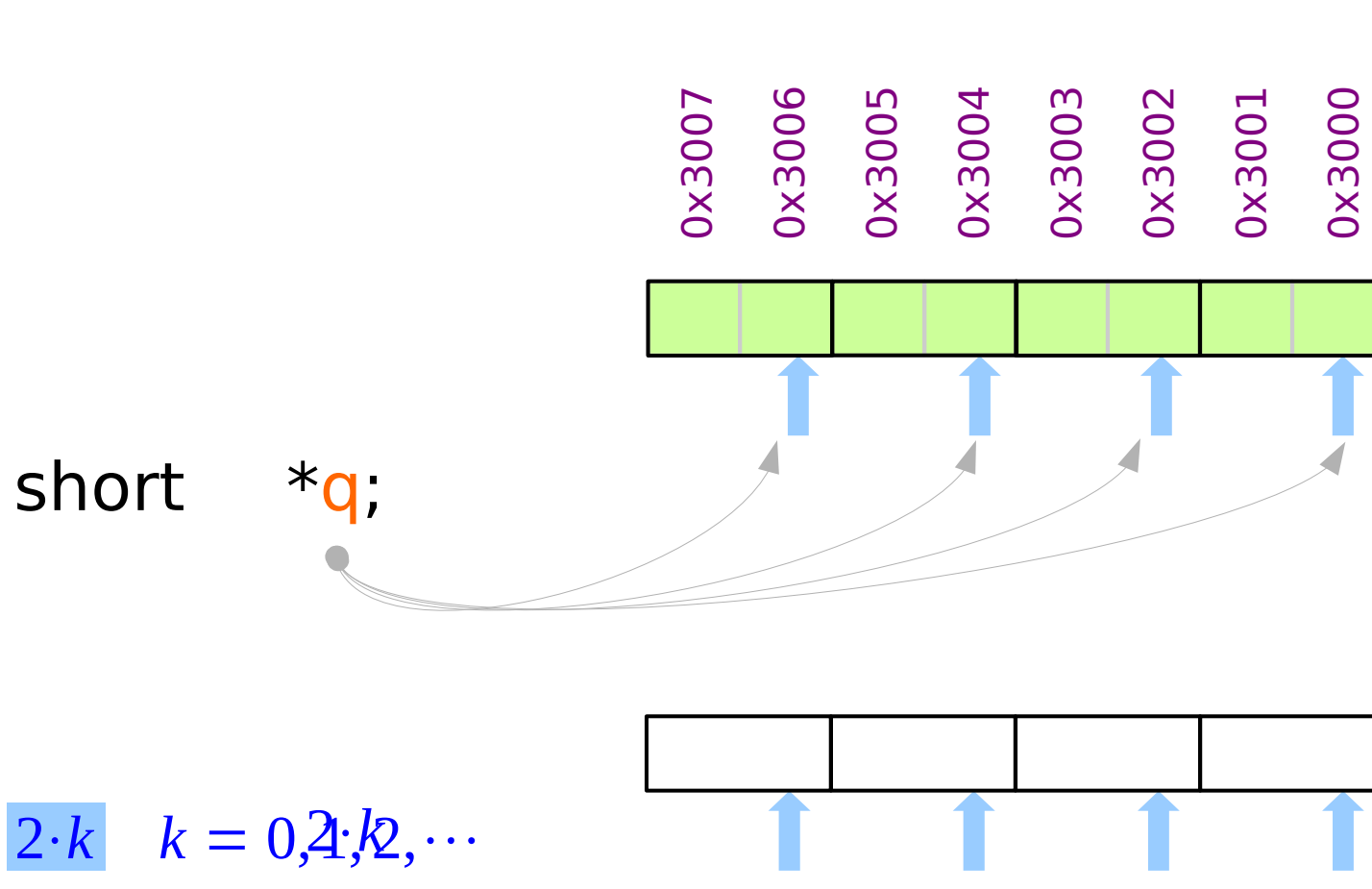


Memory Alignment  
in the Little Endian

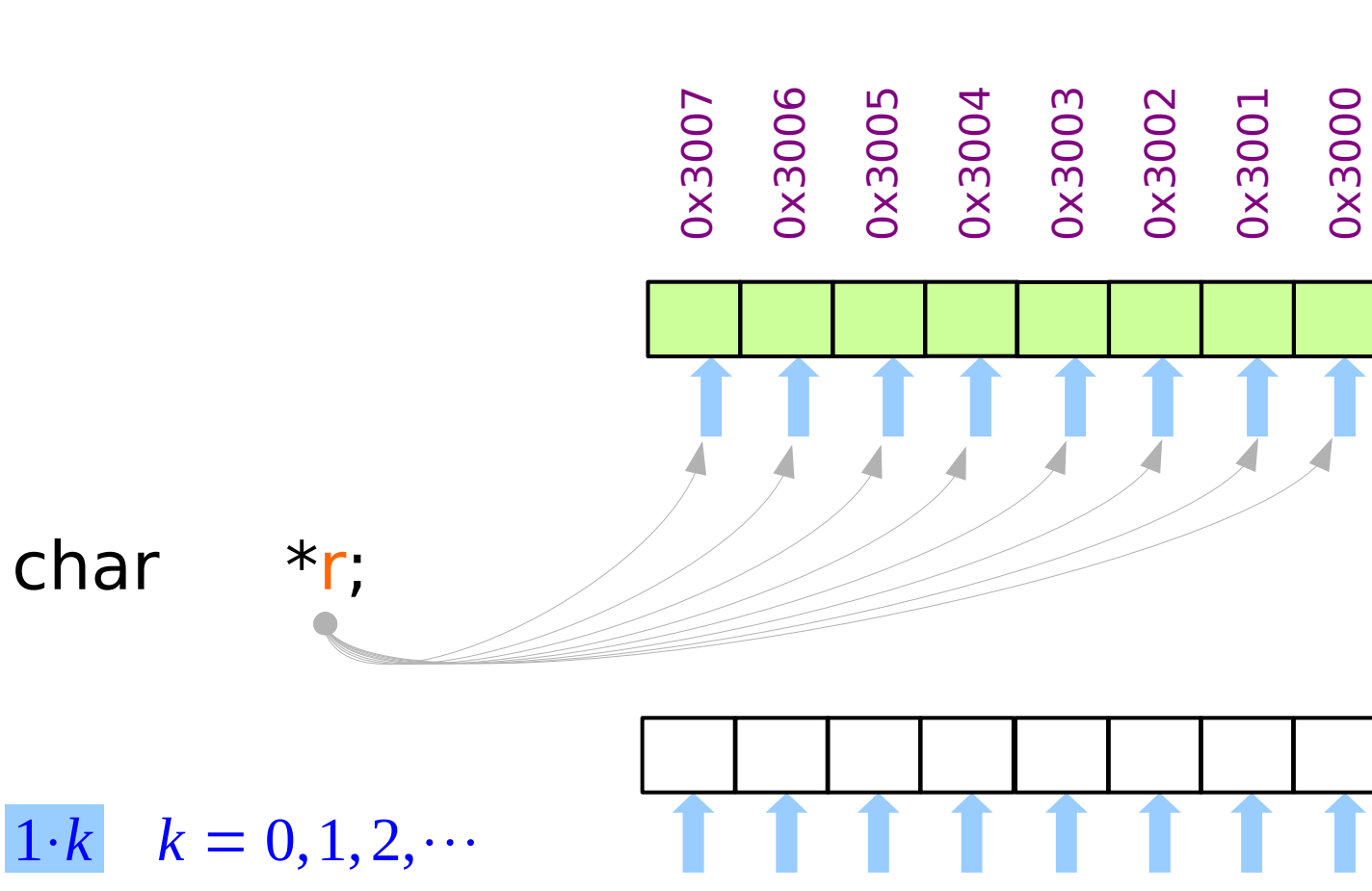
# Possible addresses for integer values



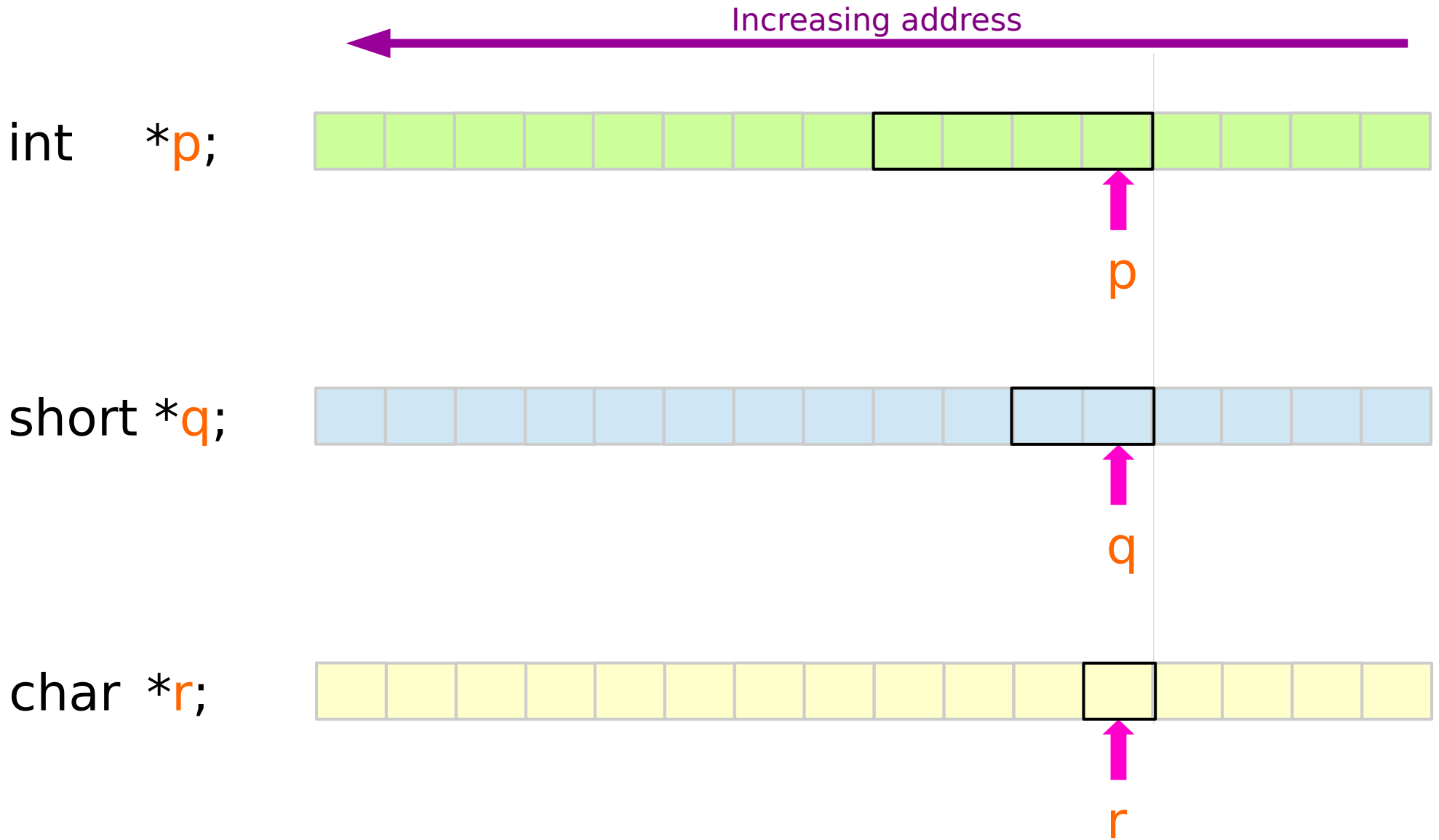
# Possible addresses for short values



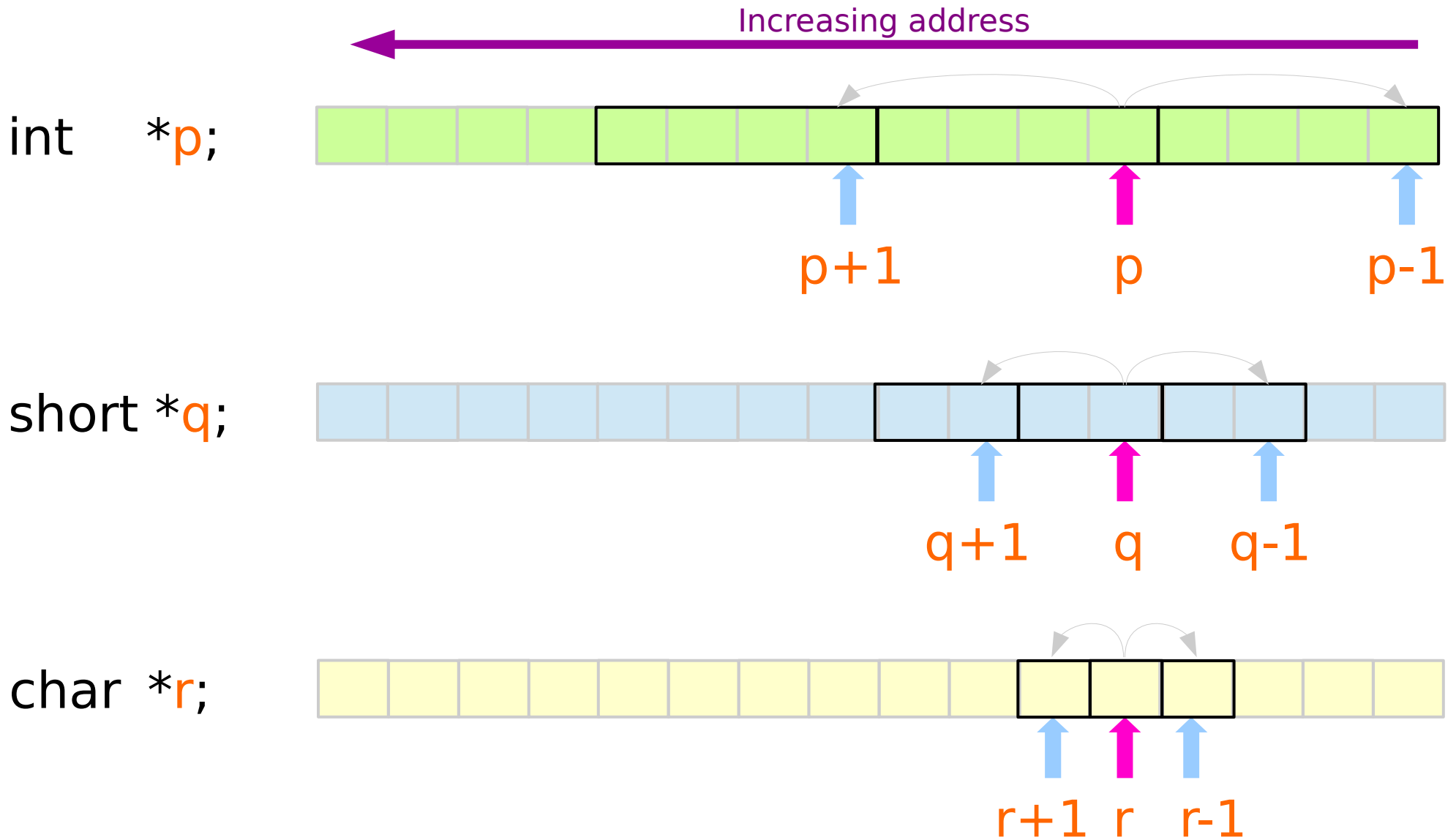
# Possible address for char values



# Pointed Addresses



# Incrementing / decrementing pointers





# Memory Alignment (1) - allocation of variables

Memory Alignment:  
the data address is a  
multiple of the data size.

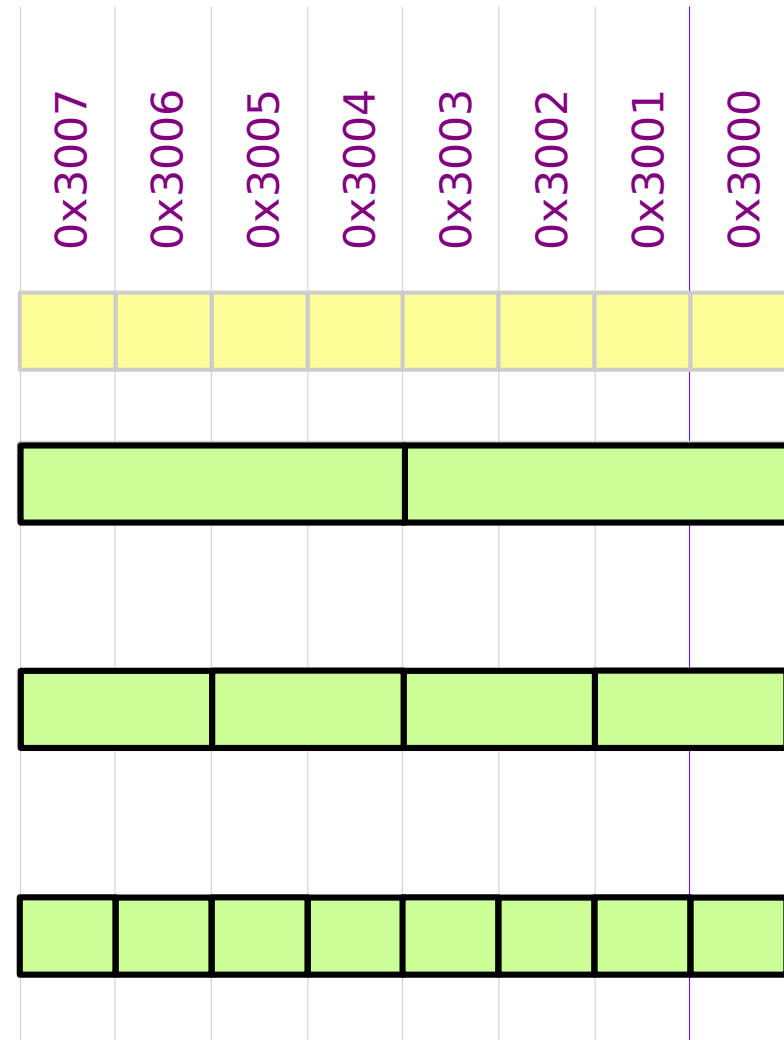
enforced by compilers

efficient memory access

int a;

short b;

char c;



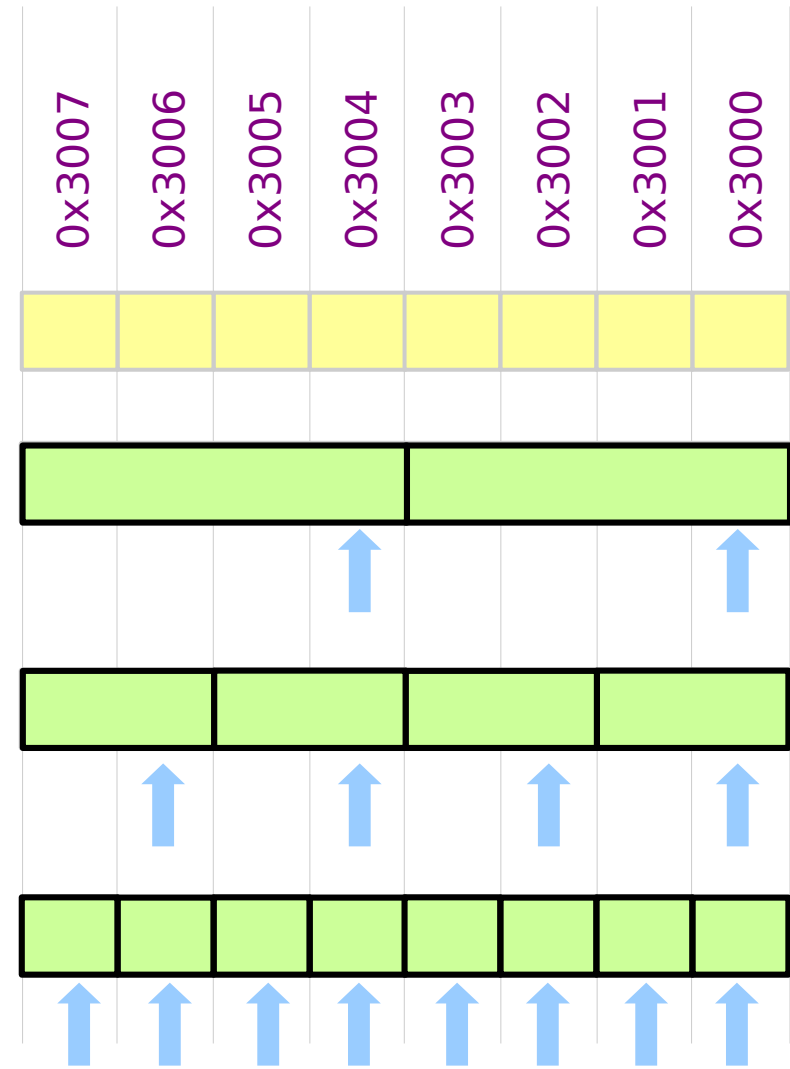
# Memory Alignment (2) - integer multiple addresses

$k = 0, 1, 2, \dots$

*integer addresses* =  $4 \cdot k$

*short addresses* =  $2 \cdot k$

*character addresses* =  $1 \cdot k$

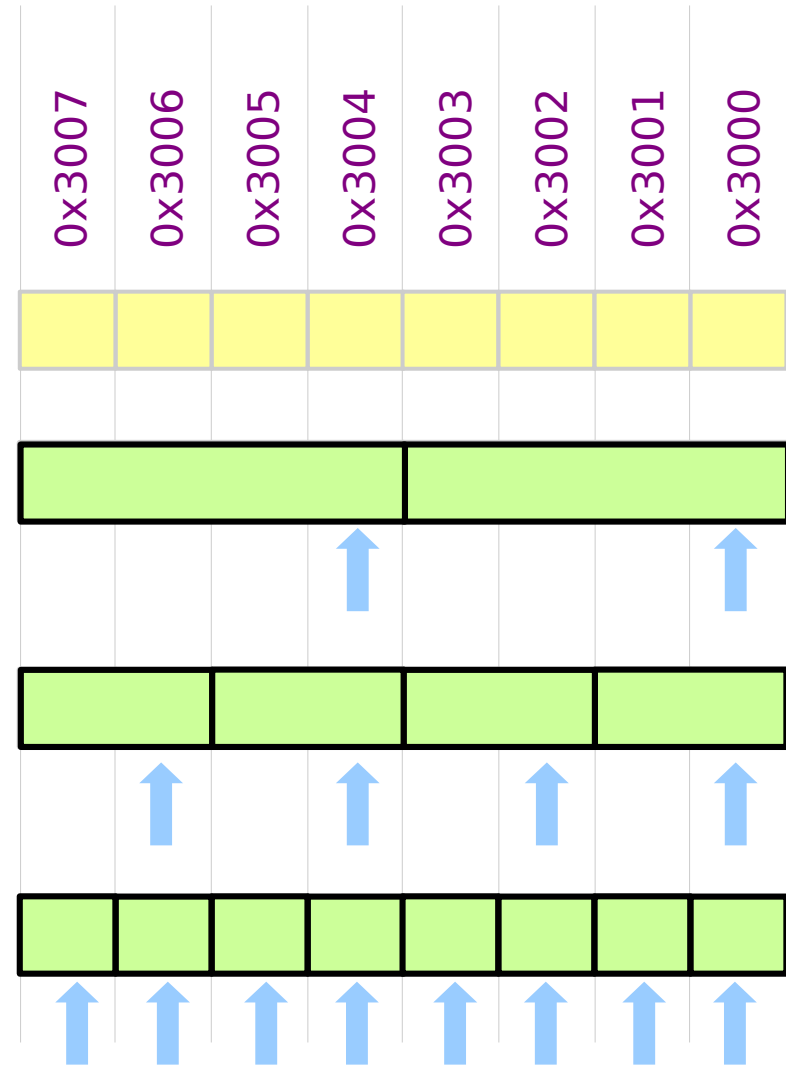


# Memory Alignment (3) - pointable addresses

`int *p;`

`short *q;`

`char *r;`



# Memory Alignment (4) - non-pointed addresses

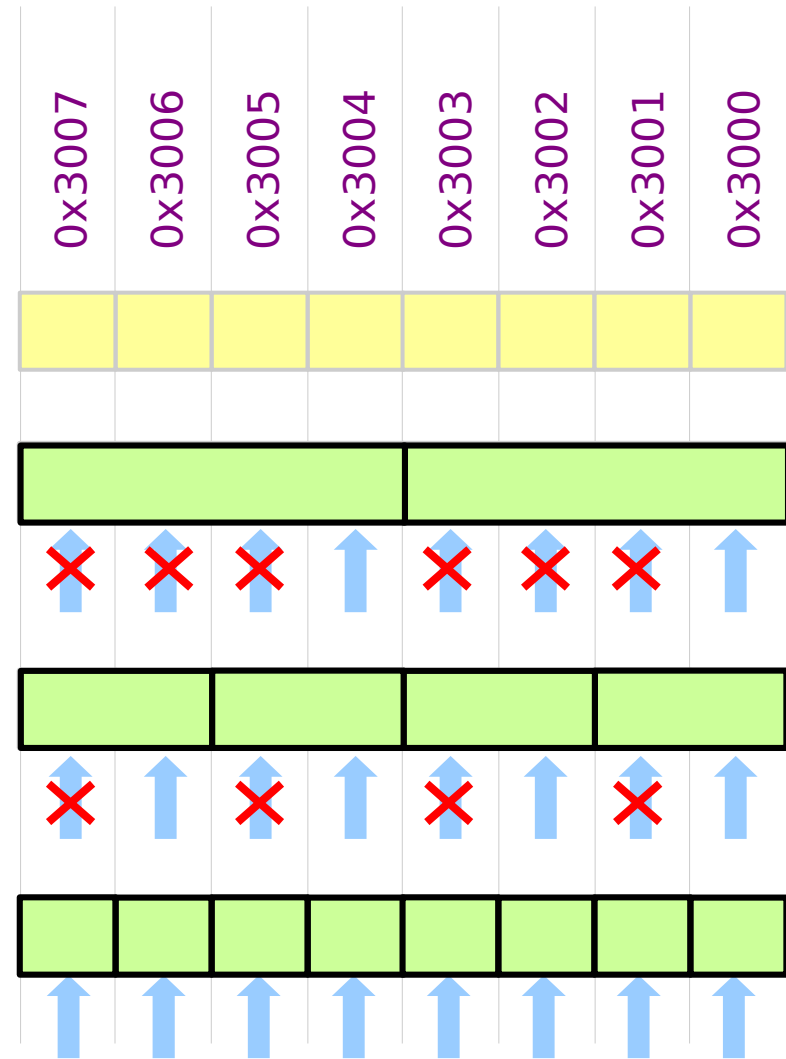
int \*p;

$$4 \cdot k + 1, 2, 3$$

short \*q;

$$2 \cdot k + 1$$

char \*r;



# Memory Alignment (5) - breaking alignment

Memory access is still possible  
but it takes **longer** to access

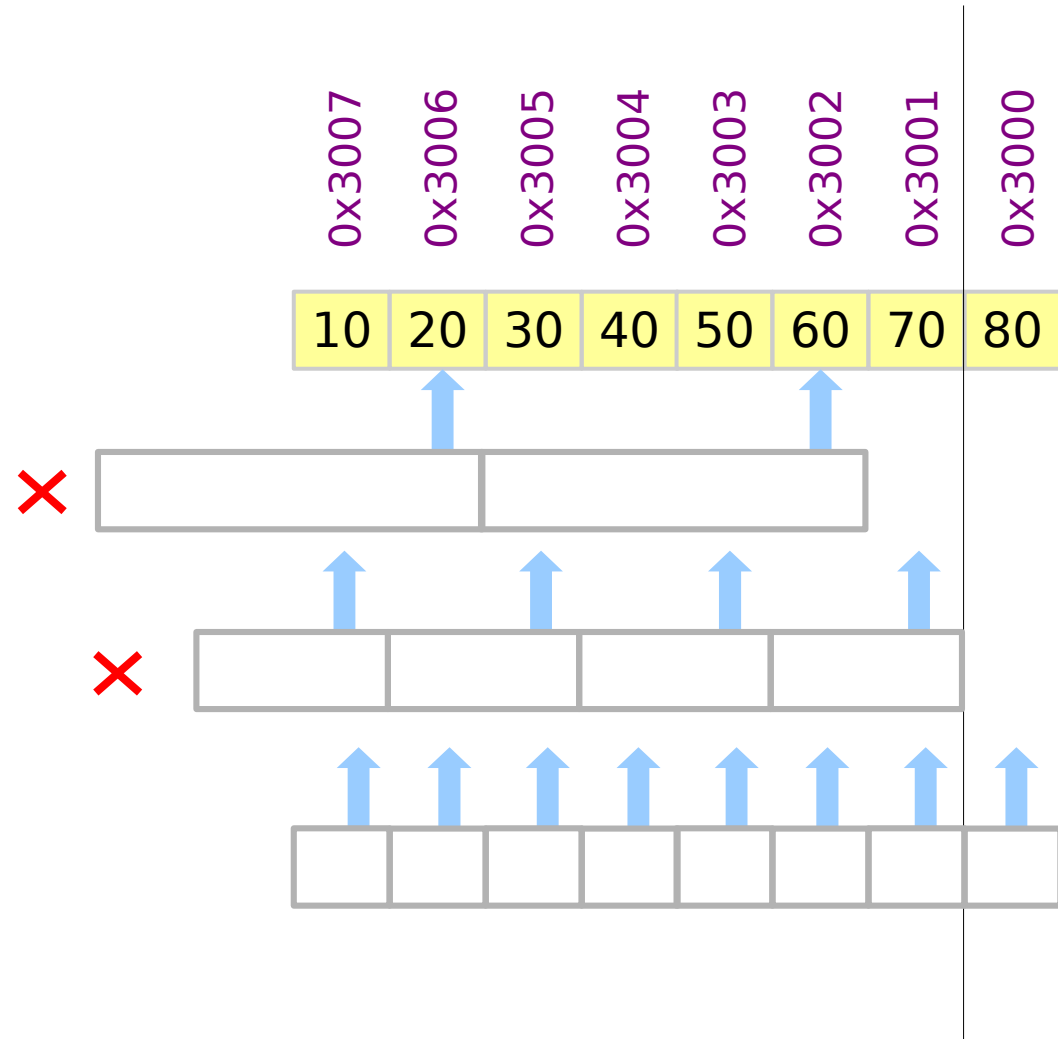
(Low Efficiency)

This can happen  
by using inadvertent  
pointer type casting

```
int *p;
```

```
short *q;
```

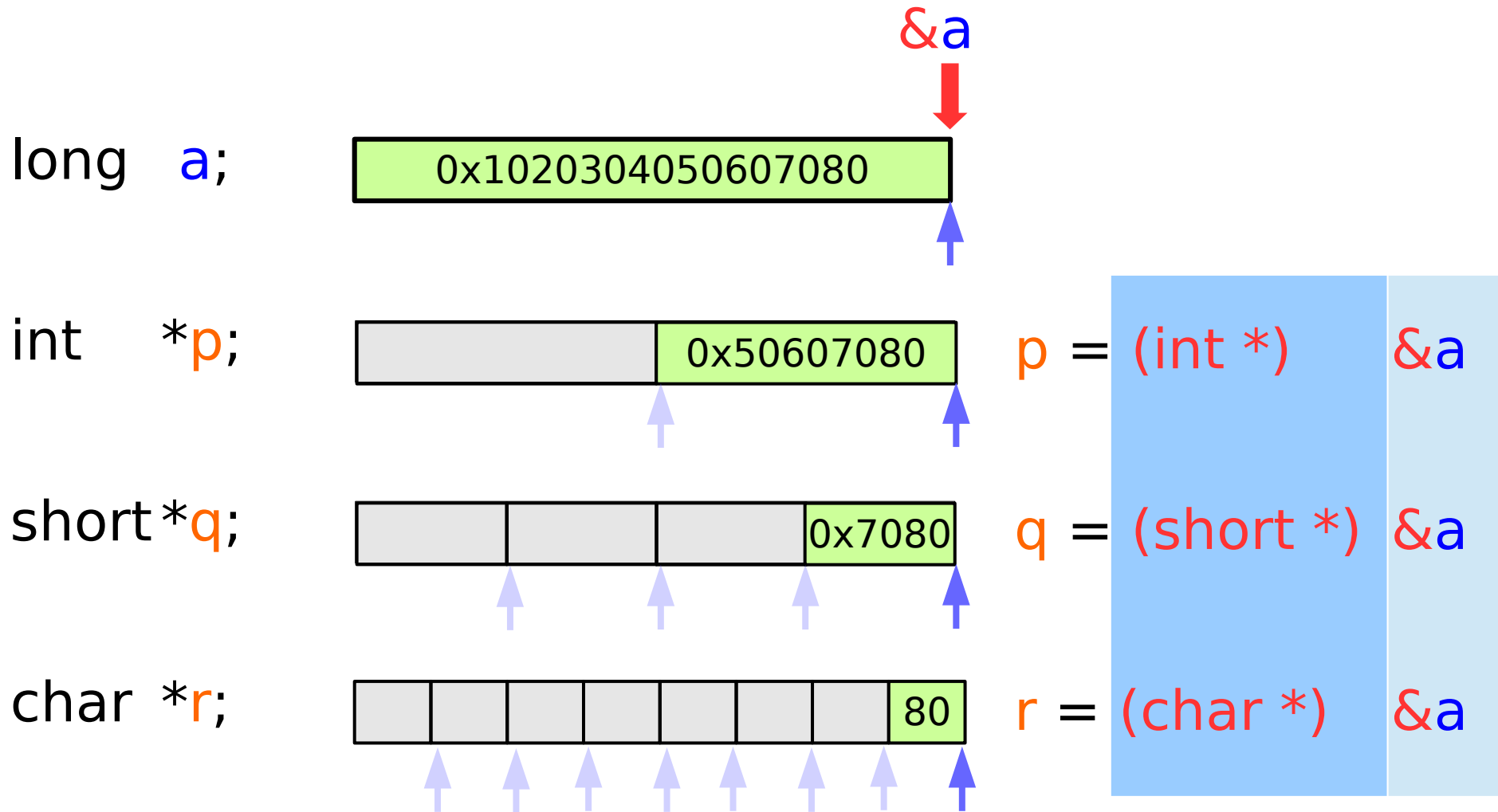
```
char *r;
```



---

# Pointer Type Cast

# Re-interpretation of memory data - case I



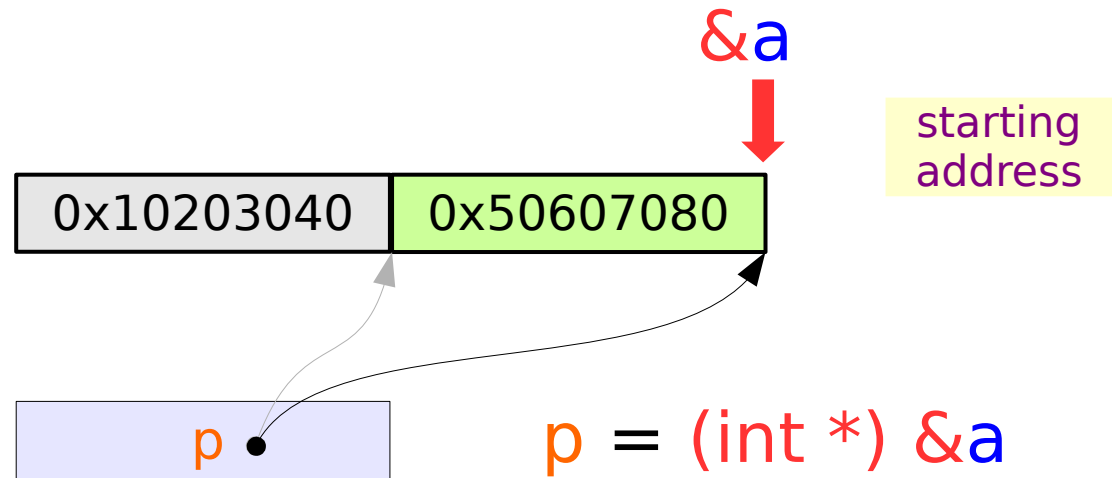
# Pointer Type Cast

```
long a;
```

```
int *p;
```

```
short *q;
```

```
char *r;
```





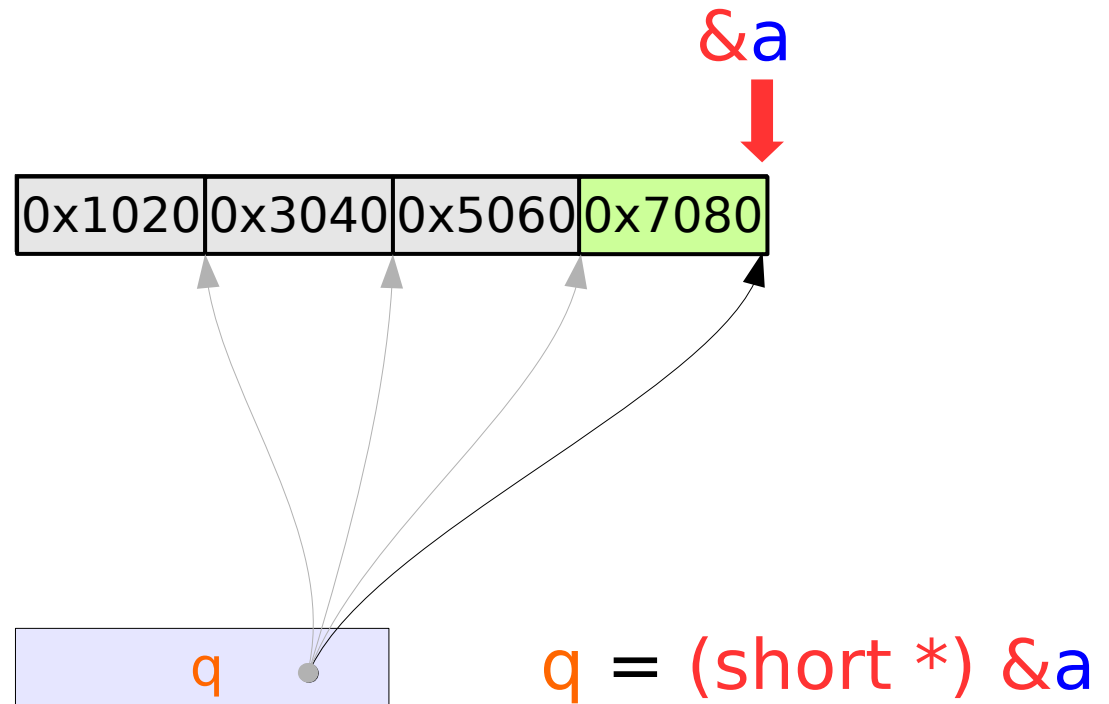
# Integer Pointer Types

long a;

int \*p;

short \*q;

char \*r;



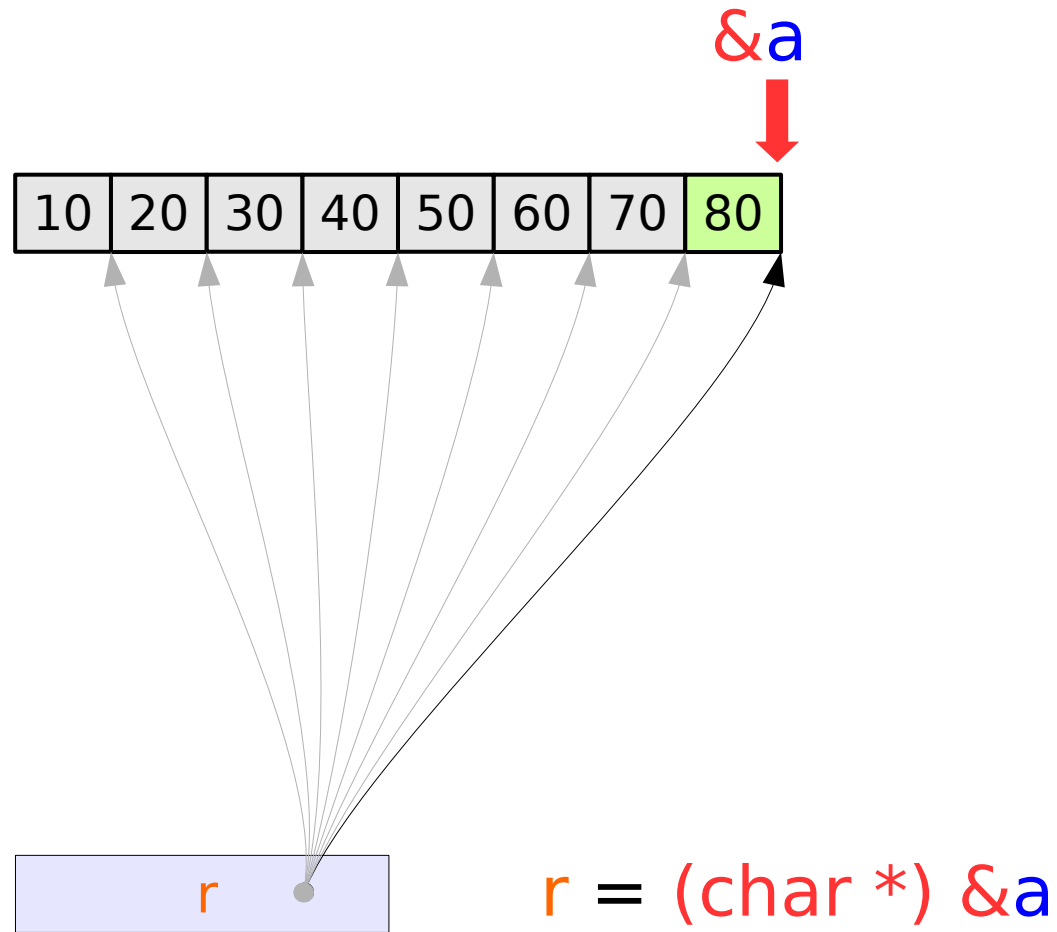
# Integer Pointer Types

long **a**;

int **\*p**;

short **\*q**;

char **\*r**;





---

# const pointers

# const type, const pointer type (1)

```
const int * p;
```

*read only integer value*

```
int * const q ;
```

*read only integer pointer*

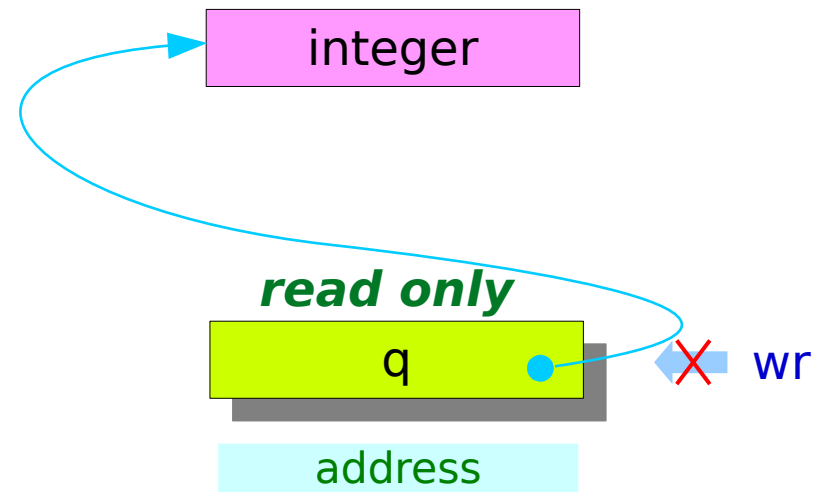
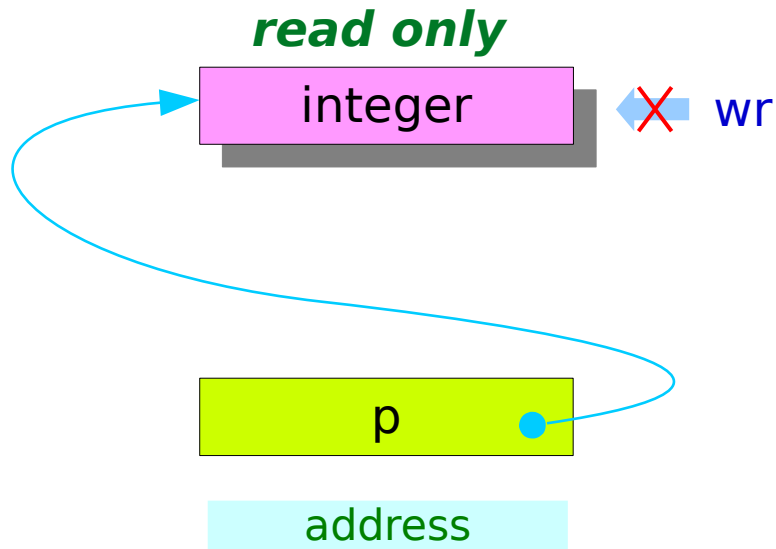
```
const int * const r ;
```

*read only integer value  
read only integer pointer*

# const type, const pointer type (2)

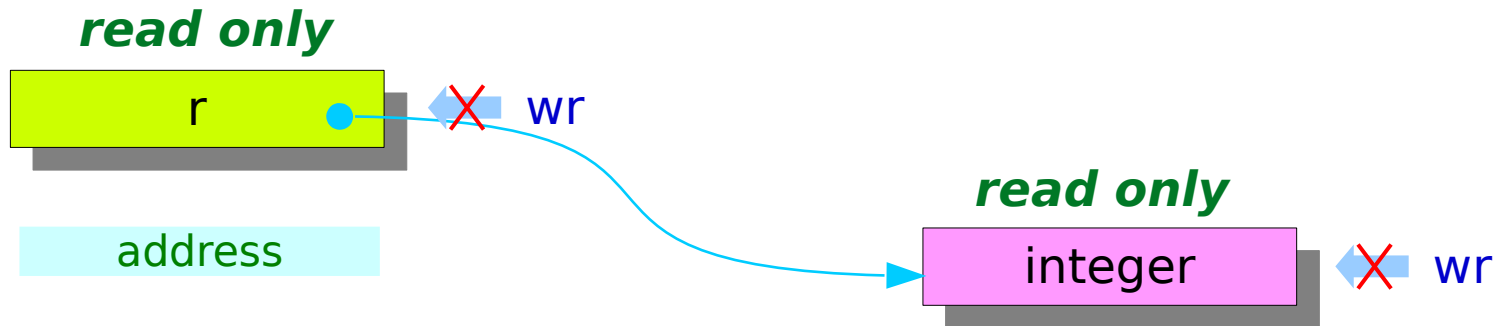
`const int * p;`

`int * const q;`

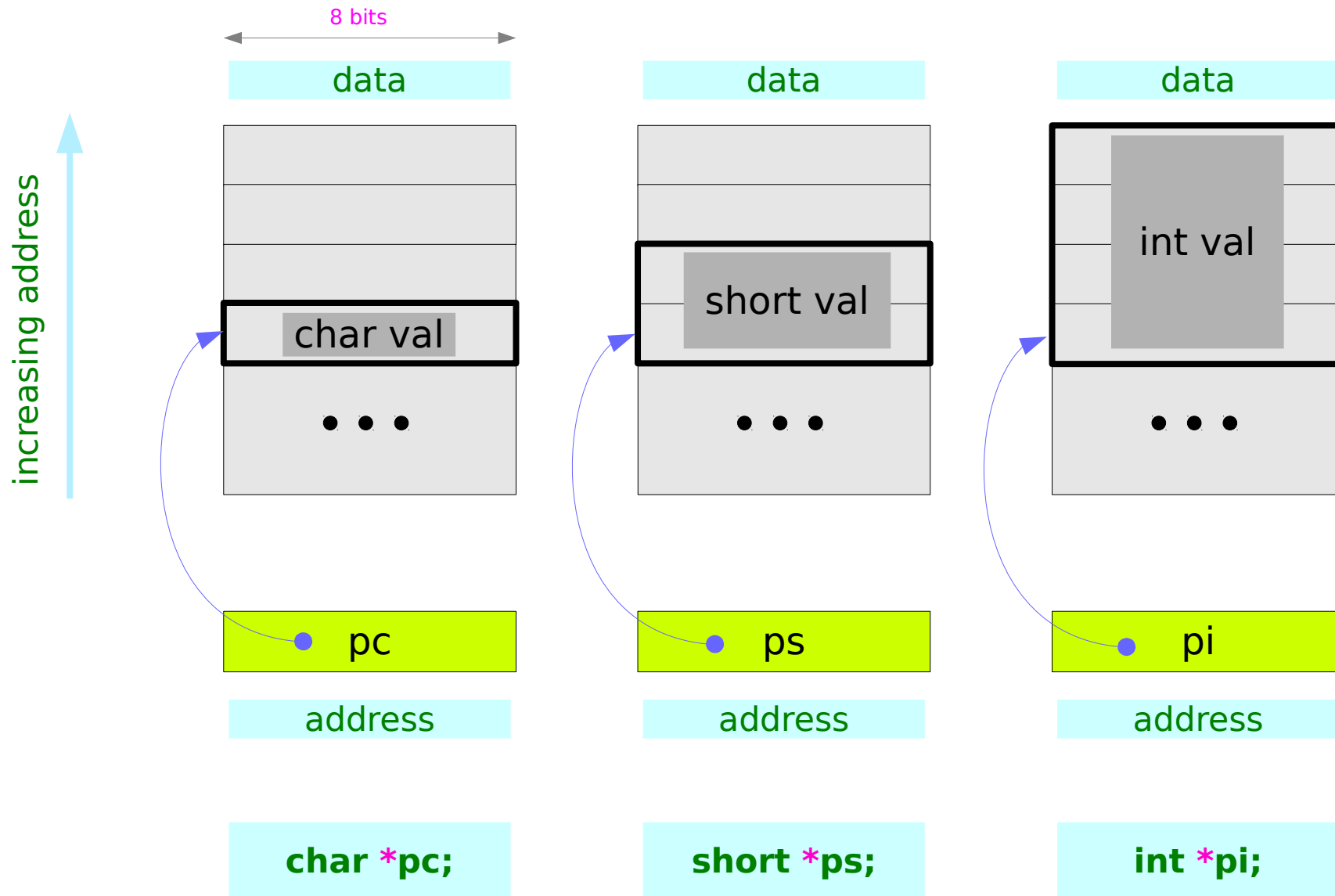


# const type, const pointer type (3)

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

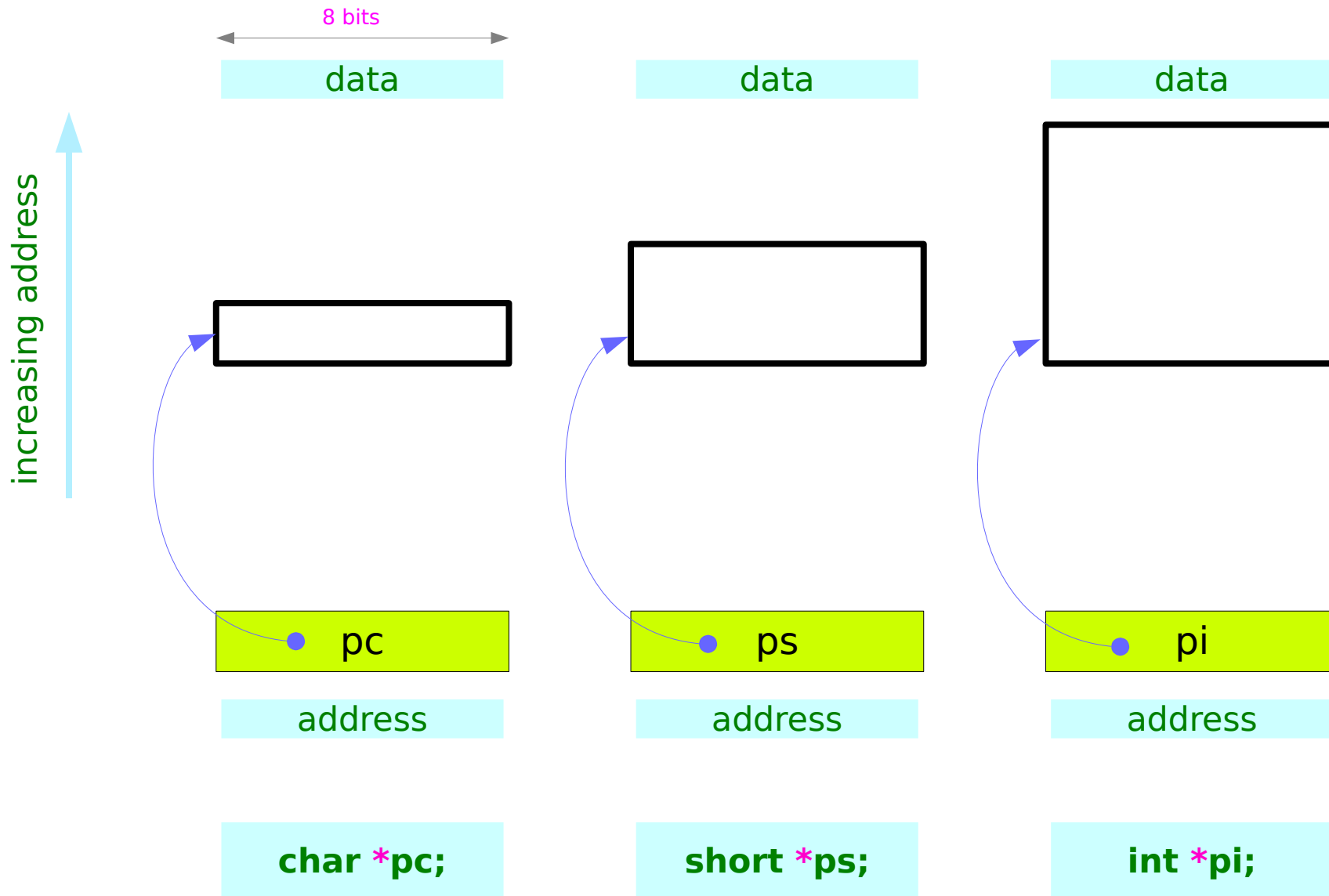


# Pointer Types and Associated Data

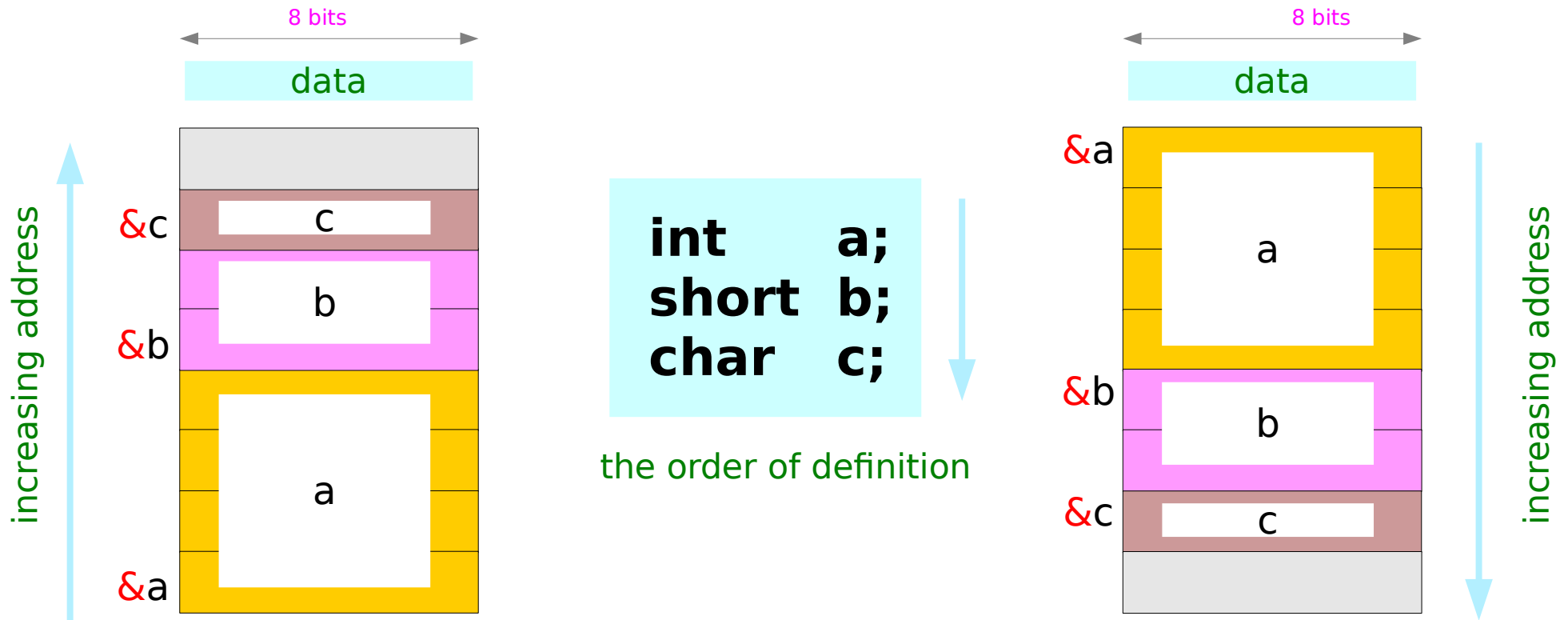




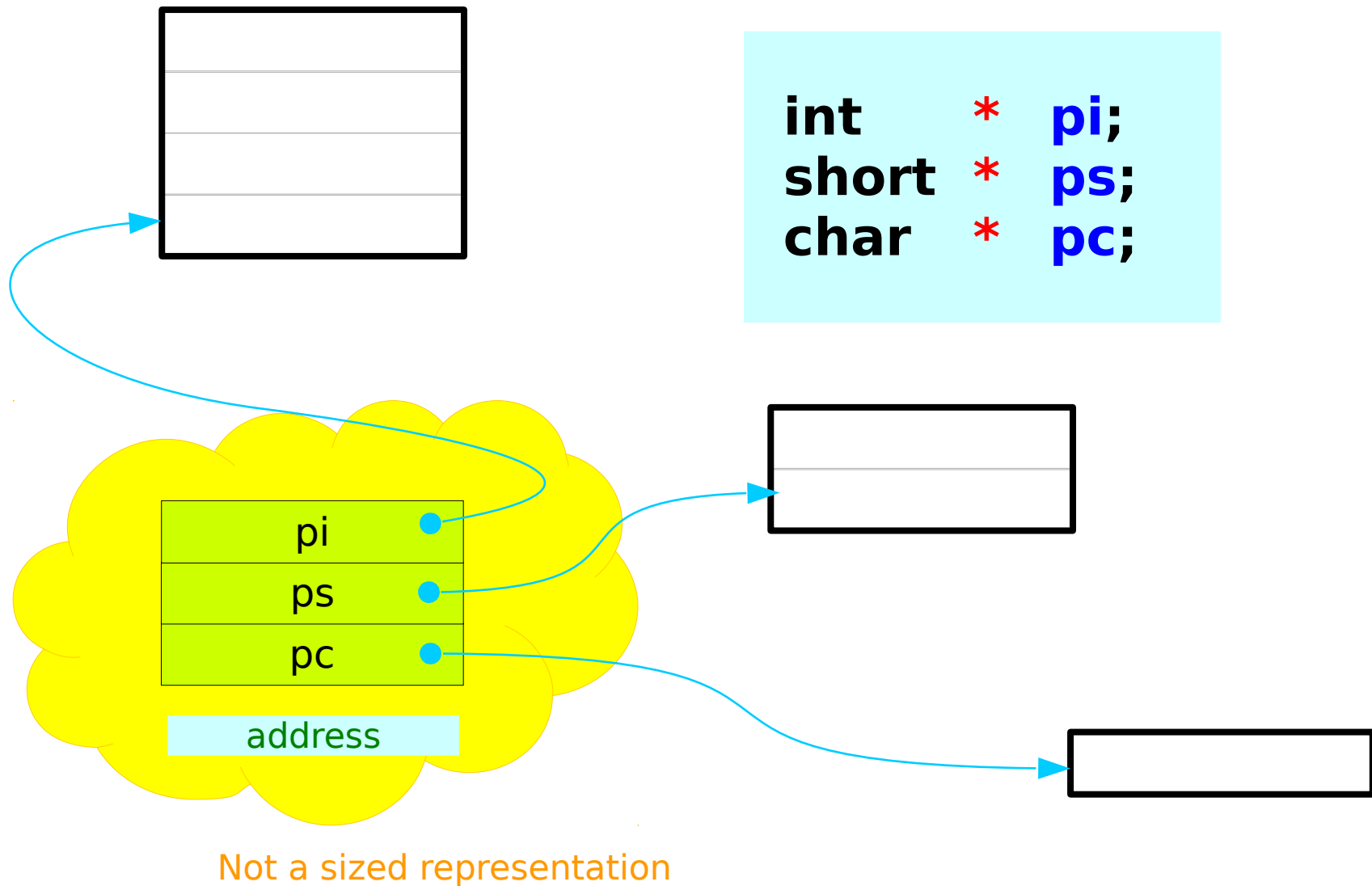
# Pointer Types



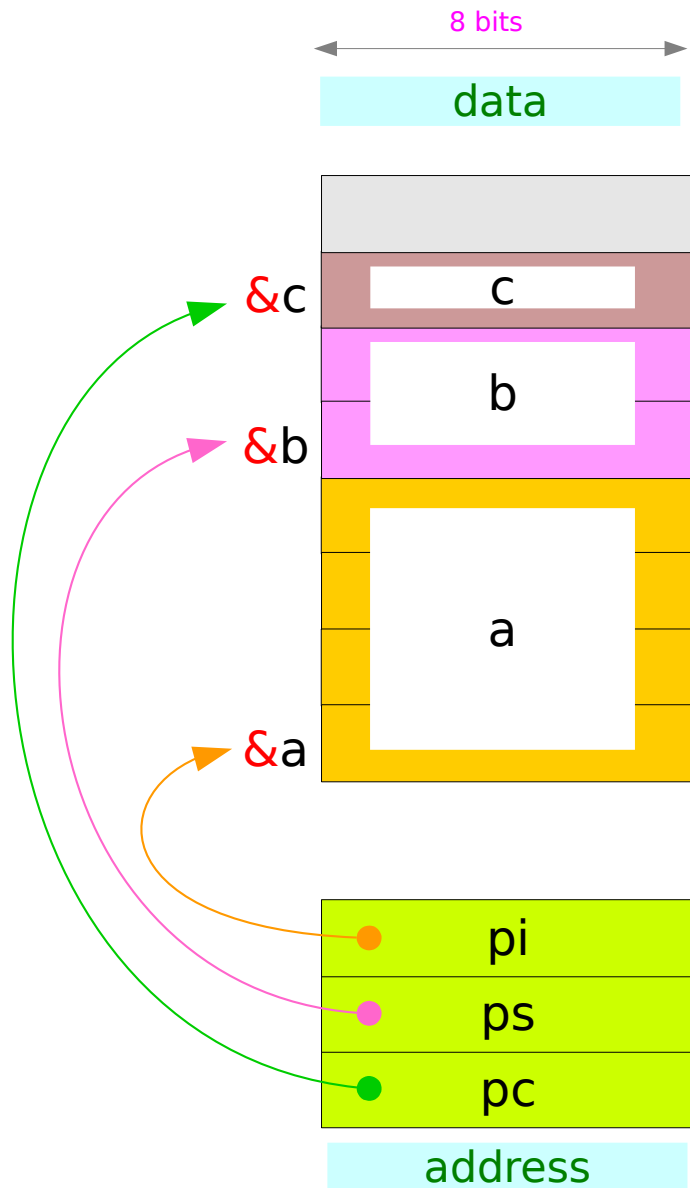
# Little Endian Example



# int \*, short \*, char \* type variables



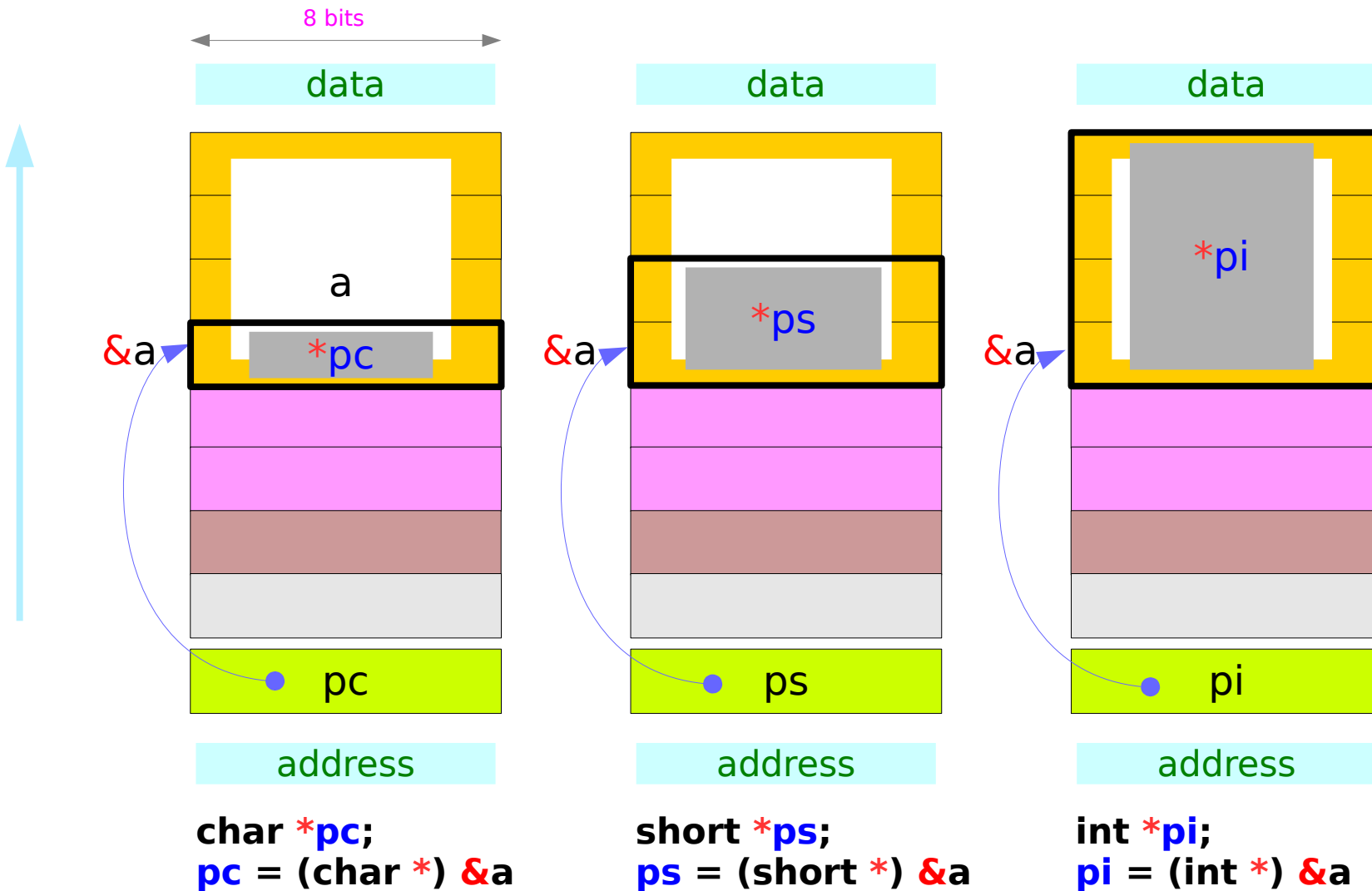
# Pointer Variable Assignment



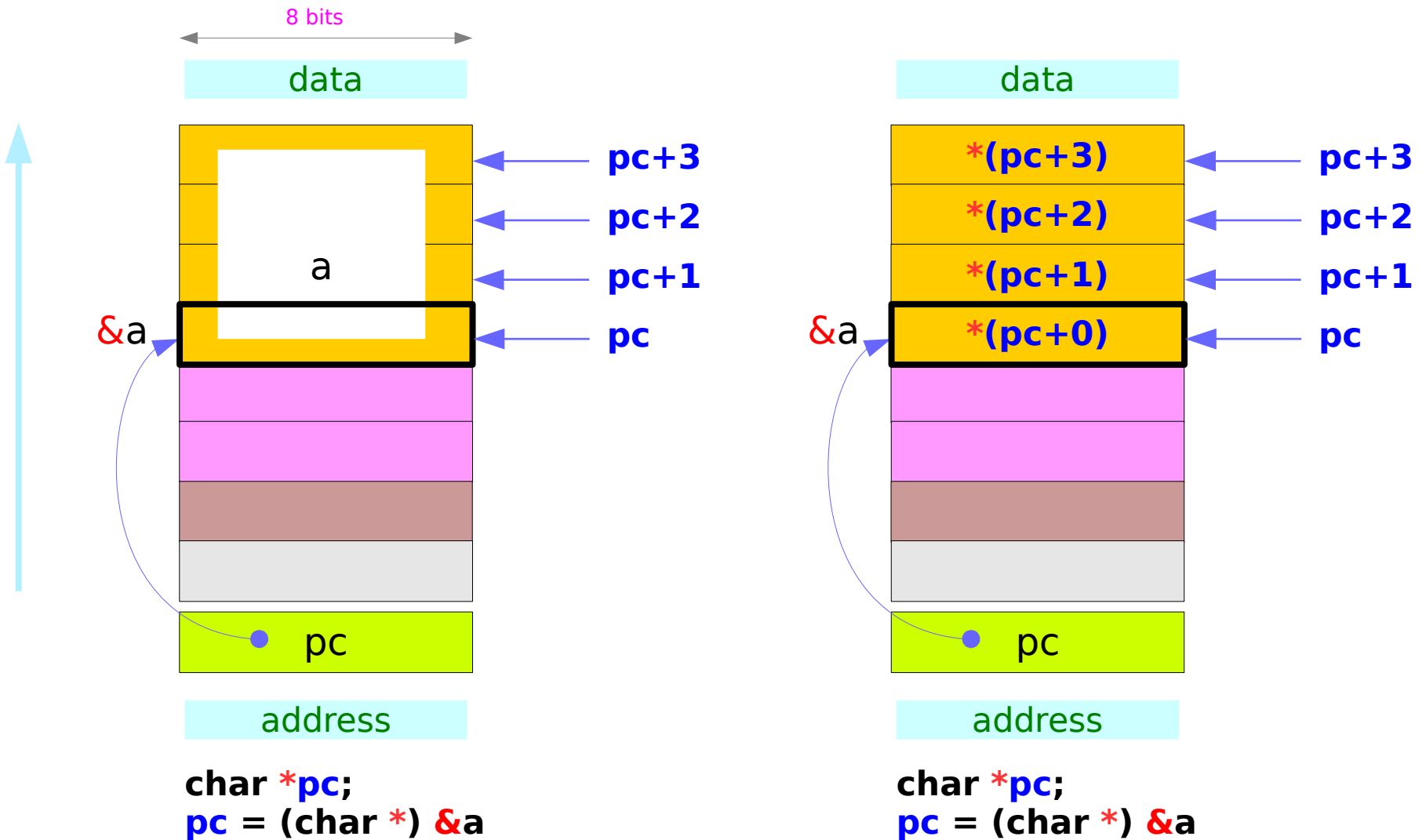
```
char * pc;  
short * ps;  
int * pi;  
  
int a;  
short b;  
char c;
```

```
pi = &a;  
ps = &b;  
pc = &c;
```

# Pointer Type Casting



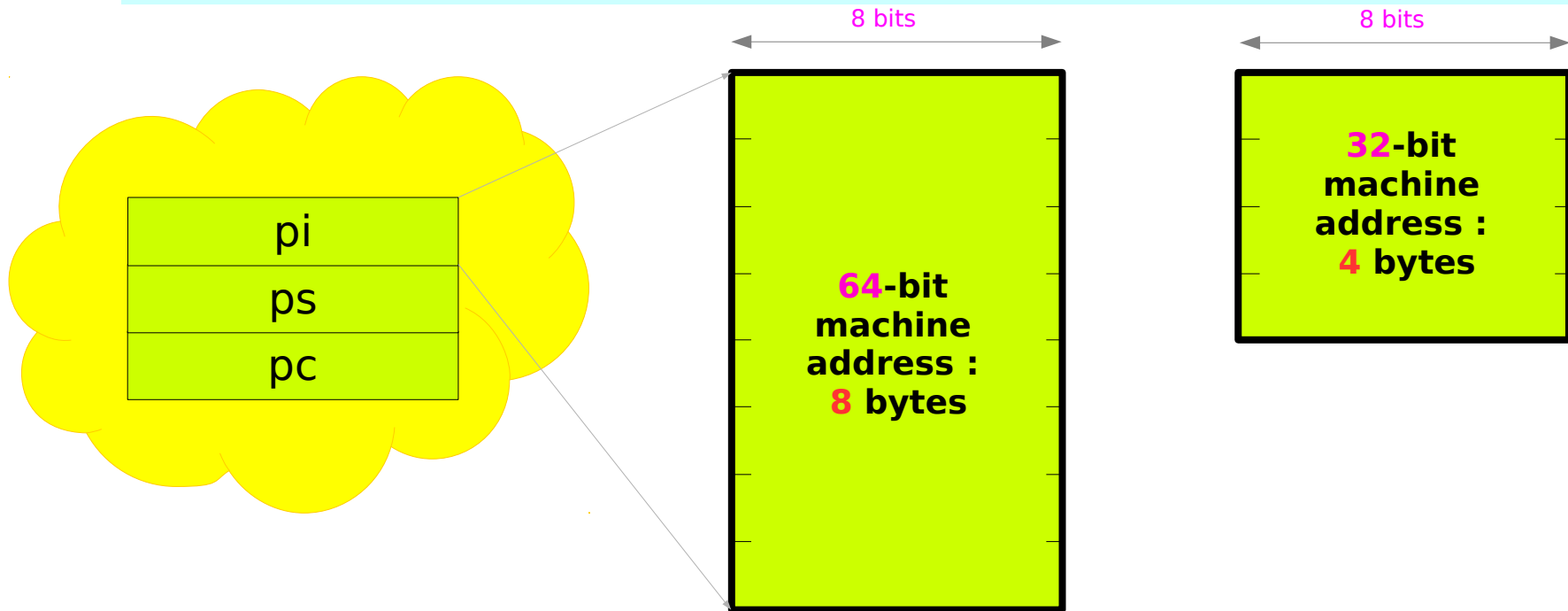
# Accessing bytes of a variable



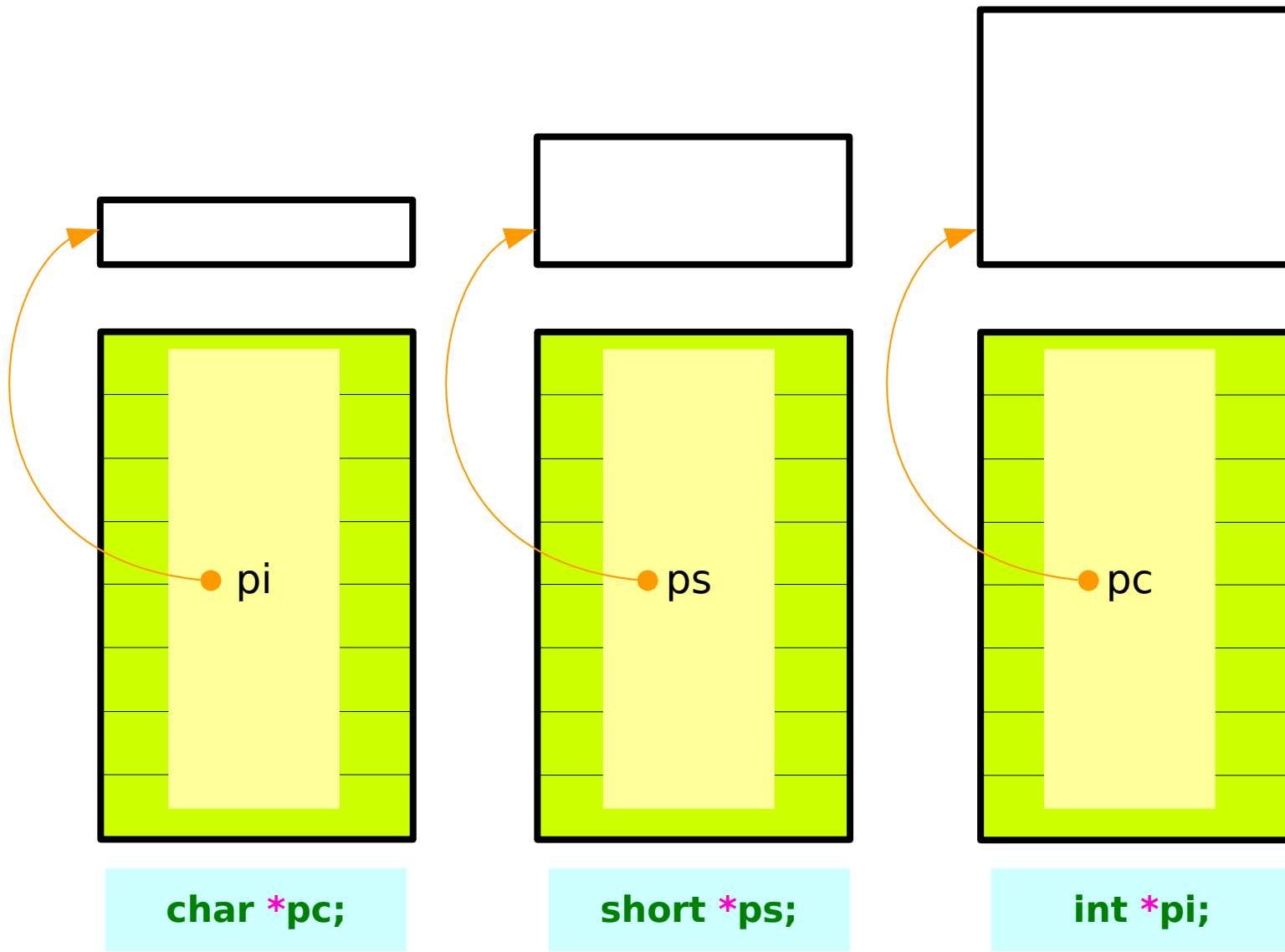
# 32-bit and 64-bit Address

**32-bit machine : address : 4 bytes**

**64-bit machine : address : 8 bytes**

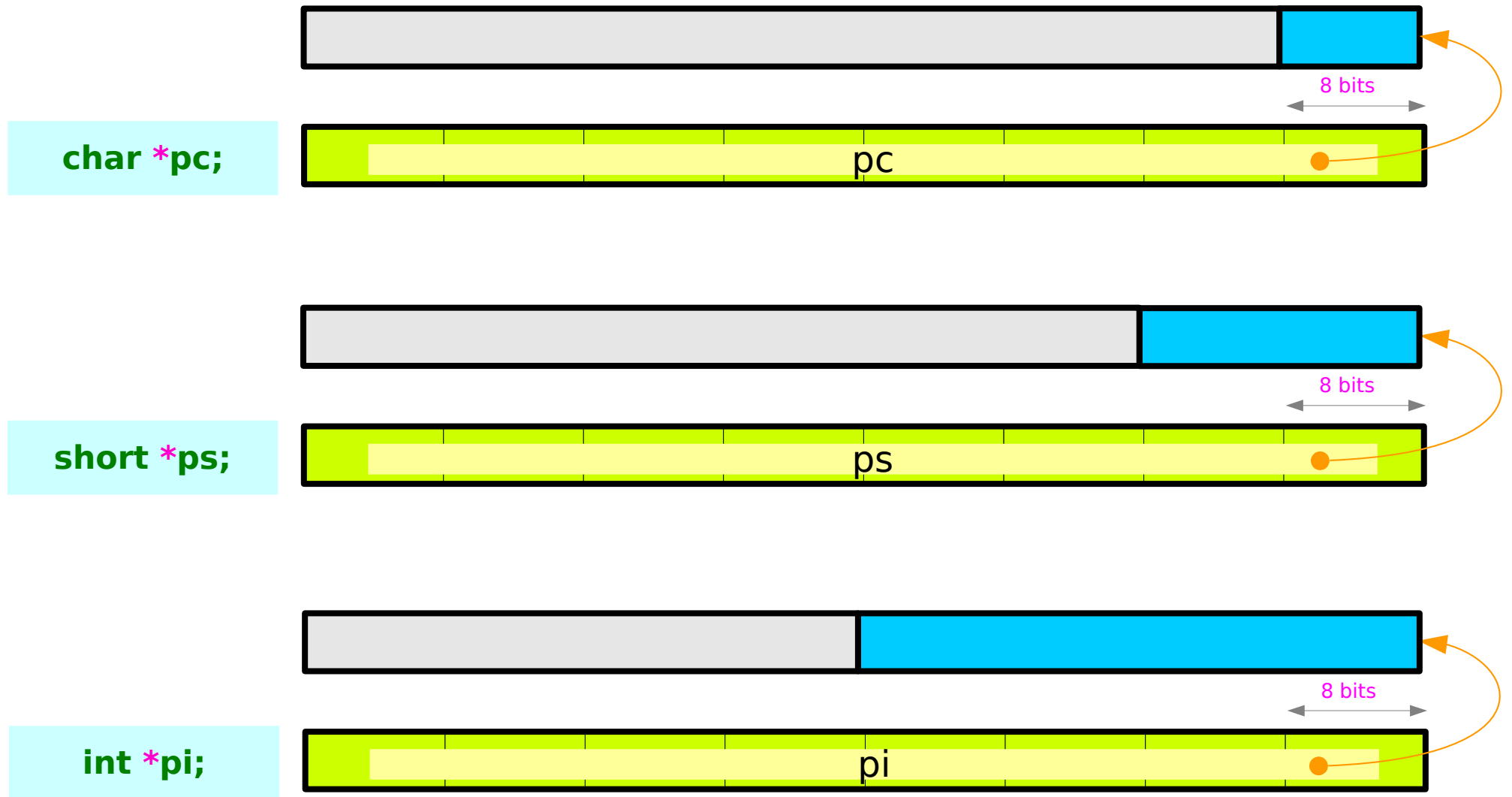


# 64-bit machine : 8-byte address

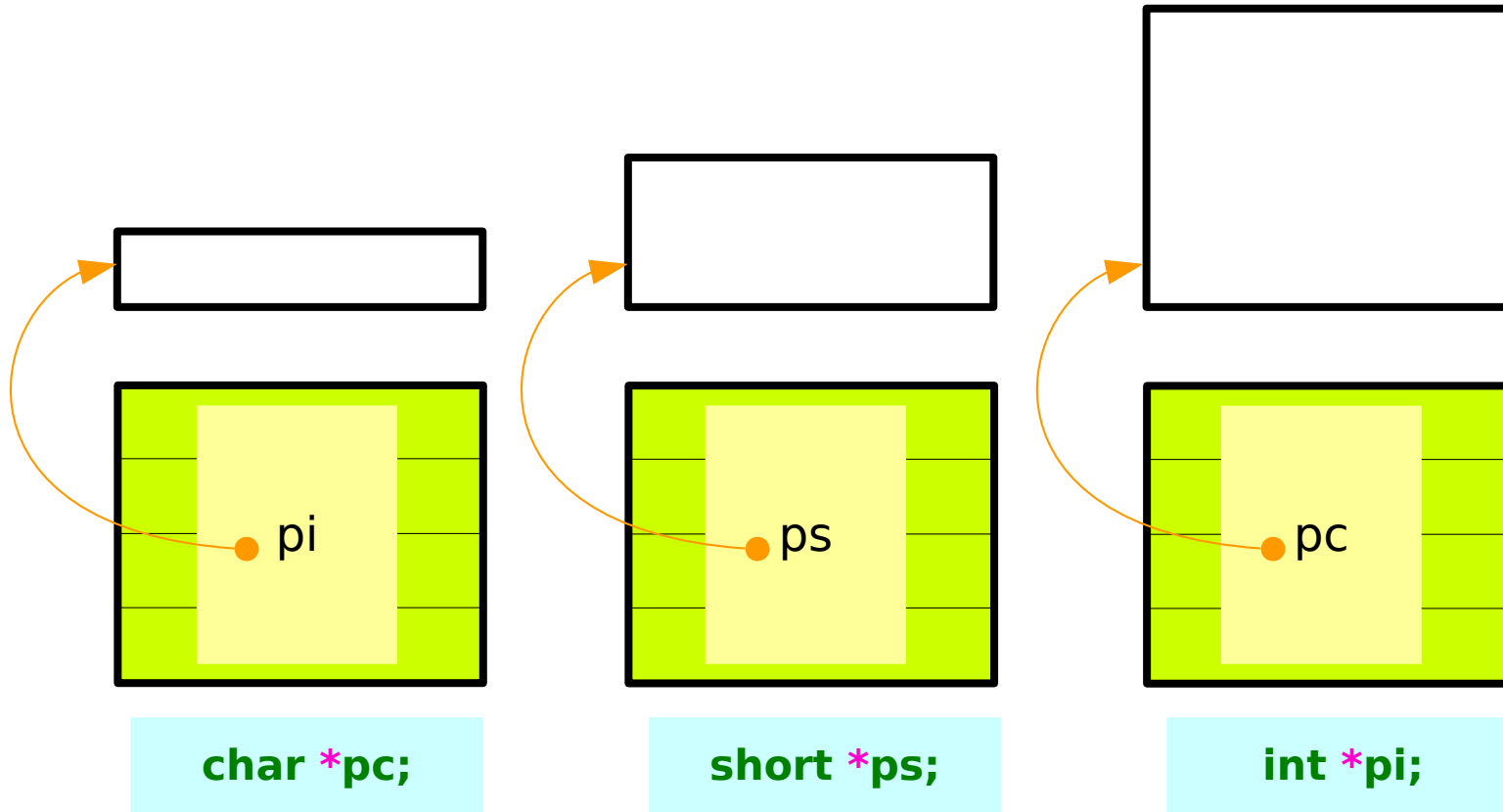




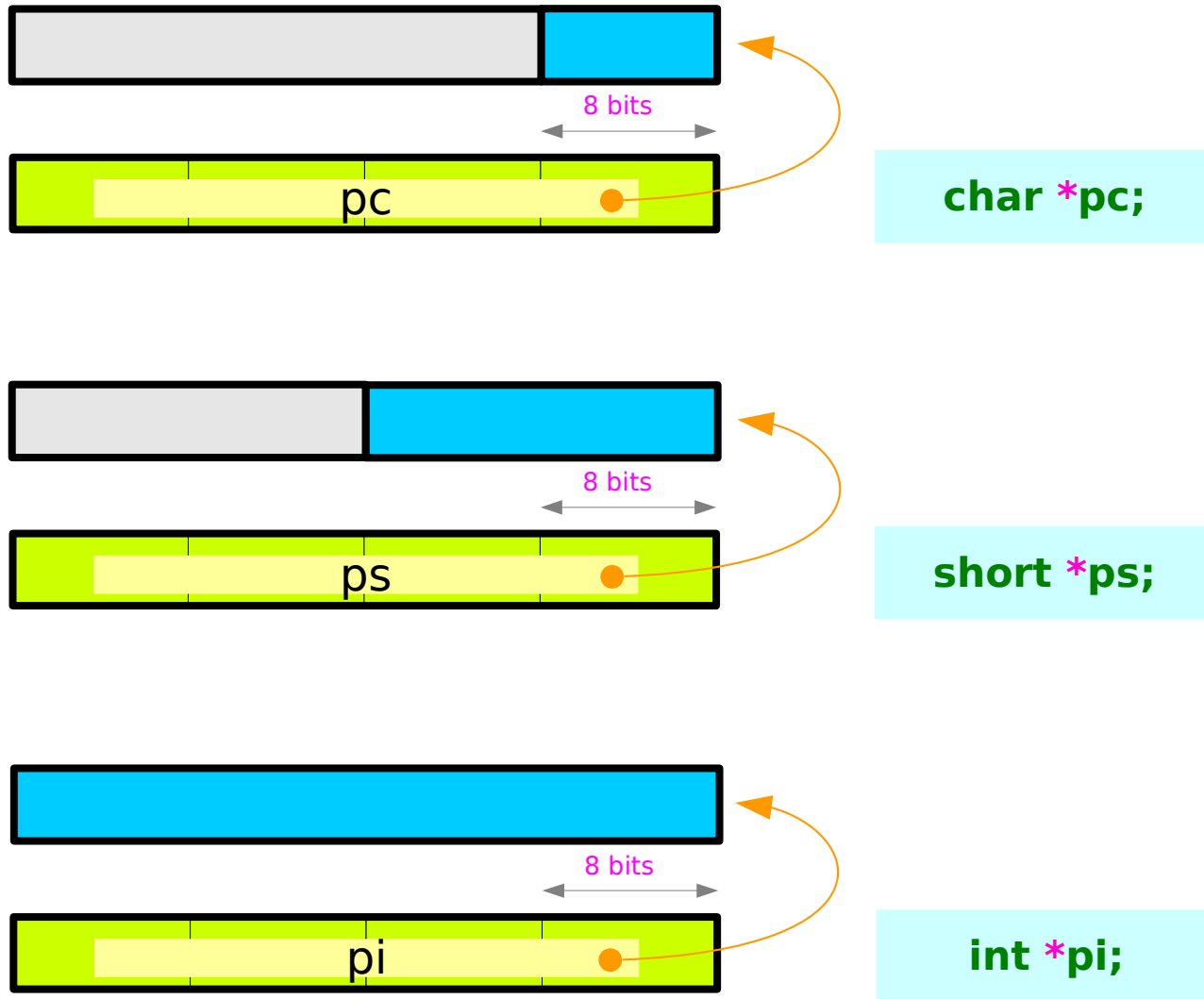
# 64-bit machine : 8-byte address & data buses



# 32-bit machine : 4-byte address



# 64-bit machine : 8-byte address and data buses



---

## 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