Pointers (1A)

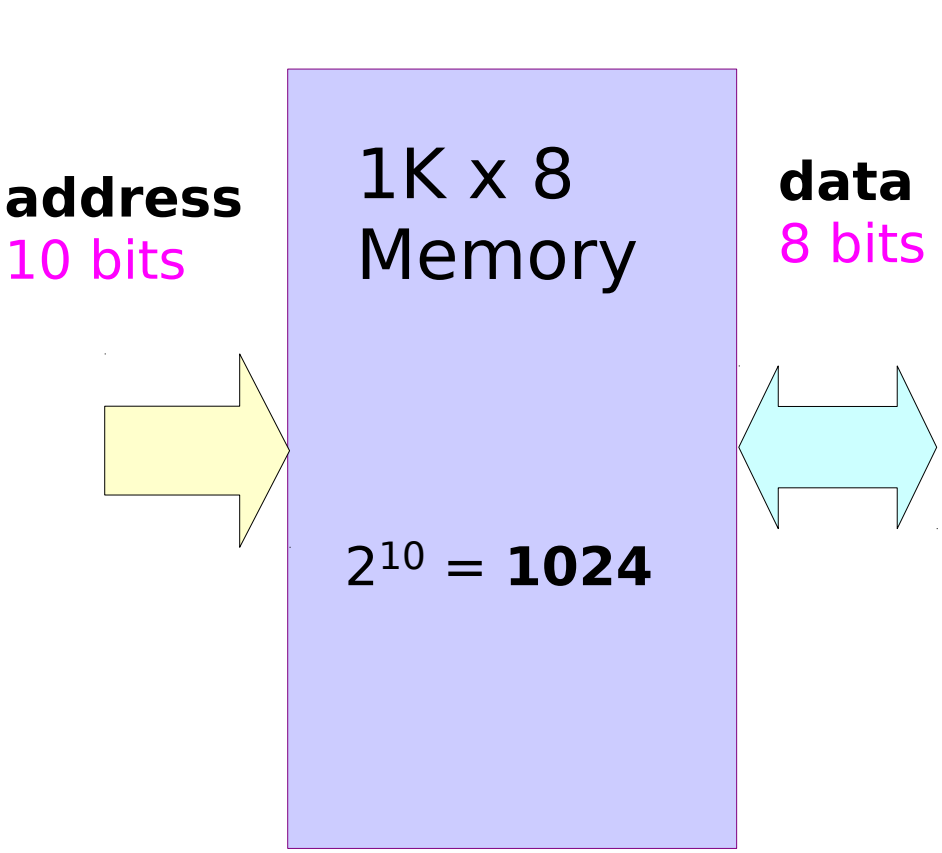
Young Won Lim
2/16/18

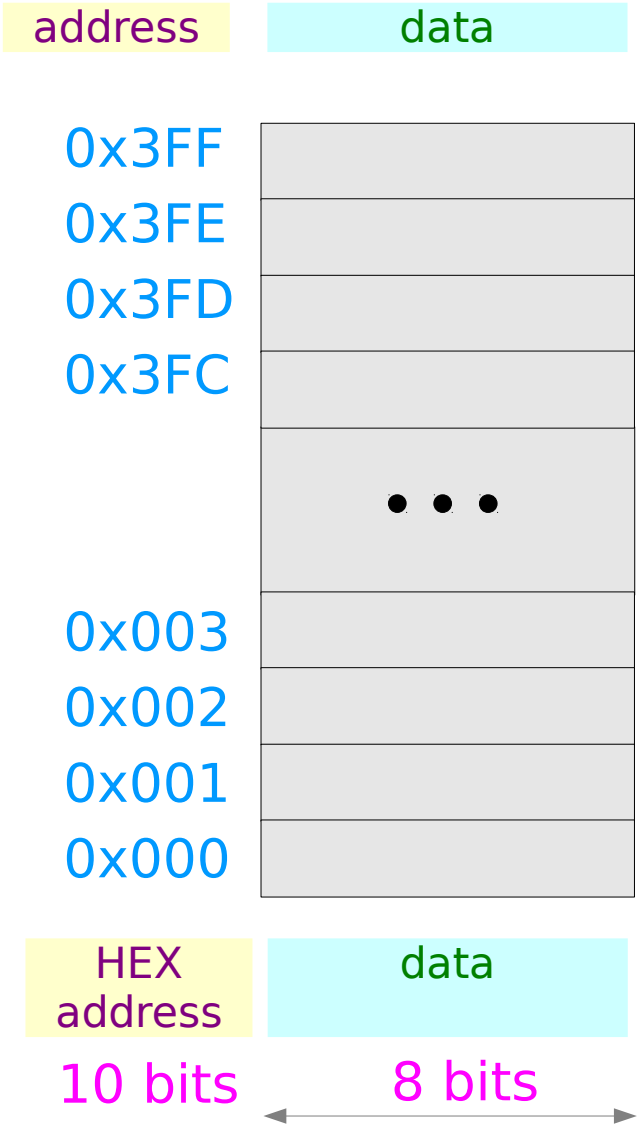
Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Young Won Lim
2/16/18

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

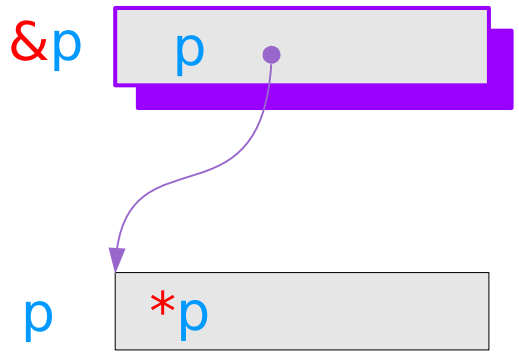
type ***variable***

```
int   *   p;
```

pointer to int

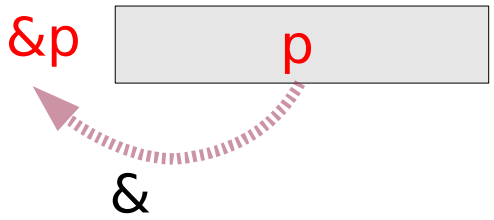
```
int   *   p;
```

int

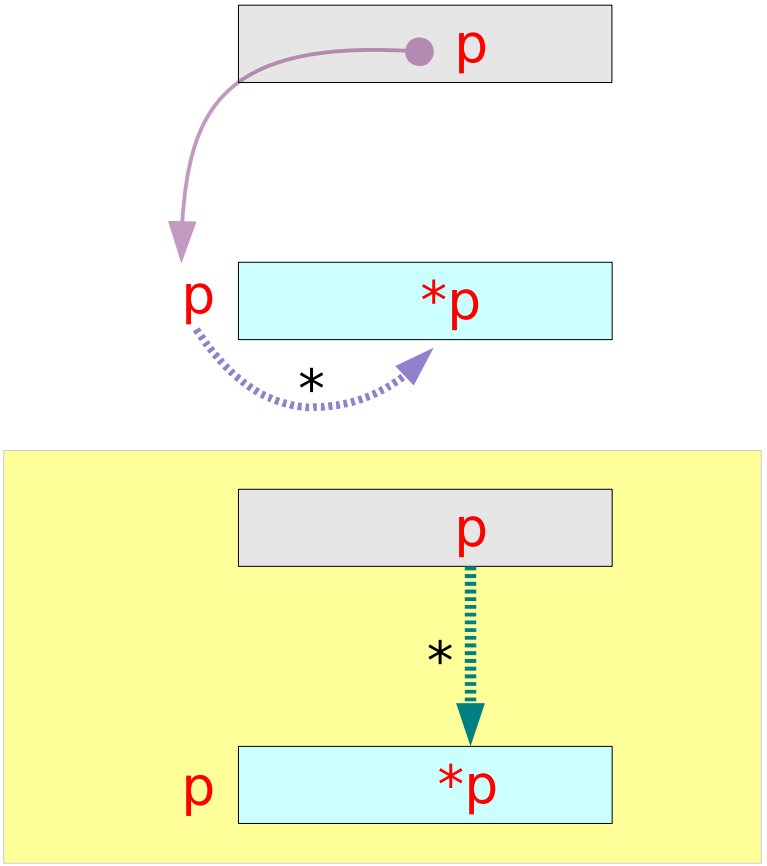


Dereferencing

The address of a variable :
Address of operator &

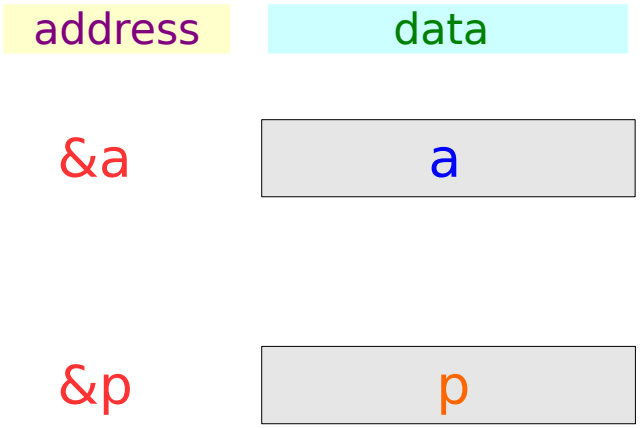


The content of a pointed location :
*Dereferencing operator **

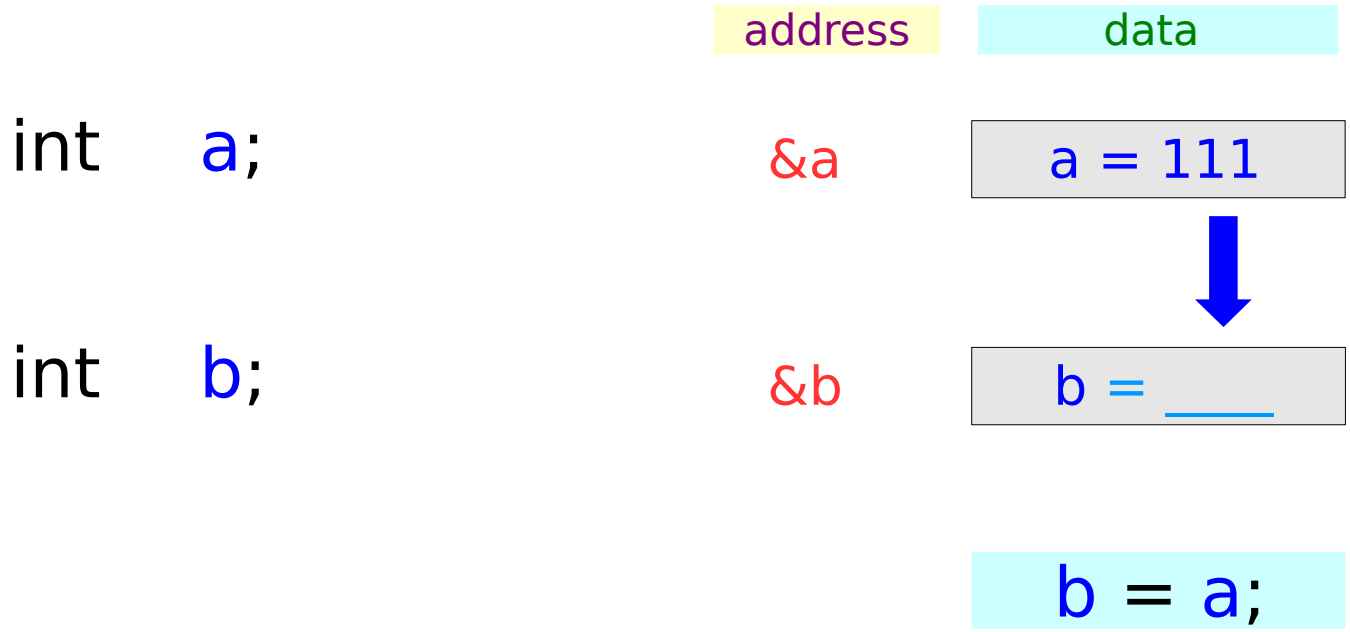


Variables and their addresses

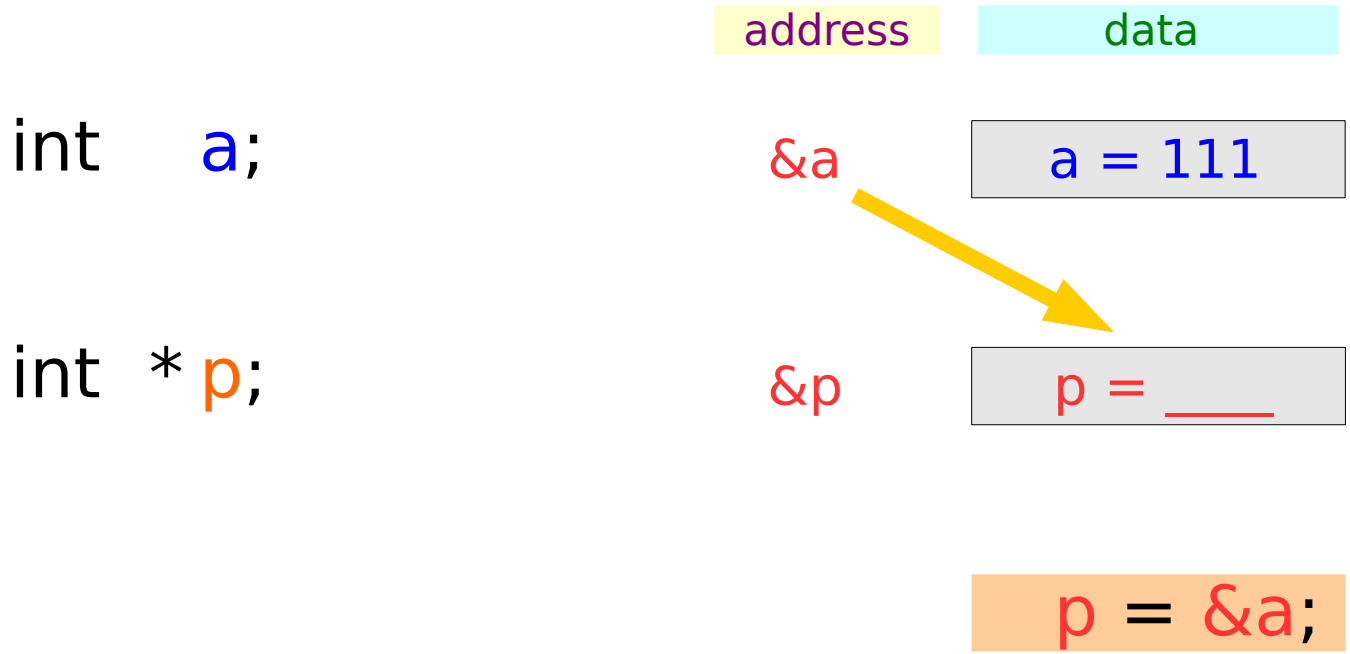
```
int    a;
int  * p;
```



Assignment of a value

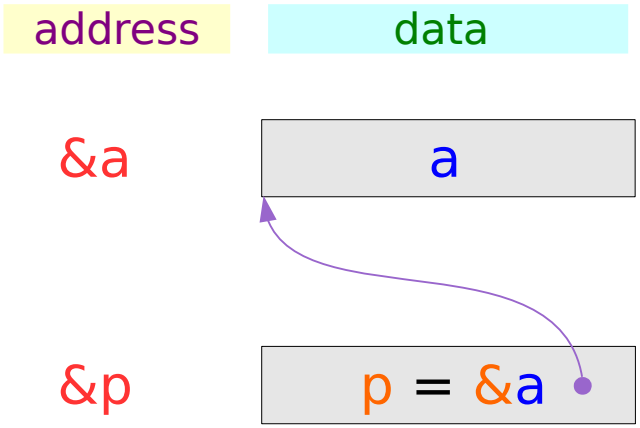


Assignment of an address



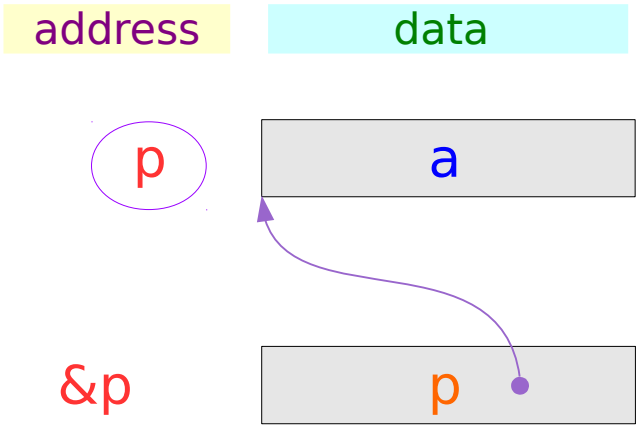
Variables with initializations

```
int    a;

int  * p = &a;
```



Pointed addresses : p

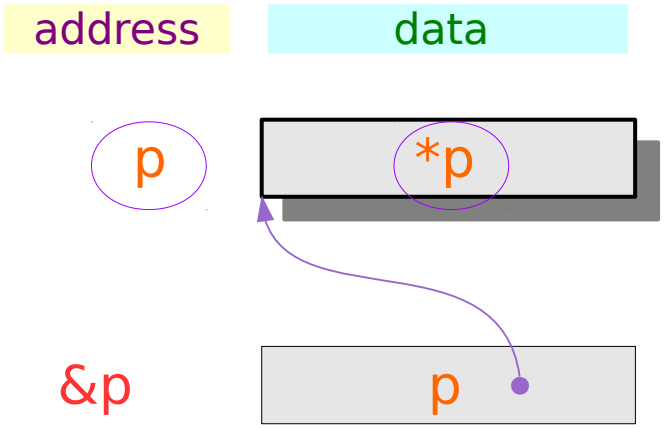
```
int    a;

int  * p = &a;
```



p ≡ &a

Dereferenced Variable : *p

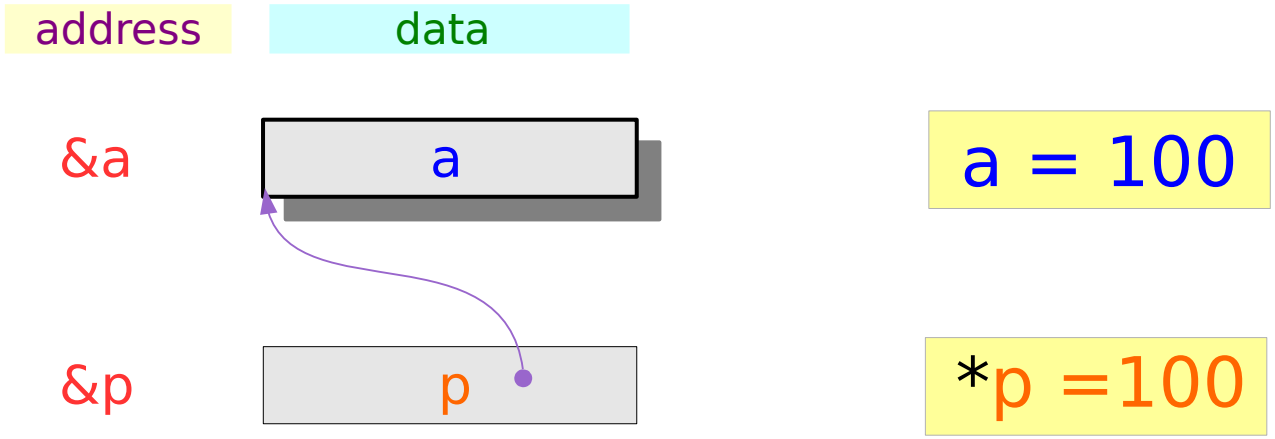
```
int    a;

int  * p = &a;
```



p ≡ &a

*p ≡ 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;
    ...
    ...


    ...
    ...
}
```

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

    func1 ( 10, 20 );

    ...
    ...
}
```

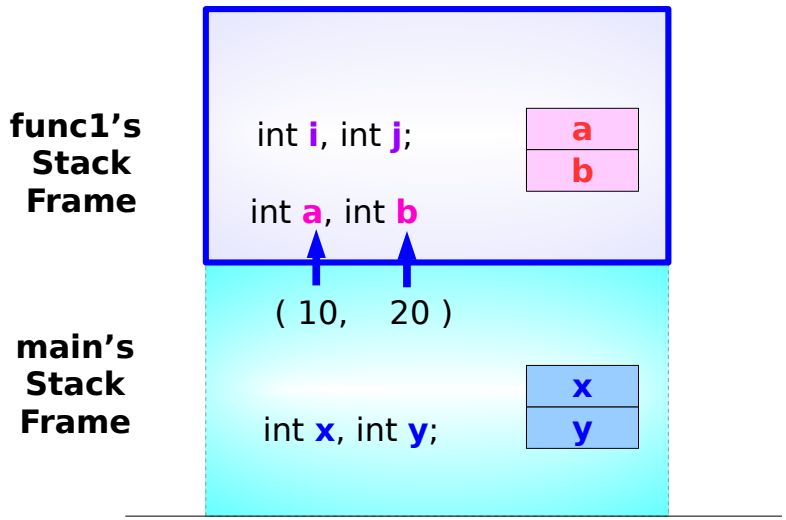
i and j's
variable scope

cannot access
each other

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;
    ...
    ...


    ...
    ...
}
```

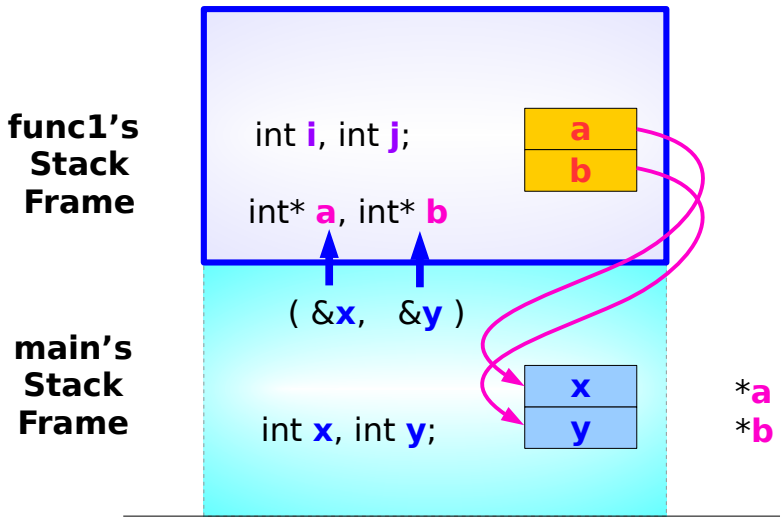
```
int main ()
{
    int x, int y;
    ...
    ...

    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



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 *	p
int	*q

int *	p
int	*q

int	tmp

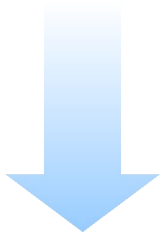
```
int a, b;
    ...
swap( &a, &b );
```

Integer and Integer Pointer Types

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

integer pointer declarations

a way of thinking



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

integer pointer variables — int *

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

treated as integer variables — int

variables ***types***

Arrays

Accessing array elements – using an address

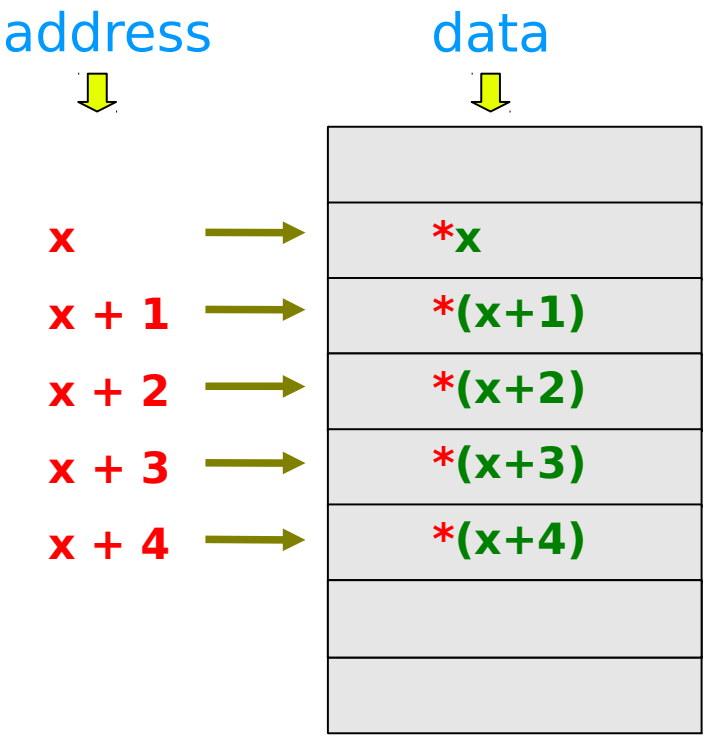
```
int     x[5];
```

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

5 int variables

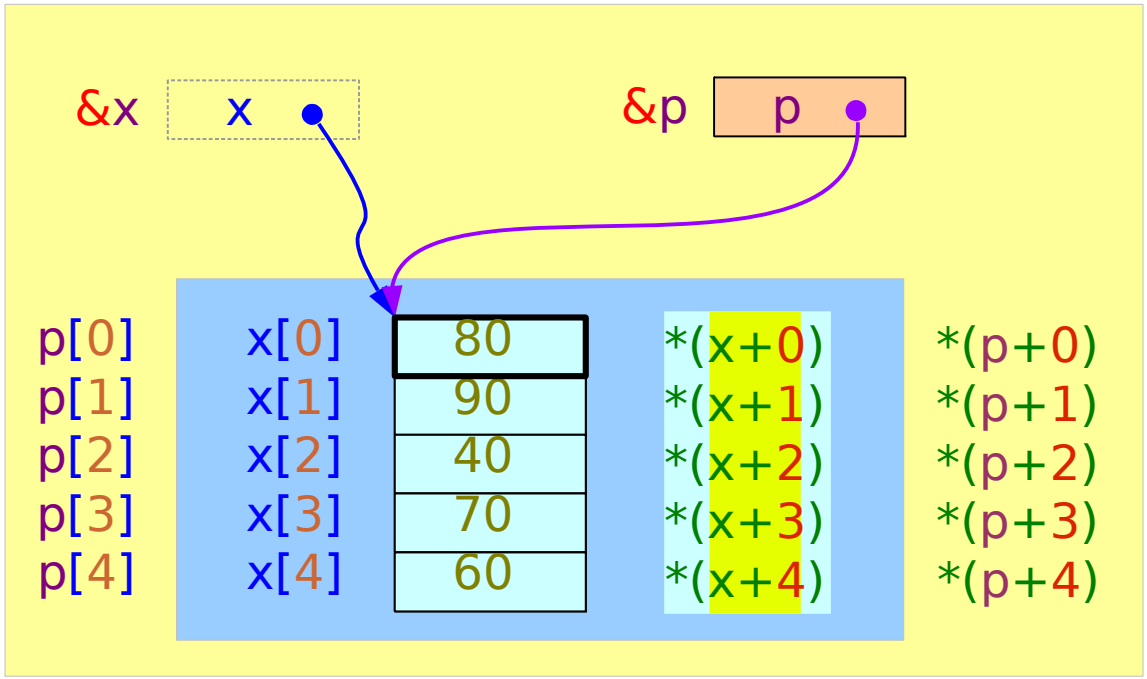
index	data
0	x[0]
1	x[1]
2	x[2]
3	x[3]
4	x[4]

cannot change
address x
(constant)



Accessing an Array with a Pointer Variable

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

int  *p = x;
```



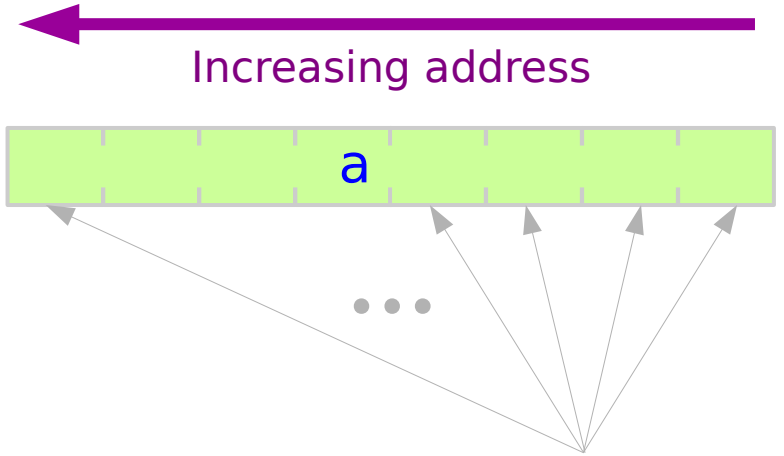
x is a constant symbol
cannot be changed

p is a variable
can point to other addresses

Byte Address
Little Endian
Big Endian

Byte Address

long a;



8-byte size data type

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

$$a_7 = 0x10 \cdots 16^7$$ the highest weight

$$a_6 = 0x20 \cdots 16^6$$

$$a_5 = 0x30 \cdots 16^5$$

$$a_4 = 0x40 \cdots 16^4$$

$$a_3 = 0x50 \cdots 16^3$$

$$a_2 = 0x60 \cdots 16^2$$

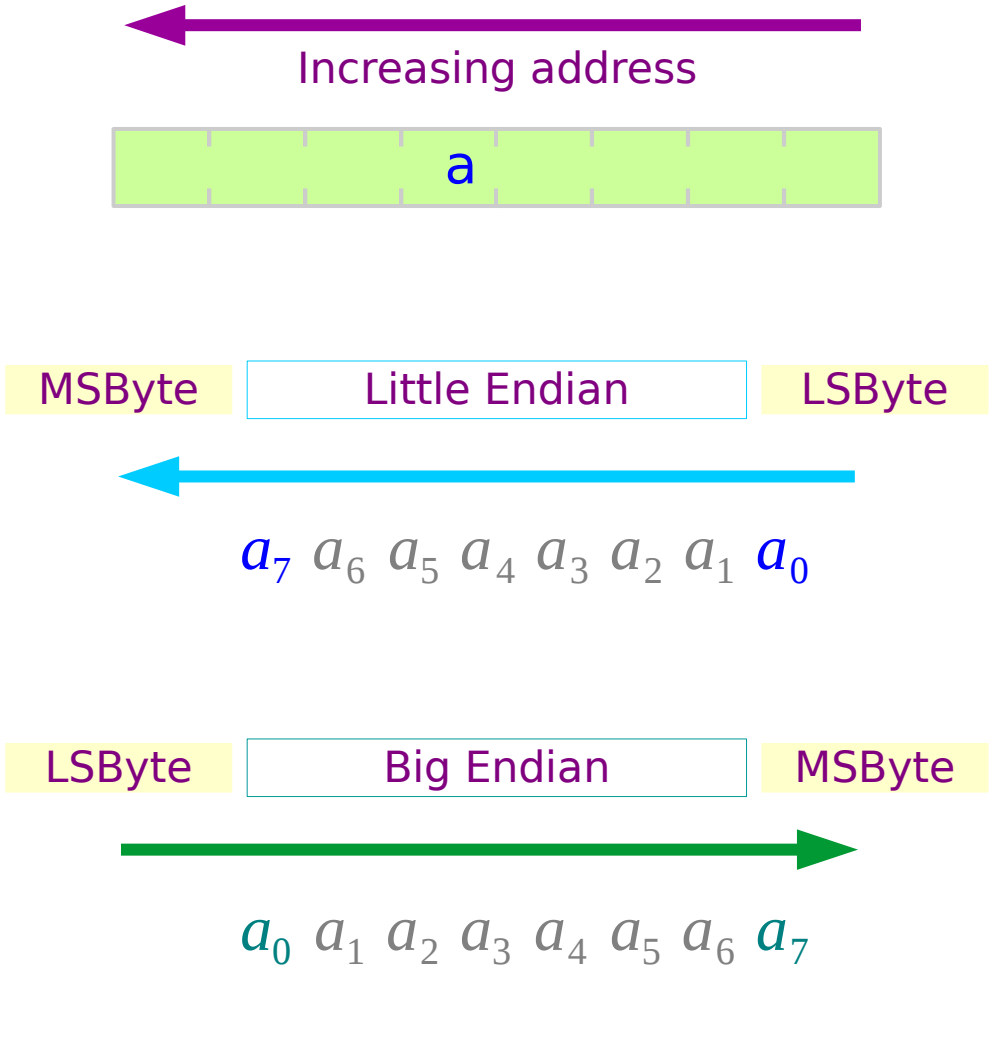
$$a_1 = 0x70 \cdots 16^1$$

Least Significant Byte

$$a_0 = 0x80 \cdots 16^0$$ the lowest weight

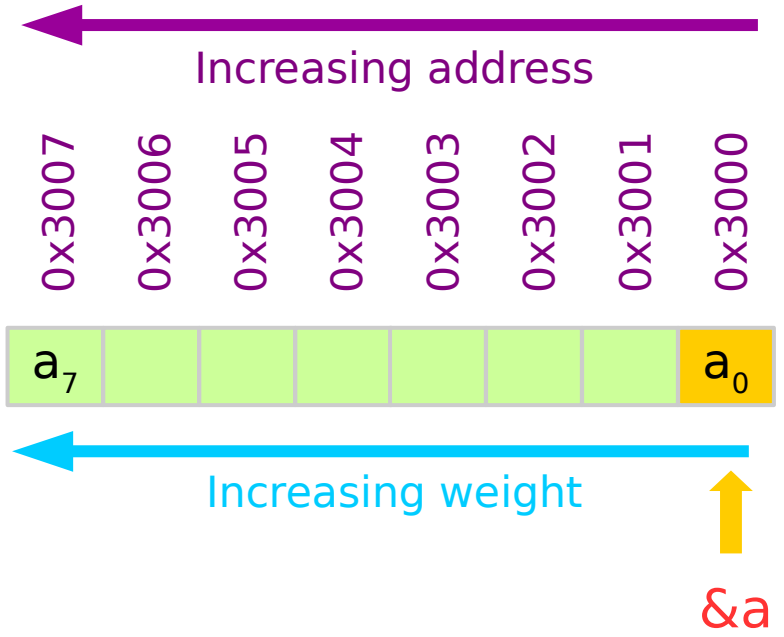
Little / Big Endian Ordering of Bytes

long a;



Little Endian Byte Address Example

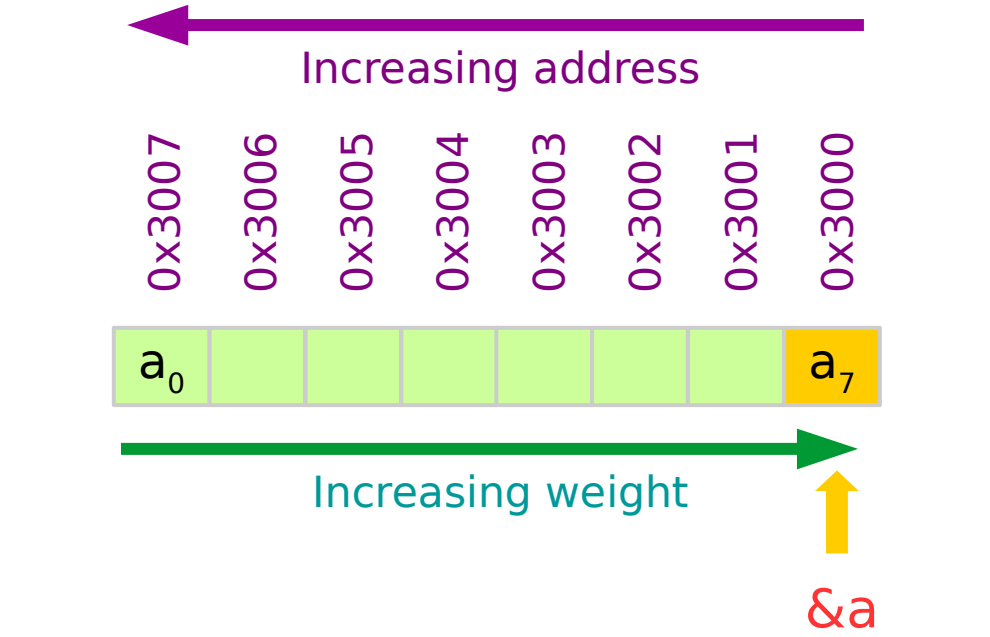
long a;



MSByte | **Little** Endian | LSByte

Big Endian Byte Address Example

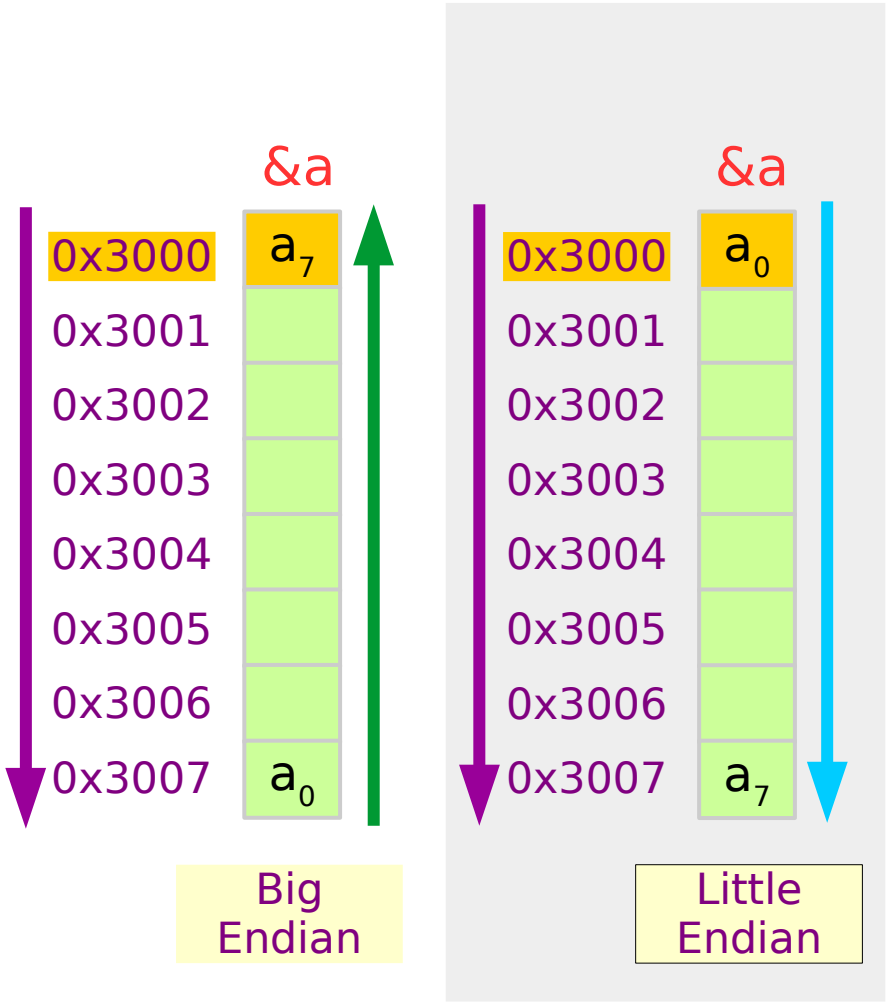
long a;



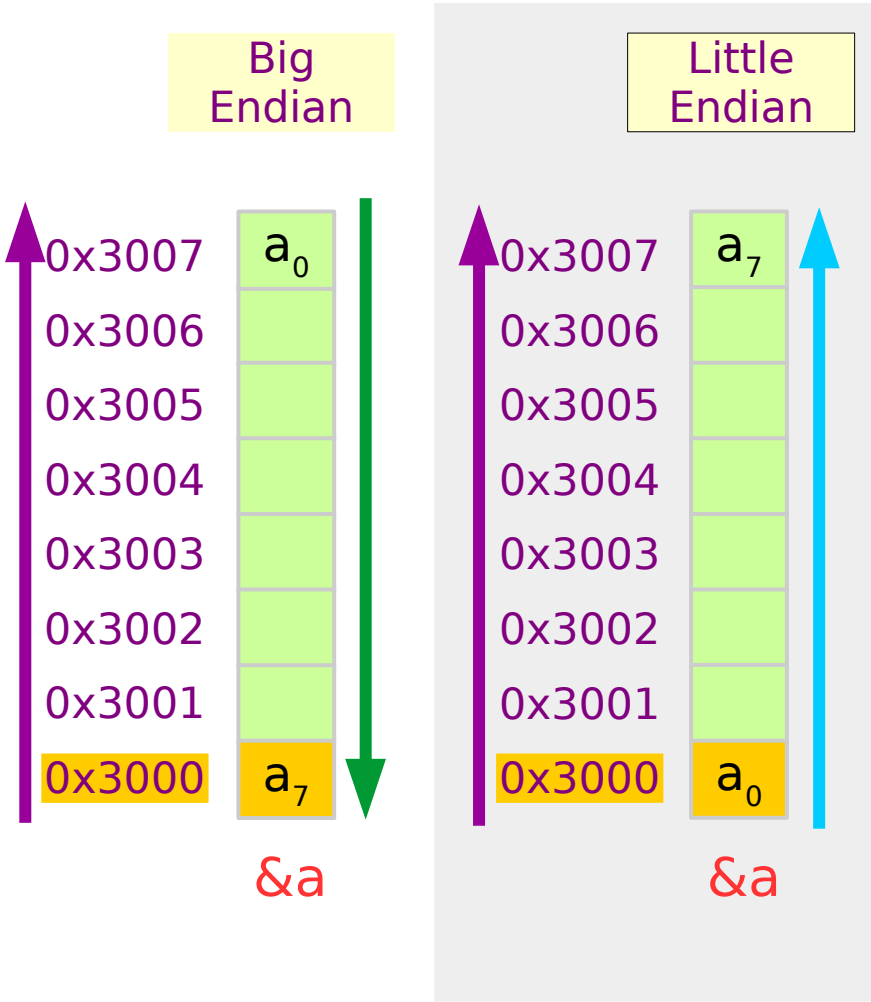
LSByte | **Big** Endian | MSByte

Representations of Endianness

downward, increasing address



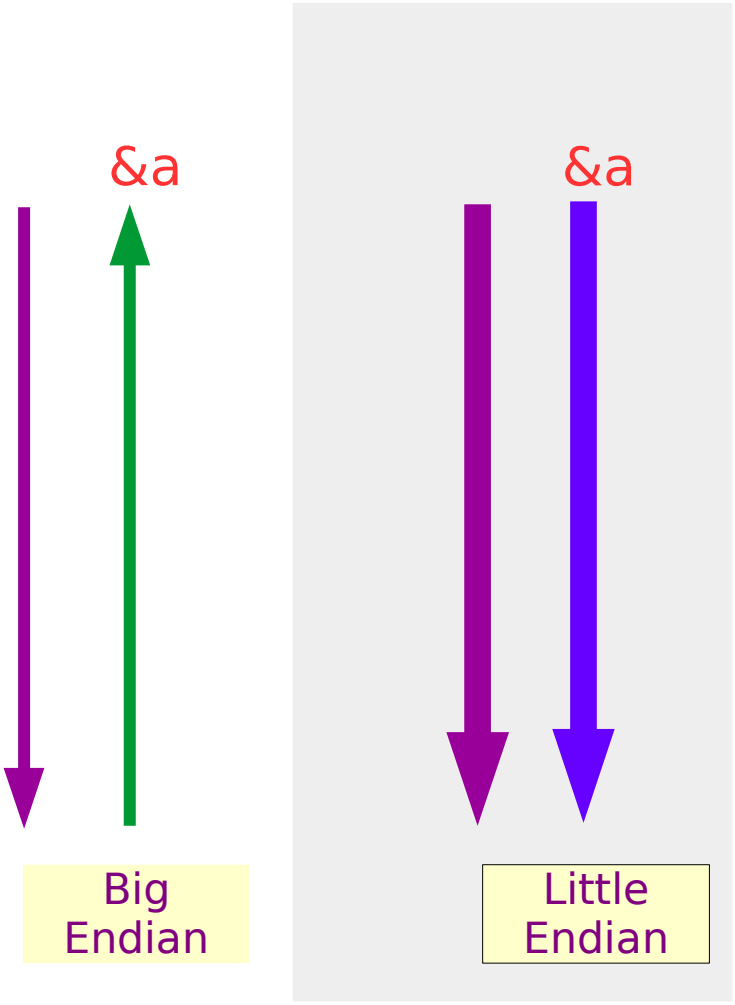
upward, increasing address



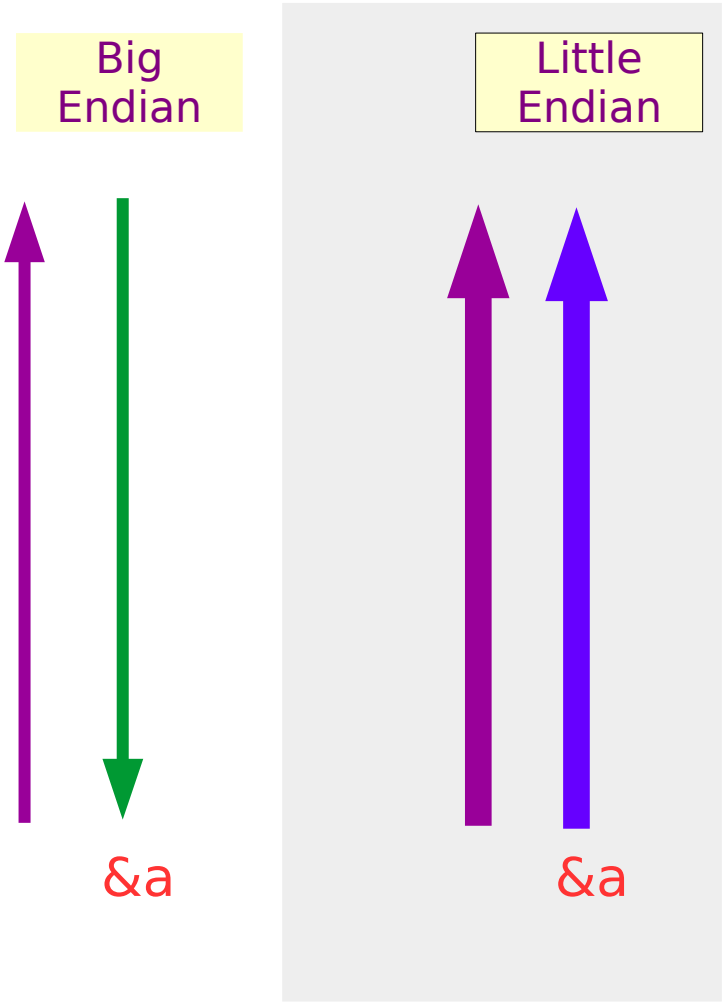
https://stackoverflow.com/questions/15620673/which-bit-is-the-address-of-an-integer

Increasing address, Increasing byte weight

downward, increasing address



upward, increasing address



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

Little / Big Endian Processors

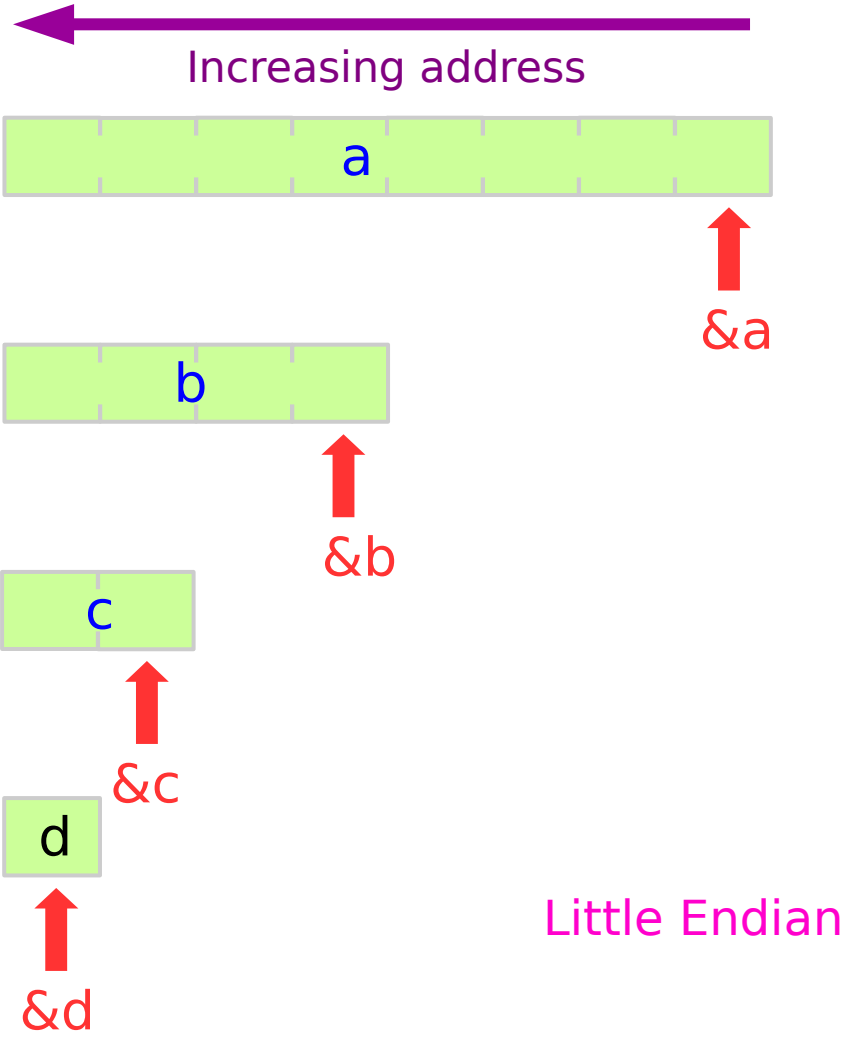
Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	(Big/Little) Endian
ARM	(Big/Little) Endian
IA-64 (64 bit)	(Big/Little) Endian
MIPS	(Big/Little)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

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

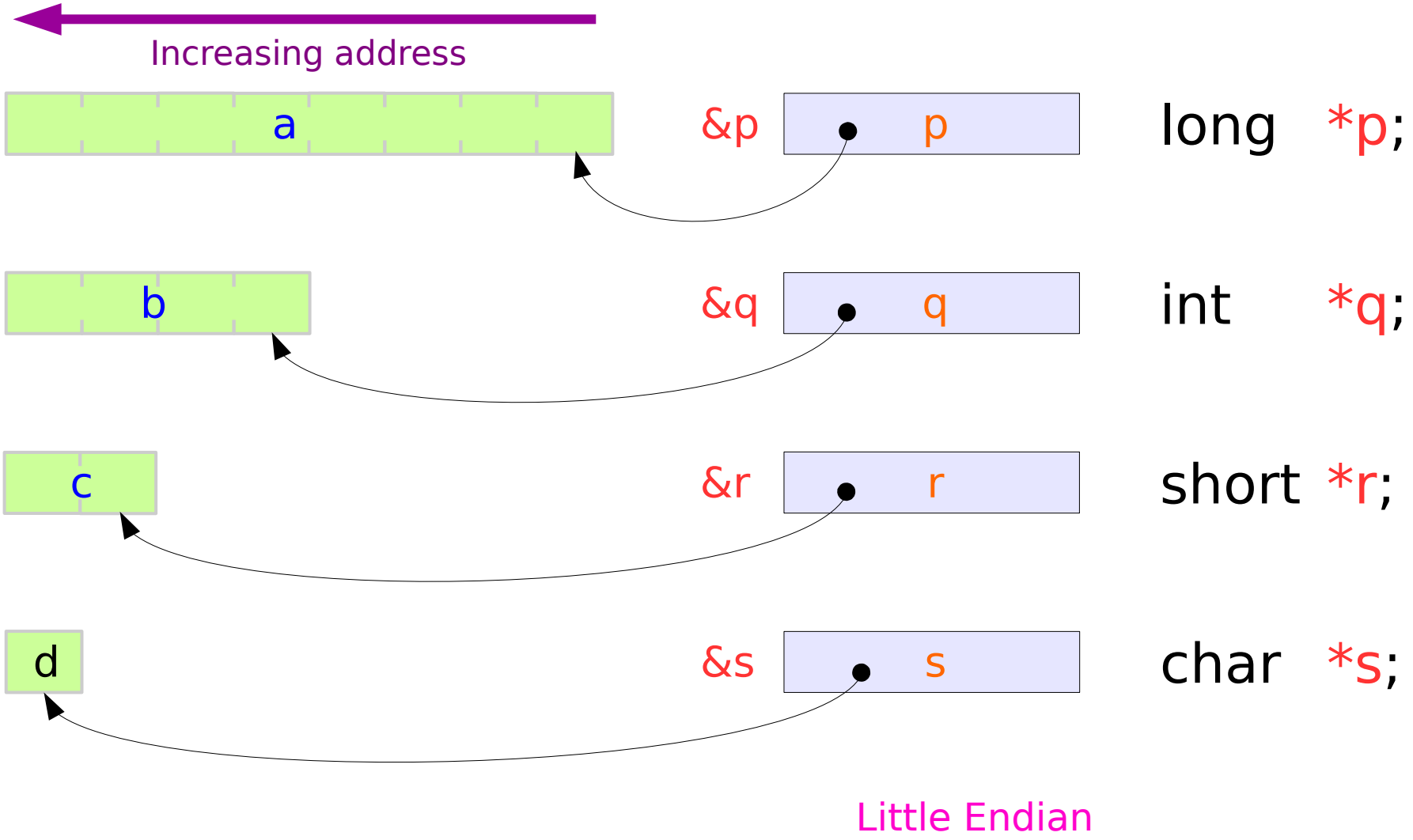
Pointer Types

Integer Type Variables and Their Addresses

```
long  a;
int   b;
short c;
char  d;
```



Points to the LSByte



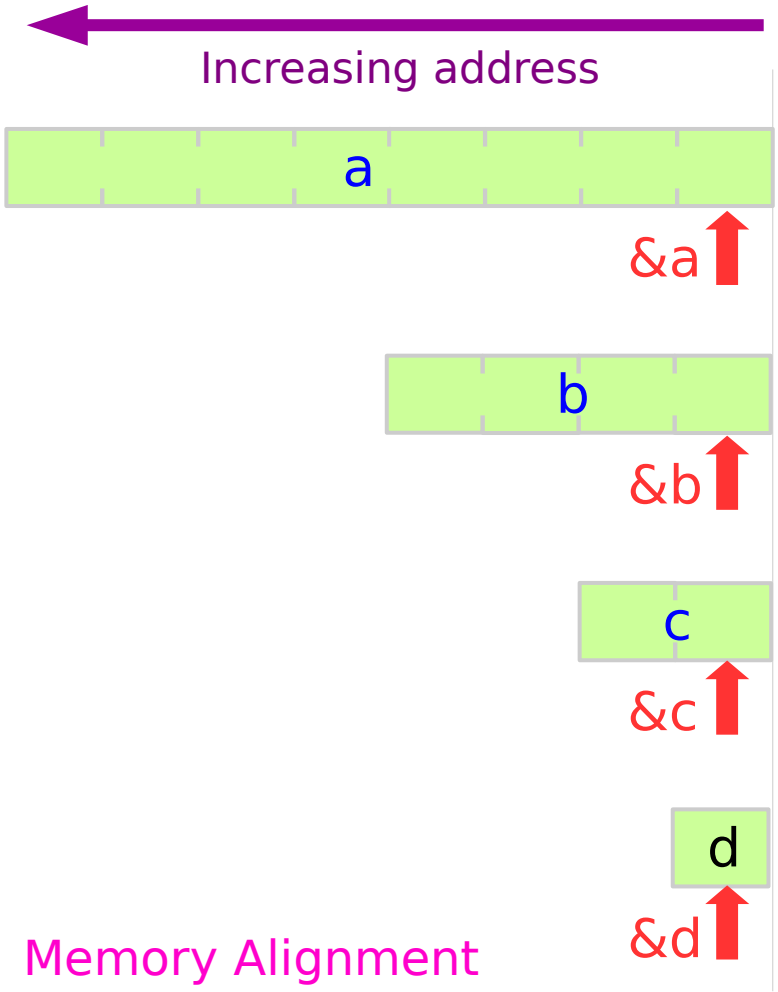
Aligning variables of different sizes

long a; sizeof(long) ➡ 8 (bytes)

int b; sizeof(int) ➡ 4 (bytes)

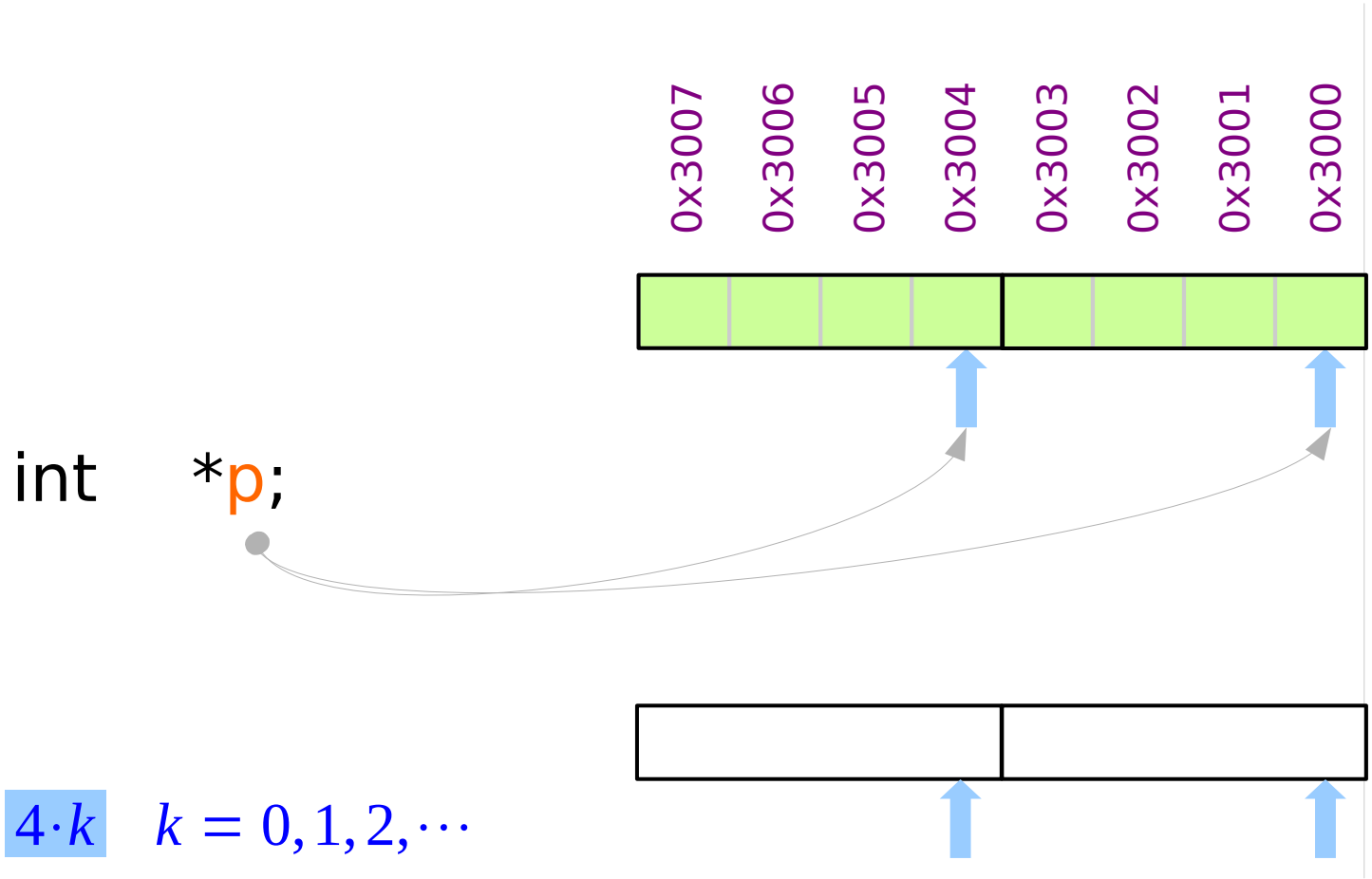
short c; sizeof(short) ➡ 2 (bytes)

char d; sizeof(char) ➡ 1 (bytes)

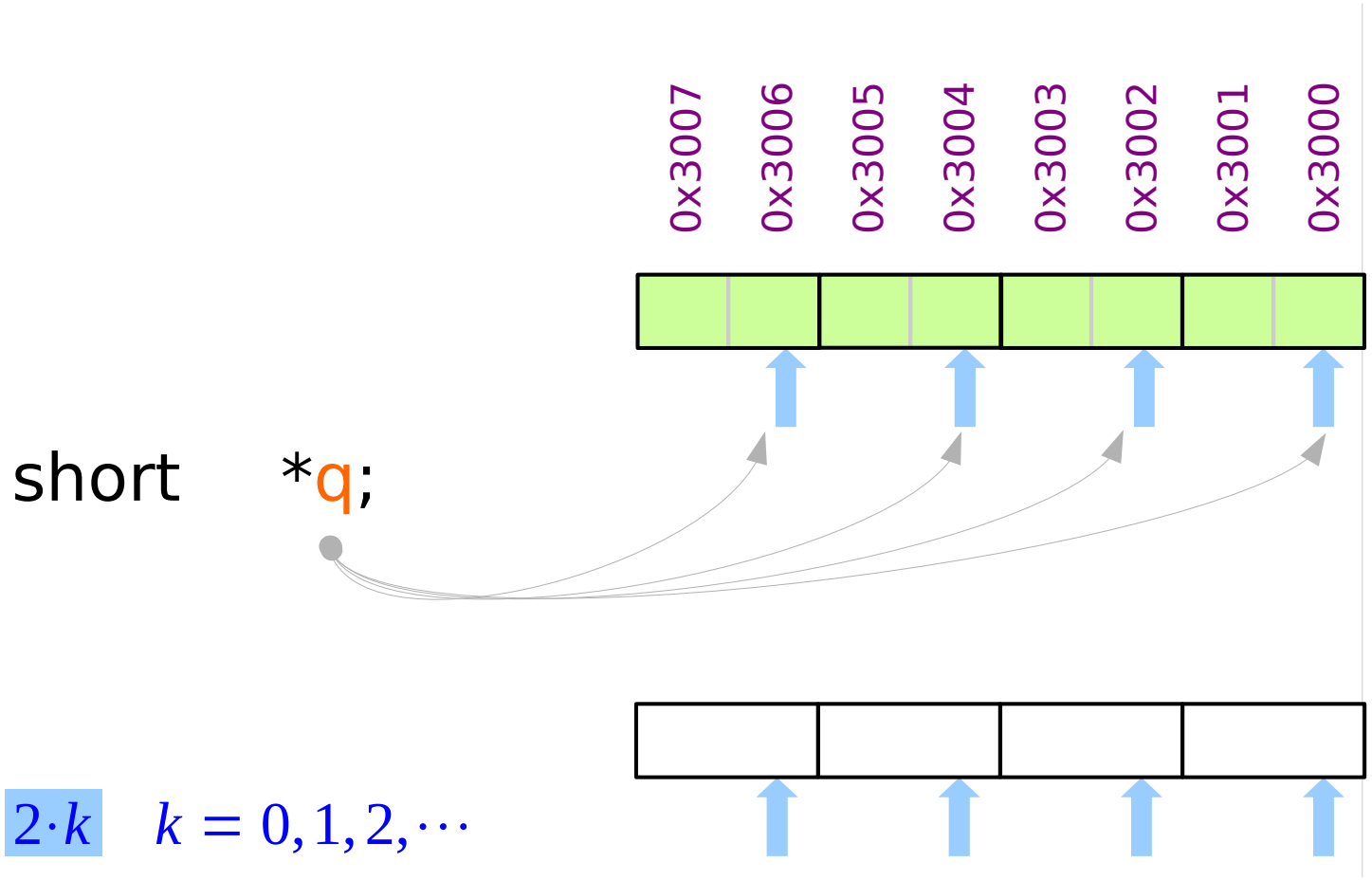


Memory Alignment
in the Little Endian

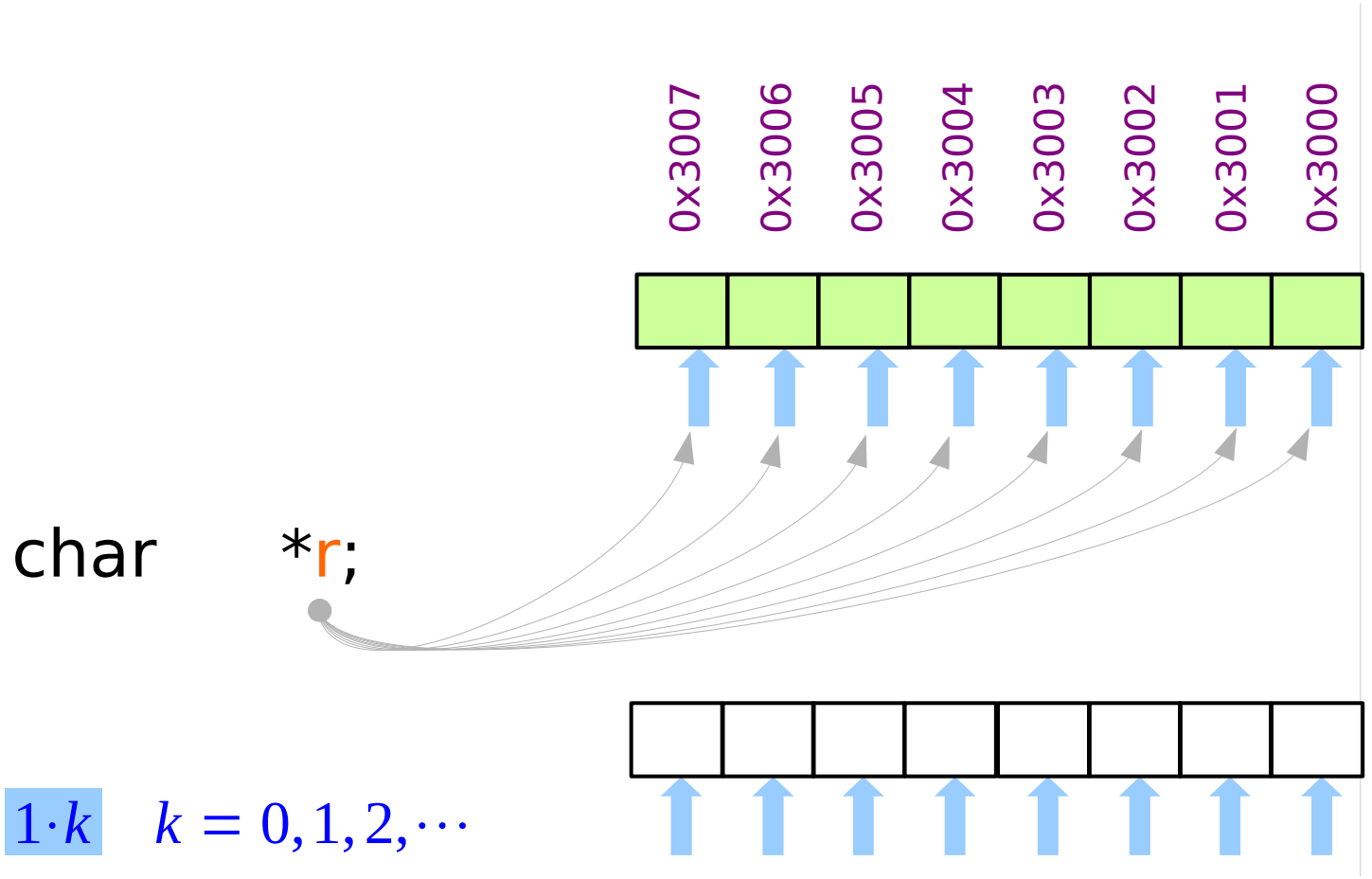
Possible addresses for **int** values



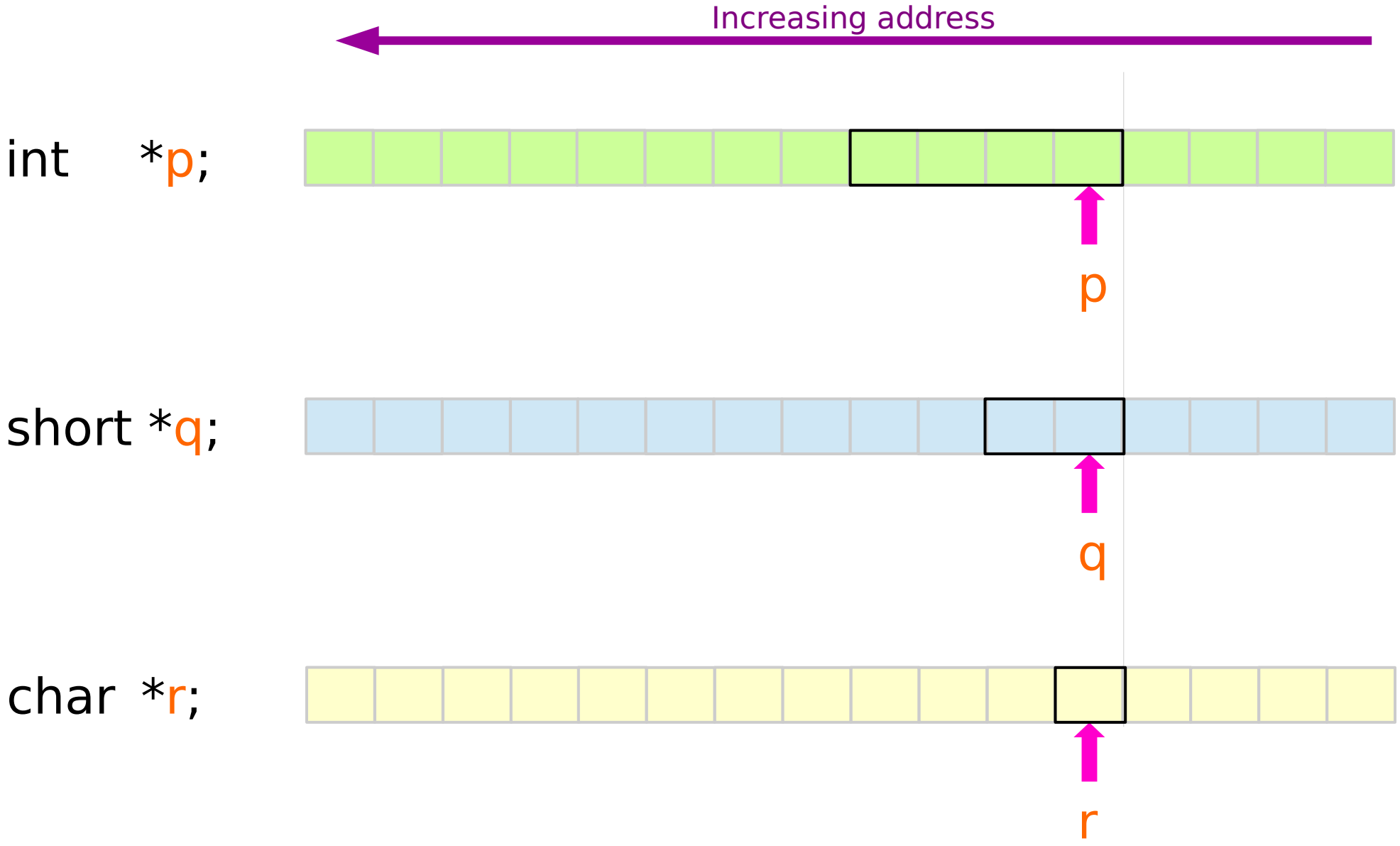
Possible addresses for **short** values



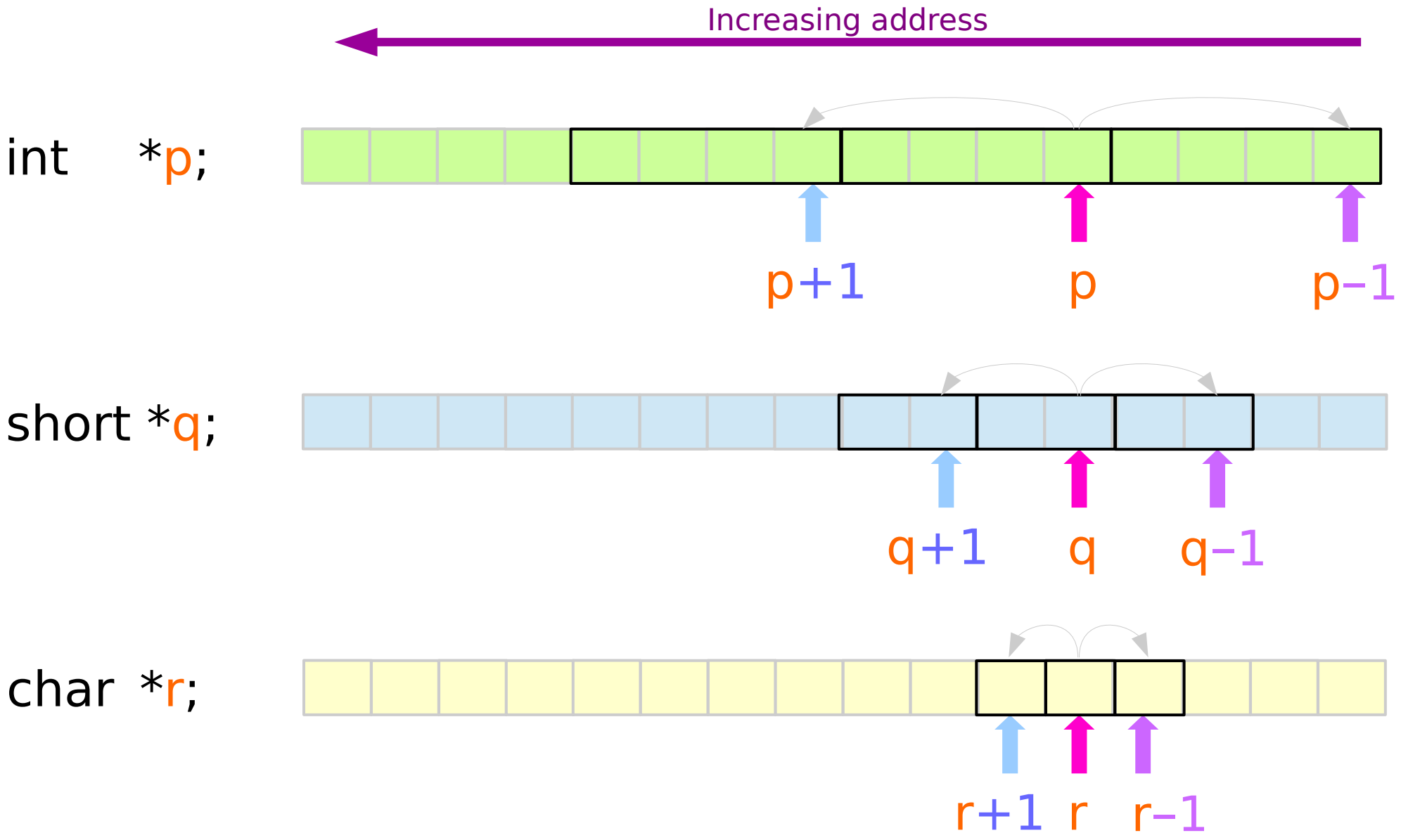
Possible addresses for **char** values



Data size at the pointed address



Incrementing / decrementing pointers

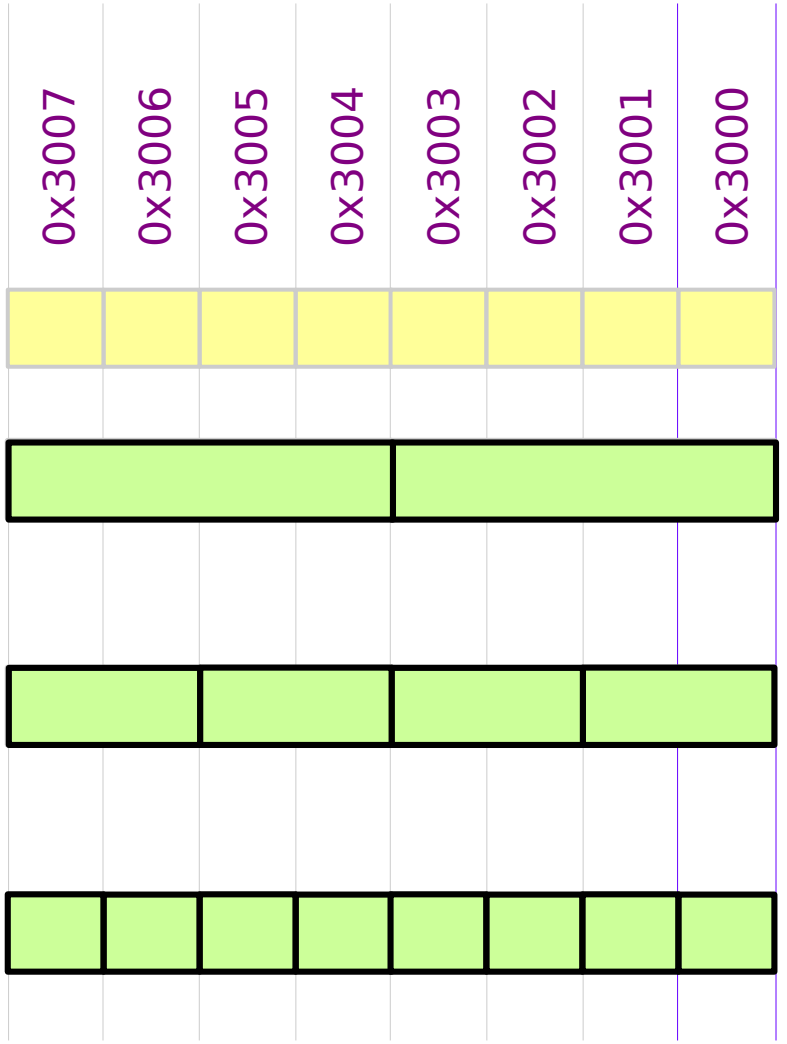


Memory Alignment (1) - allocation of variables

enforced by compilers

efficient memory access

```
int   a;

short b;

char  c;
```



Memory Alignment (2) – integer multiple addresses

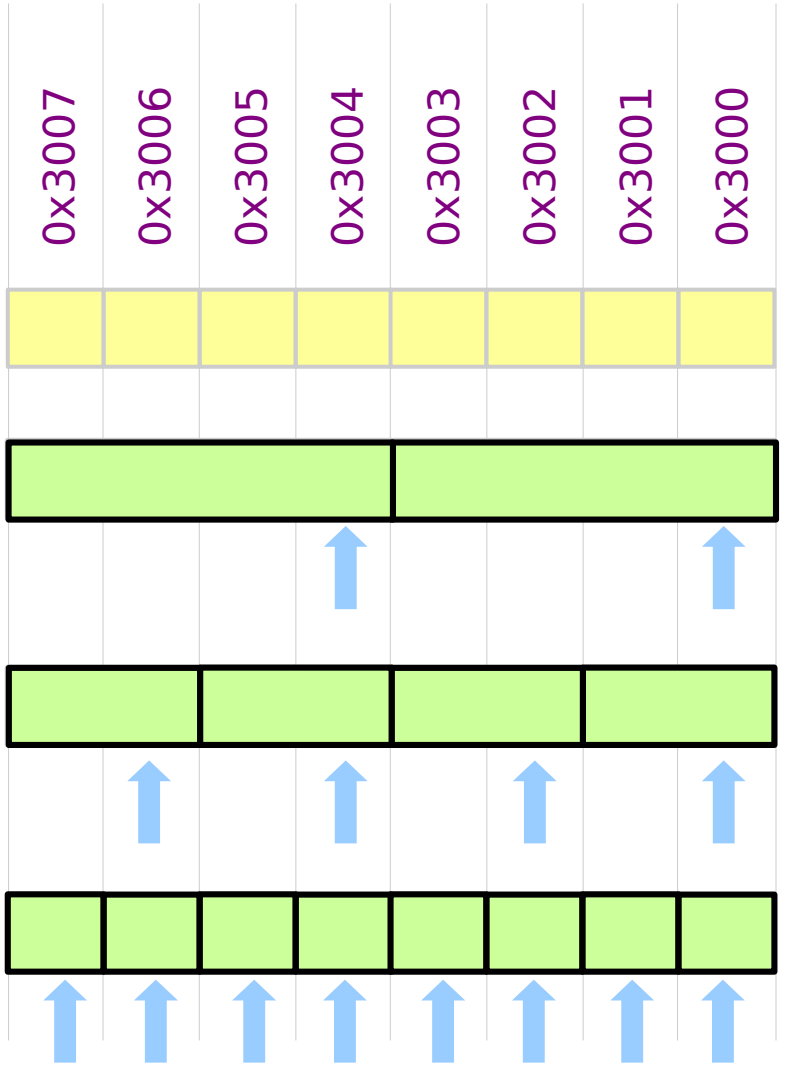
Memory Alignment:
the data **address** is a
multiple of the data **size**.

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

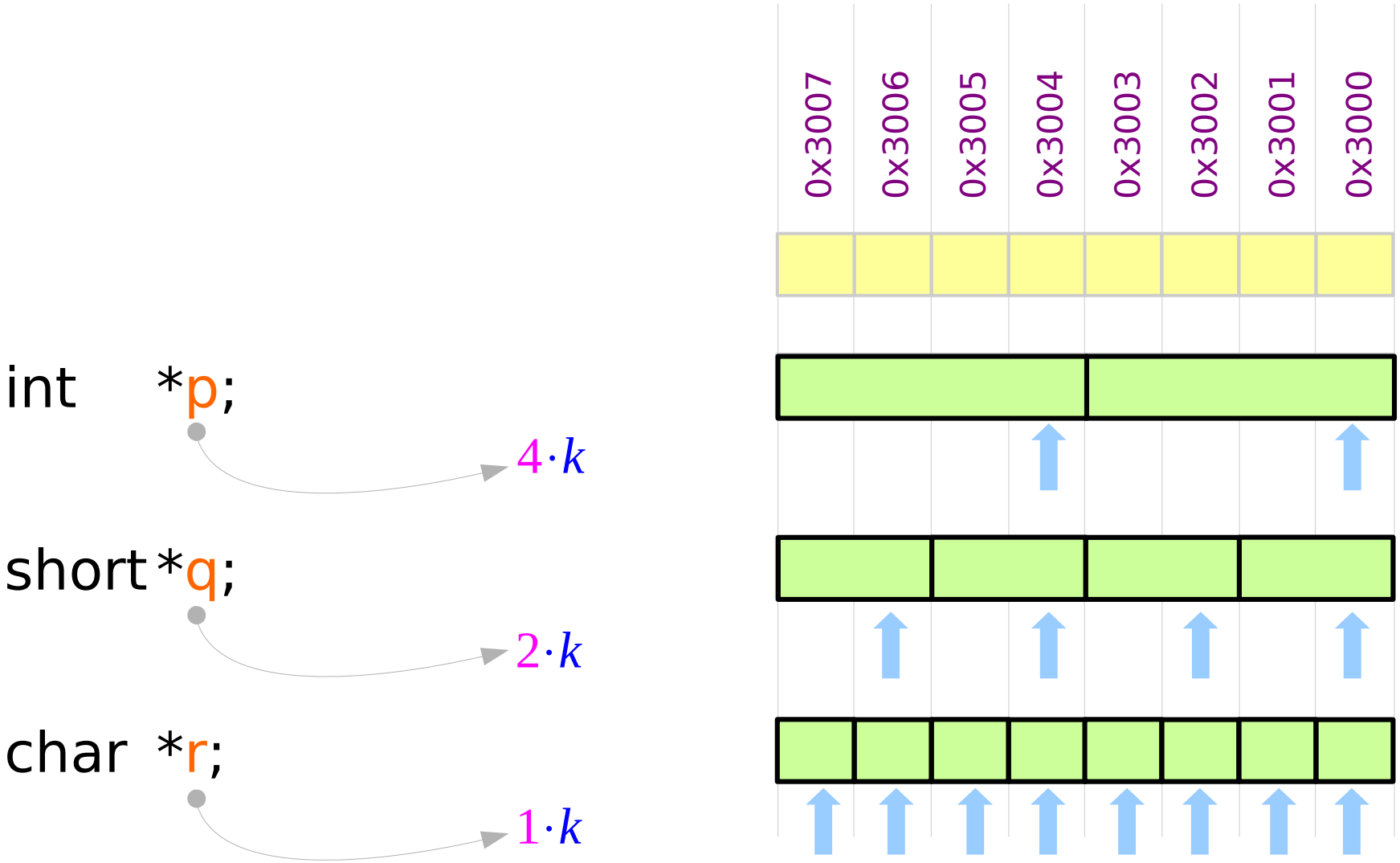
$integer\ addresses = 4 \cdot k$

$short\ addresses = 2 \cdot k$

$character\ addresses = 1 \cdot k$



Memory Alignment (3) – pointed addresses

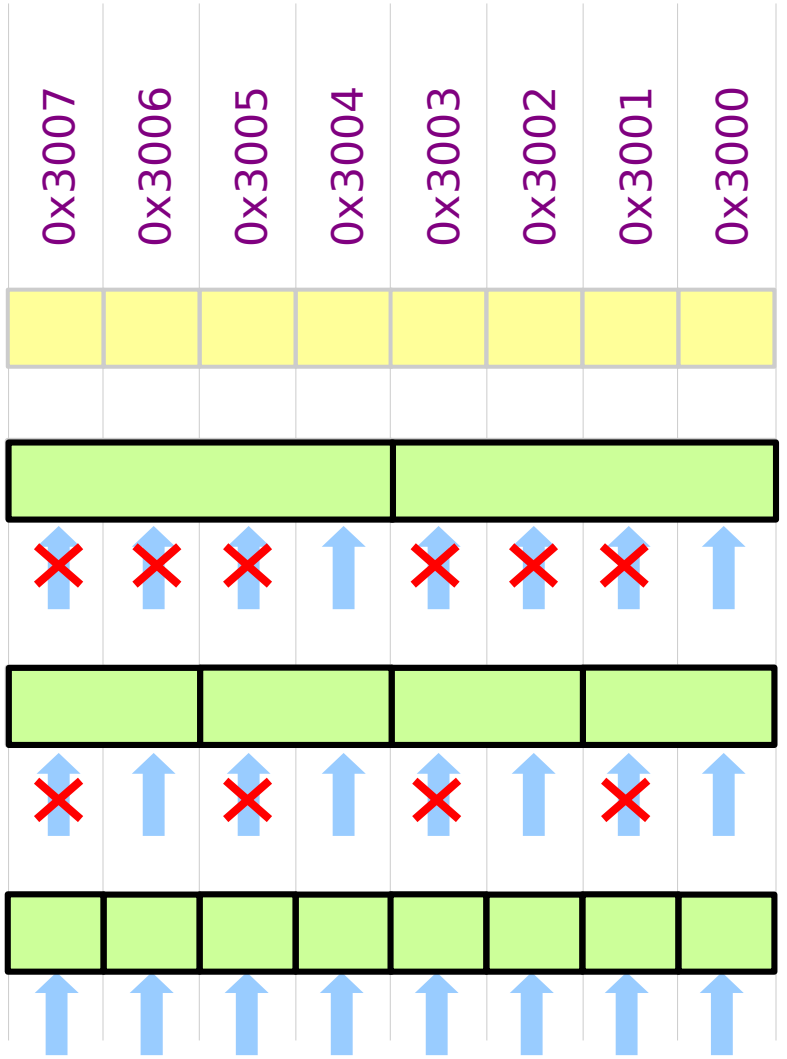


Memory Alignment (4) – non-pointed addresses

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

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

$2 \cdot k + 1$



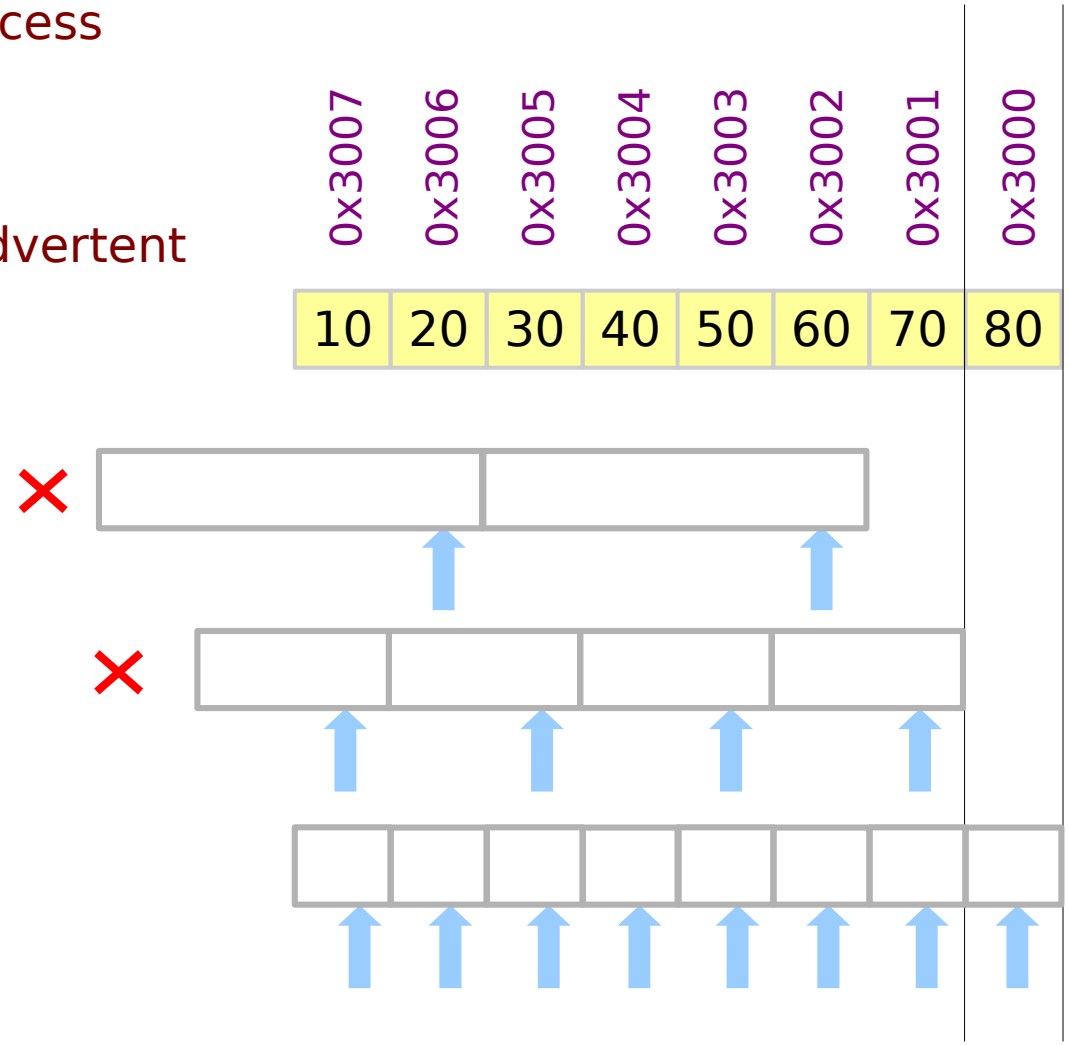
Memory Alignment (5) – broken alignment

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

(Low Efficiency)

This can happen by using inadvertent
pointer type casting

```
int    *p;

short *q;

char  *r;
```



Pointer Type Cast

Changing the associated data type of an address

long 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

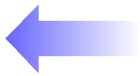
long a;

address of a long value

&a

int * p;

address of an int value



p = (int *) &a

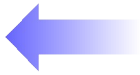
short* q;

address of a short value

q = (short *) &a

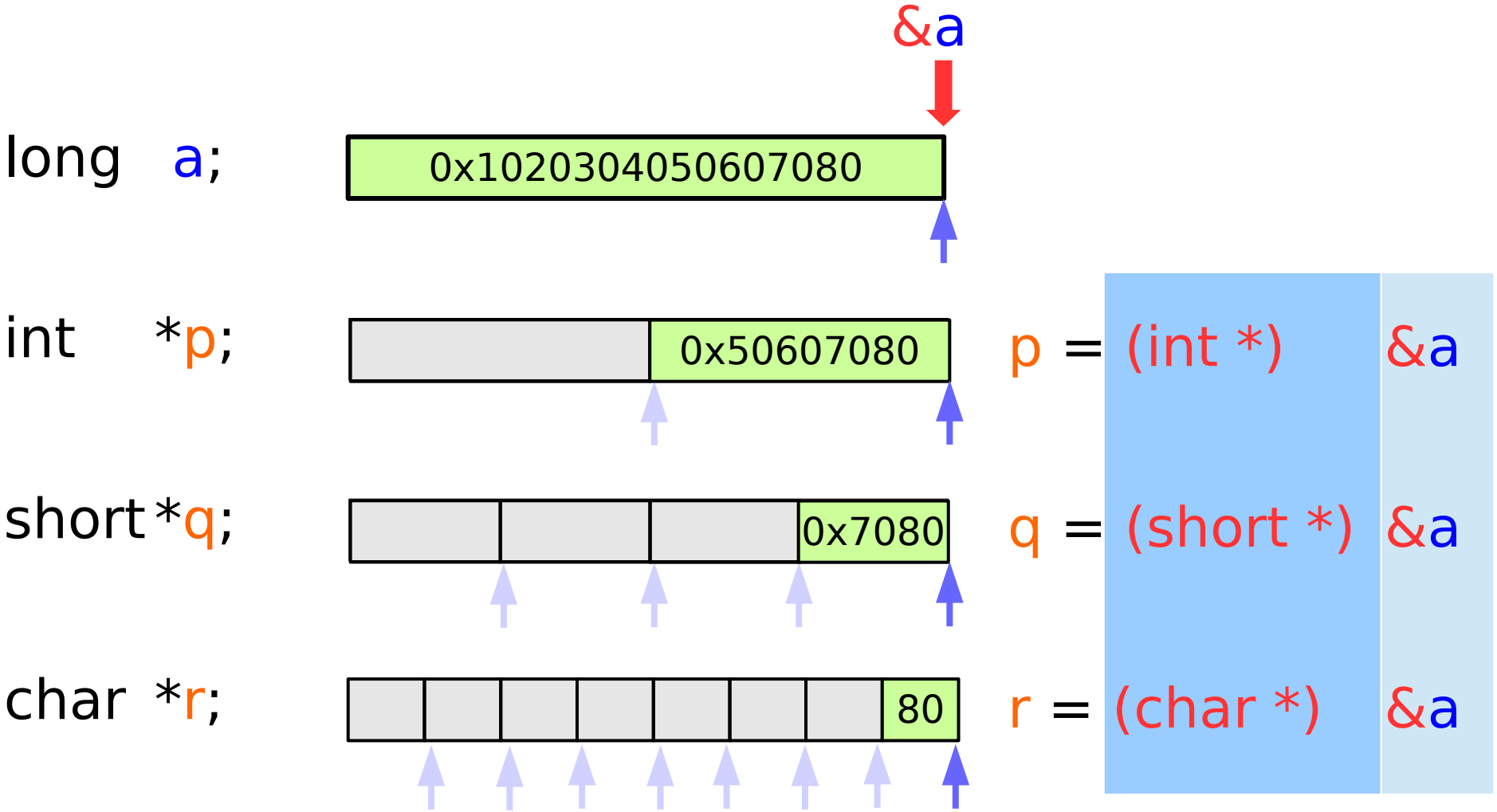
char * r;

address of a char value



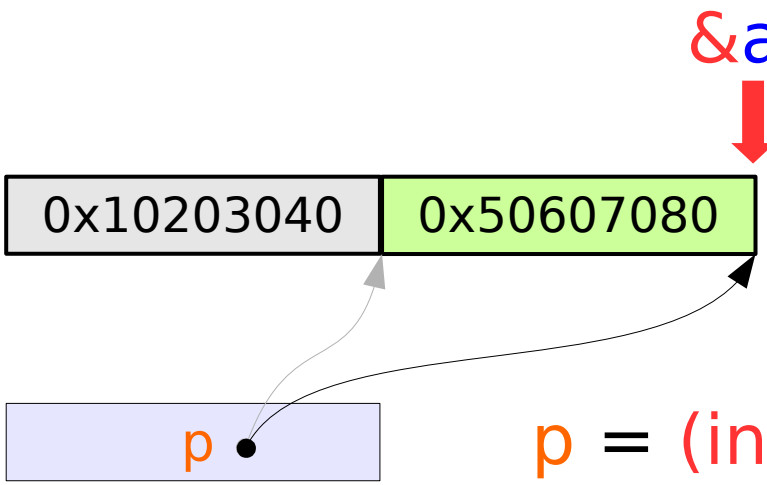
r = (char *) &a

Re-interpretation of memory data – case I



Pointer Type Cast

```
long   a;

int    *p;

short *q;

char  *r;
```

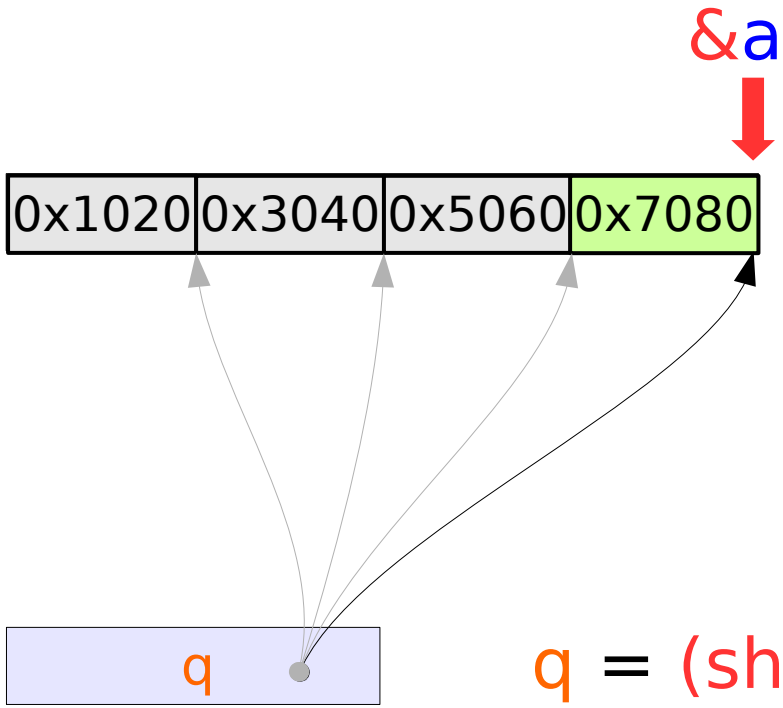


p = (int *) &a

*p ➡ 0x50607080

Integer Pointer Types

```
long   a;

int    *p;

short *q;

char  *r;
```

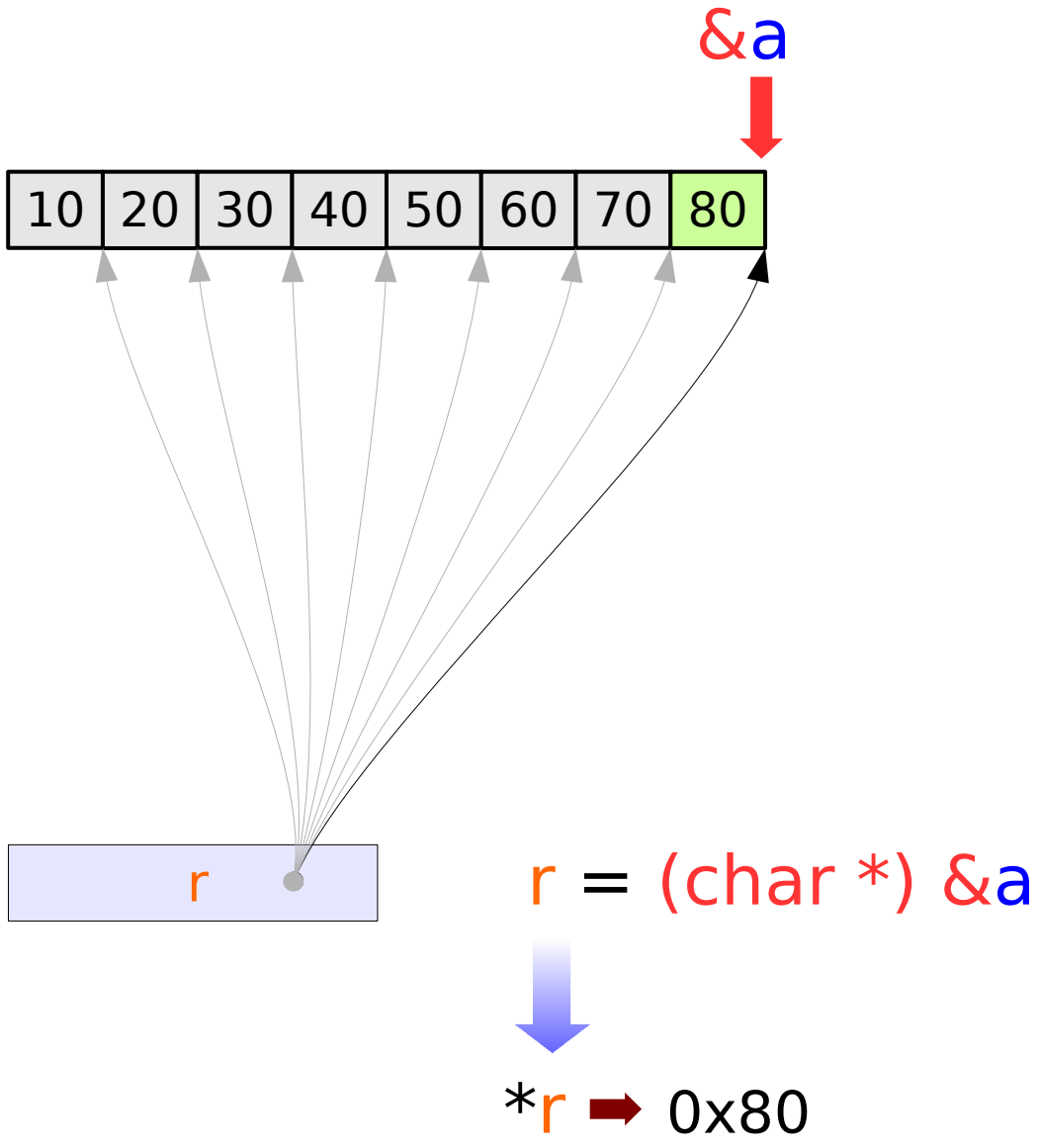


q = (short *) &a

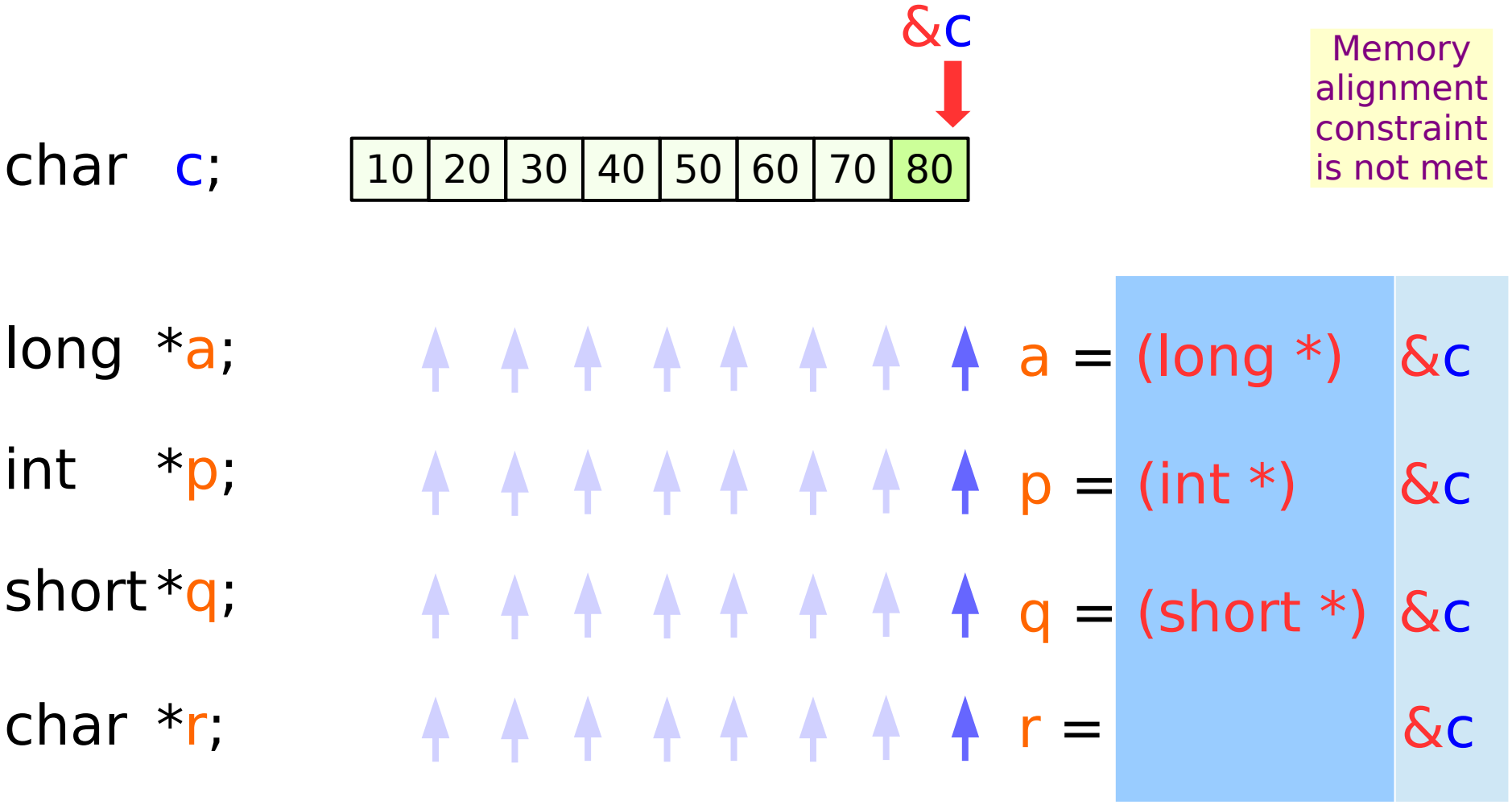
*q ➡ 0x7080

Integer Pointer Types

```
long   a;

int    *p;

short *q;

char  *r;
```



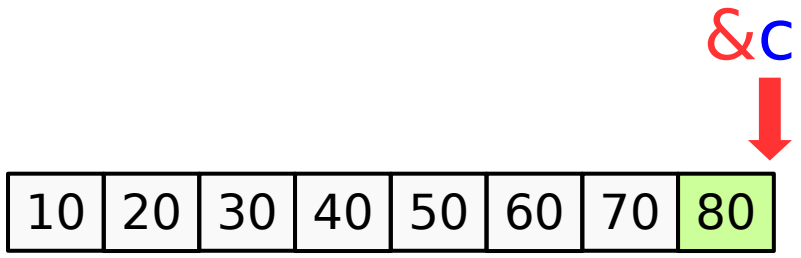
Re-interpretation of memory data – case II



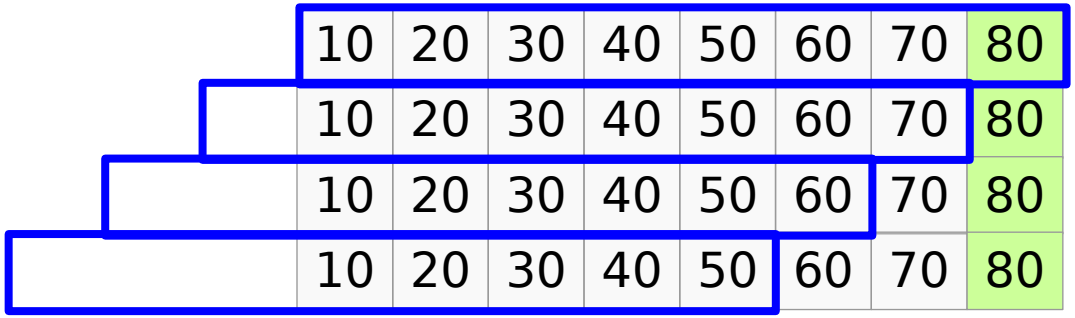
In this case, the memory alignment constraint can be broken

Pointer Type Cast

```
char   c;
```

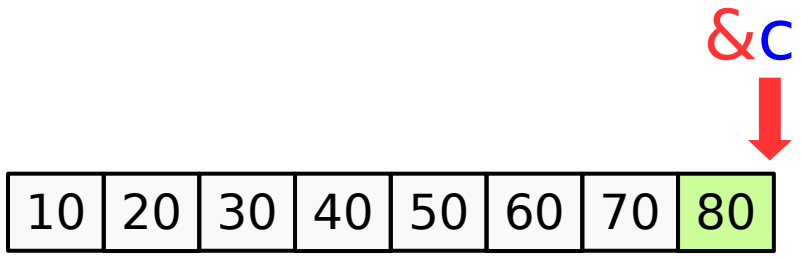



```
long  *a;
```

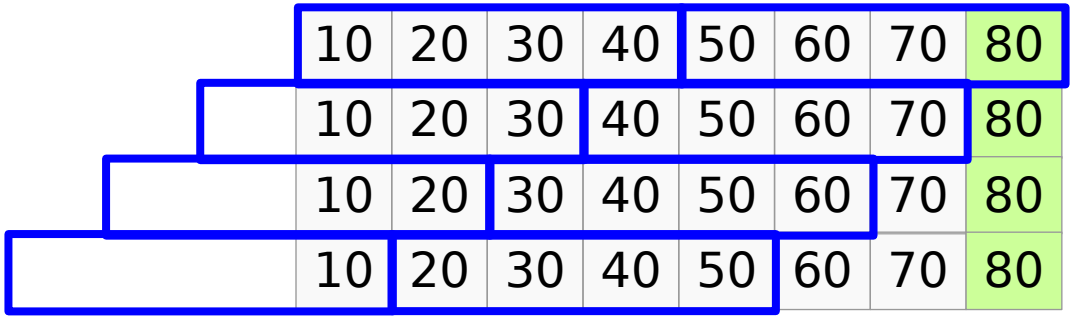


Pointer Type Cast

```
char    c;
```

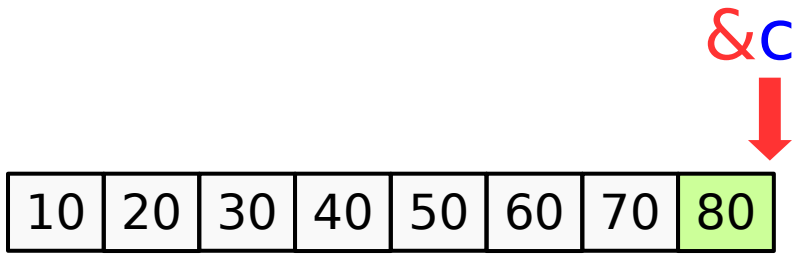



```
int *p;
```

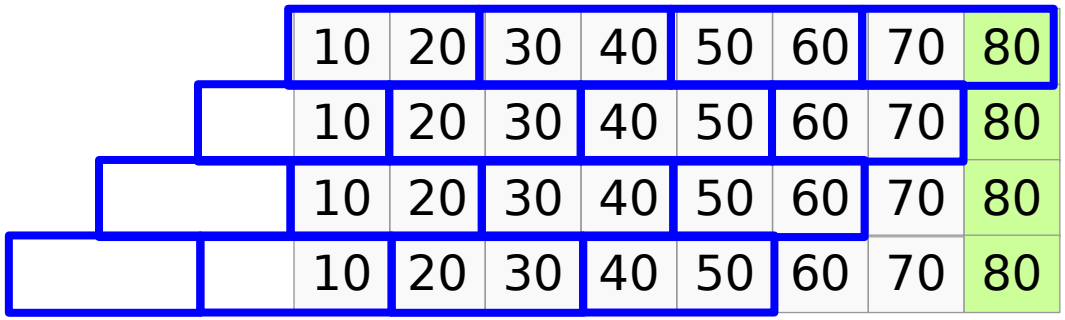


Pointer Type Cast

char c;



short *q;



const pointers

const type, const pointer type (1)

```
const int * p;

int * const q ;

const int * const r ;
```

read only integer value

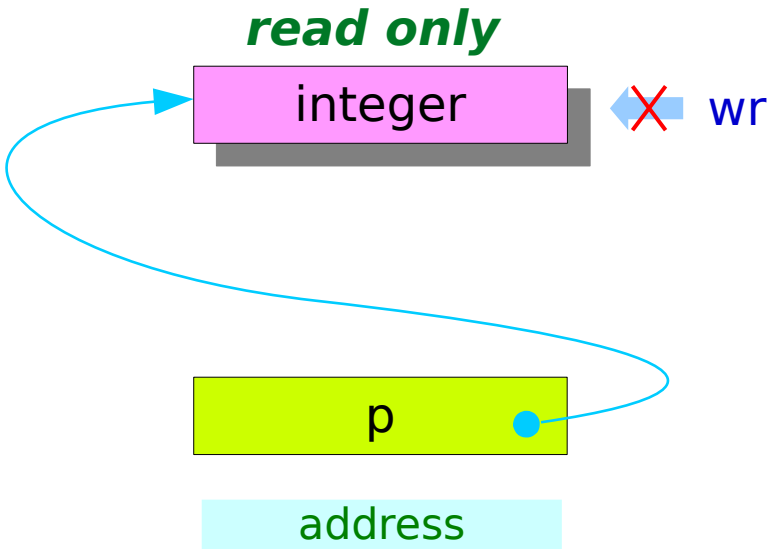
read only integer pointer

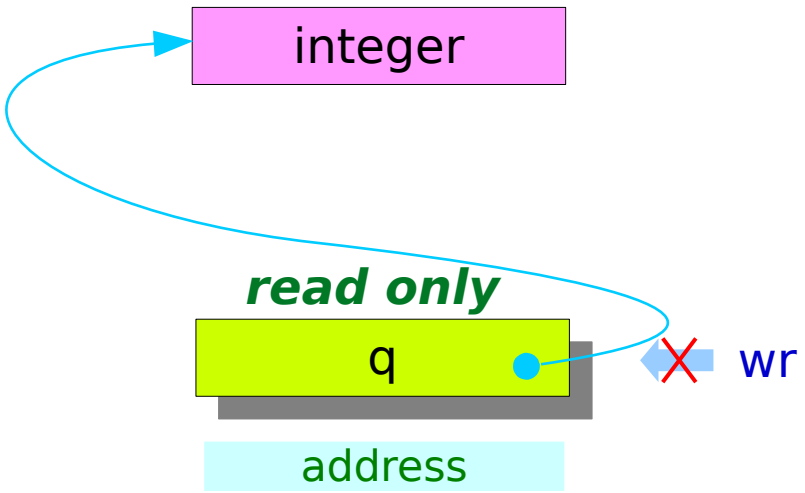
read only integer value
read only integer pointer

const type, const pointer type (2)

const int * **p;**

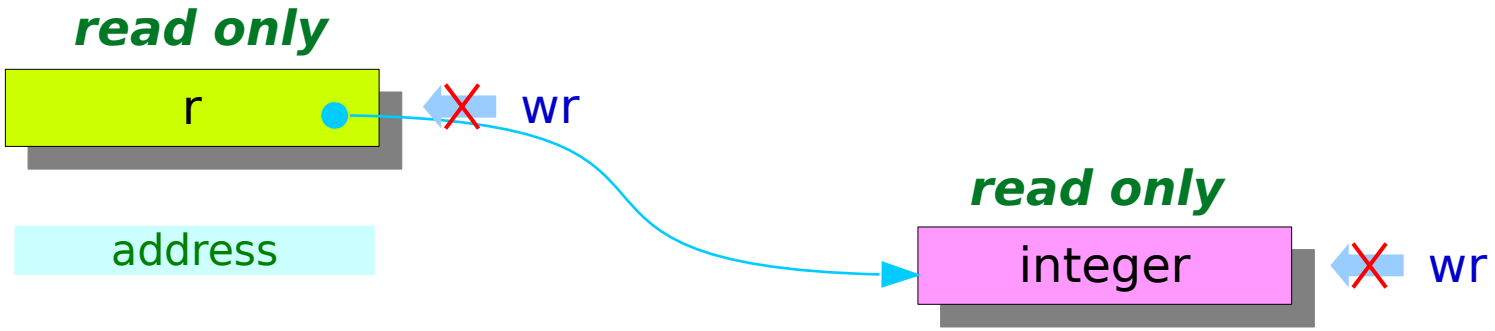
int * **const q** **;**



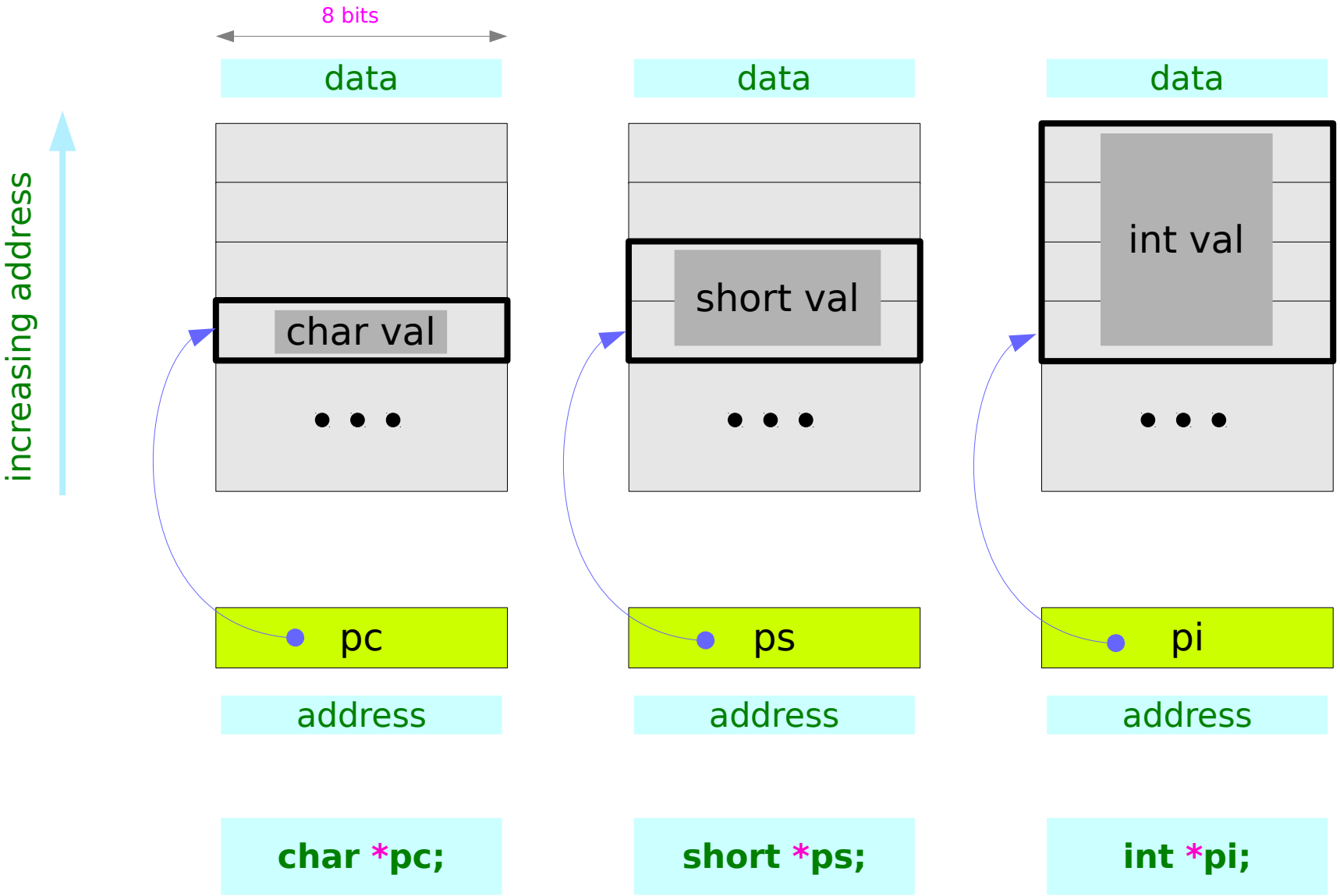


const type, const pointer type (3)

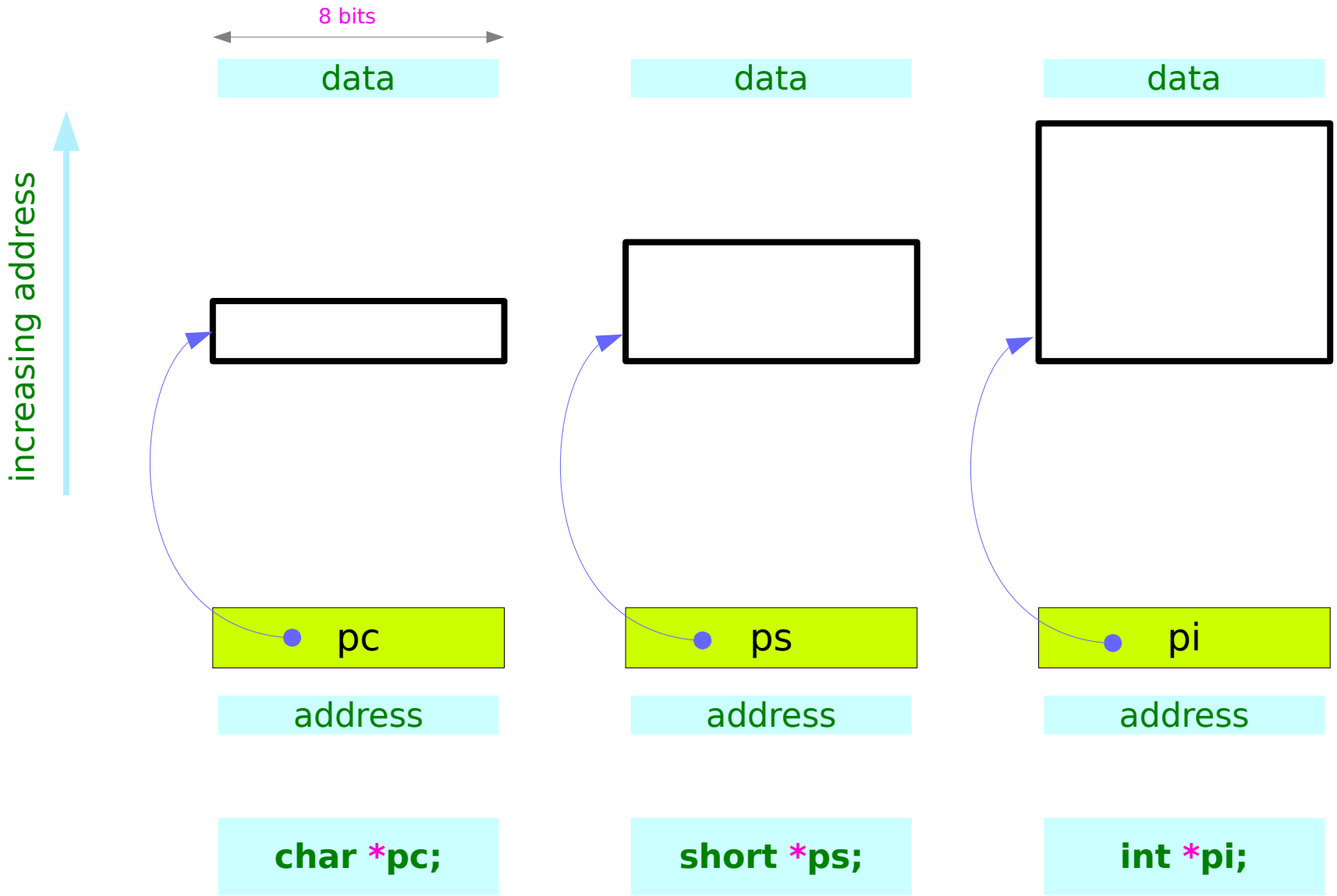
```
const int * const r ;
```



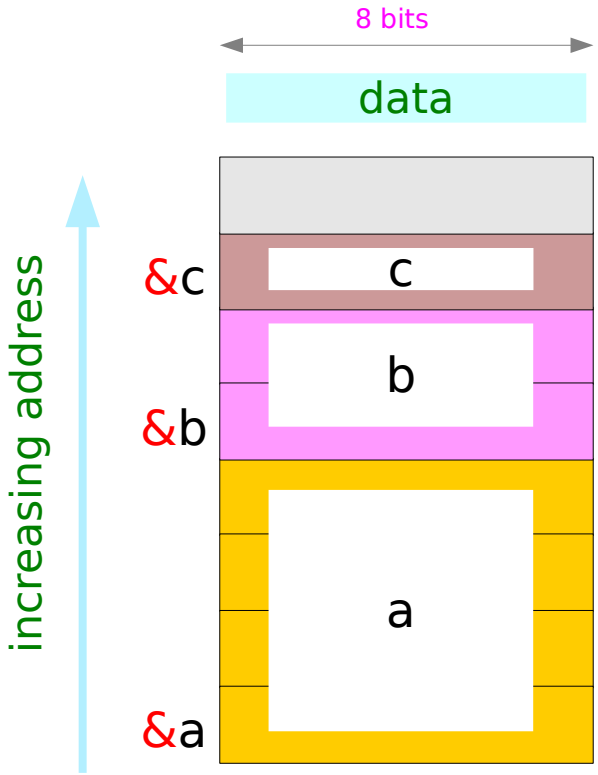
Pointer Types and Associated Data



Pointer Types

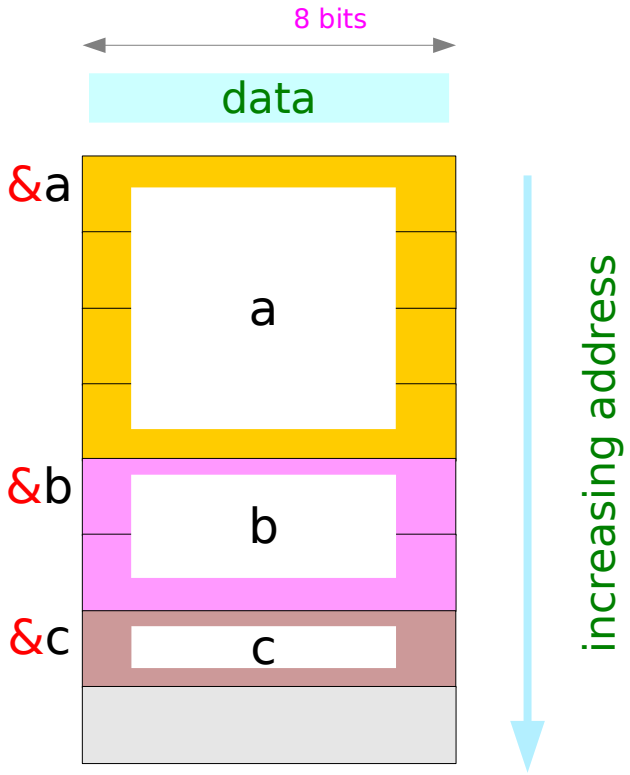


Little Endian Example

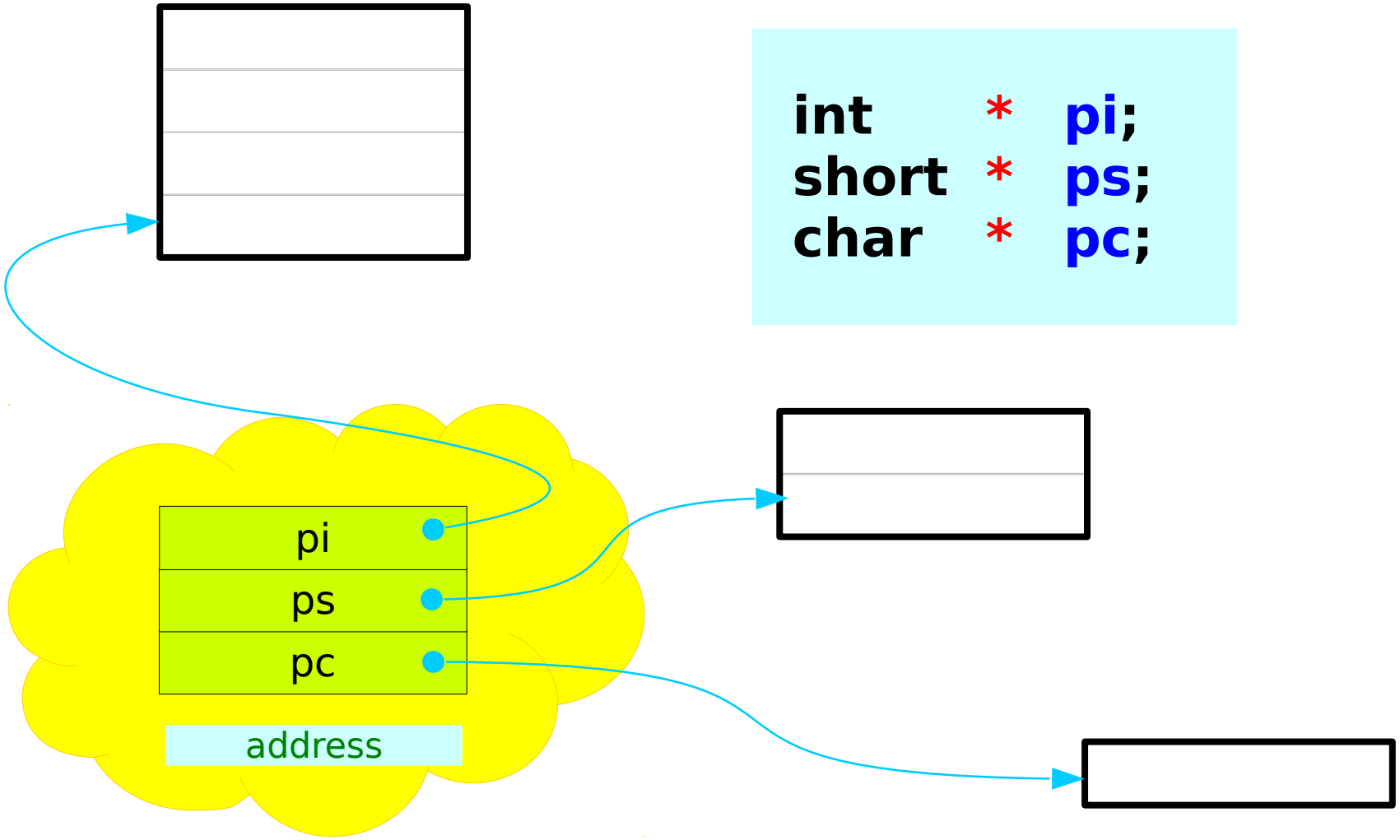



```
int   a;
short b;
char  c;
```

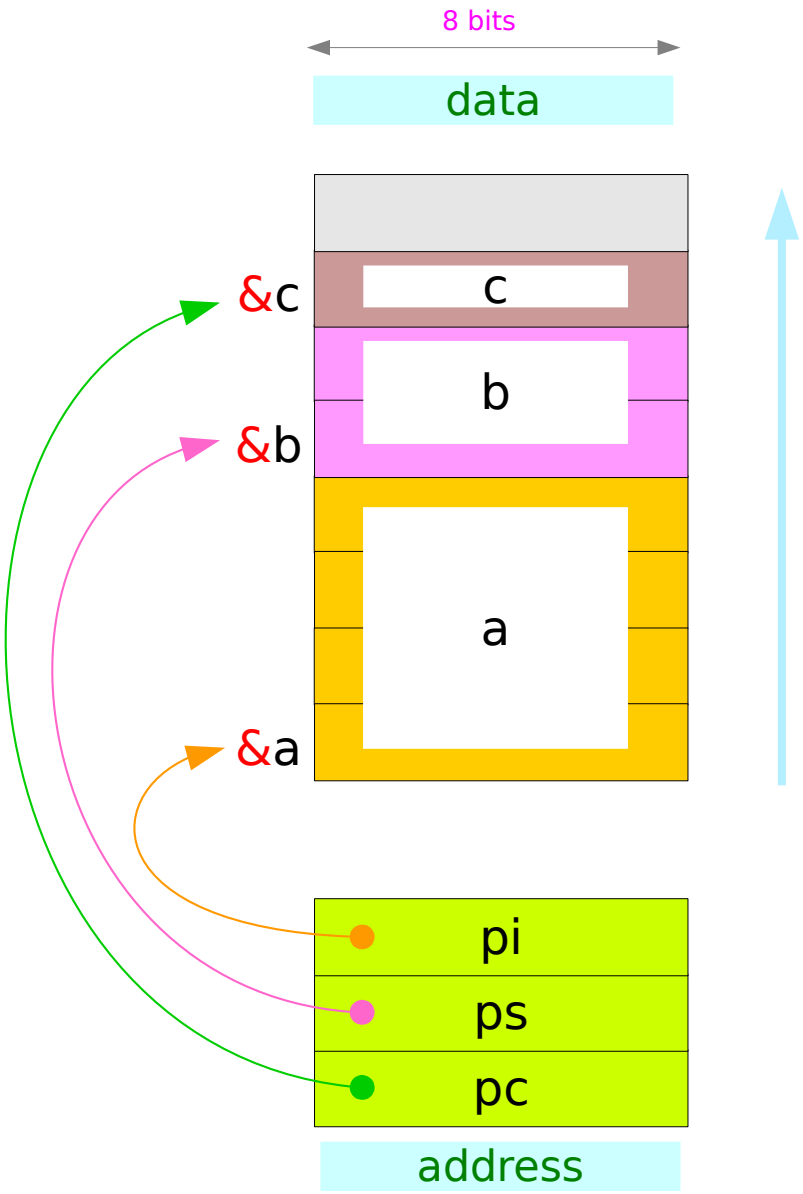
the order of definition



int *, short *, char * type variables



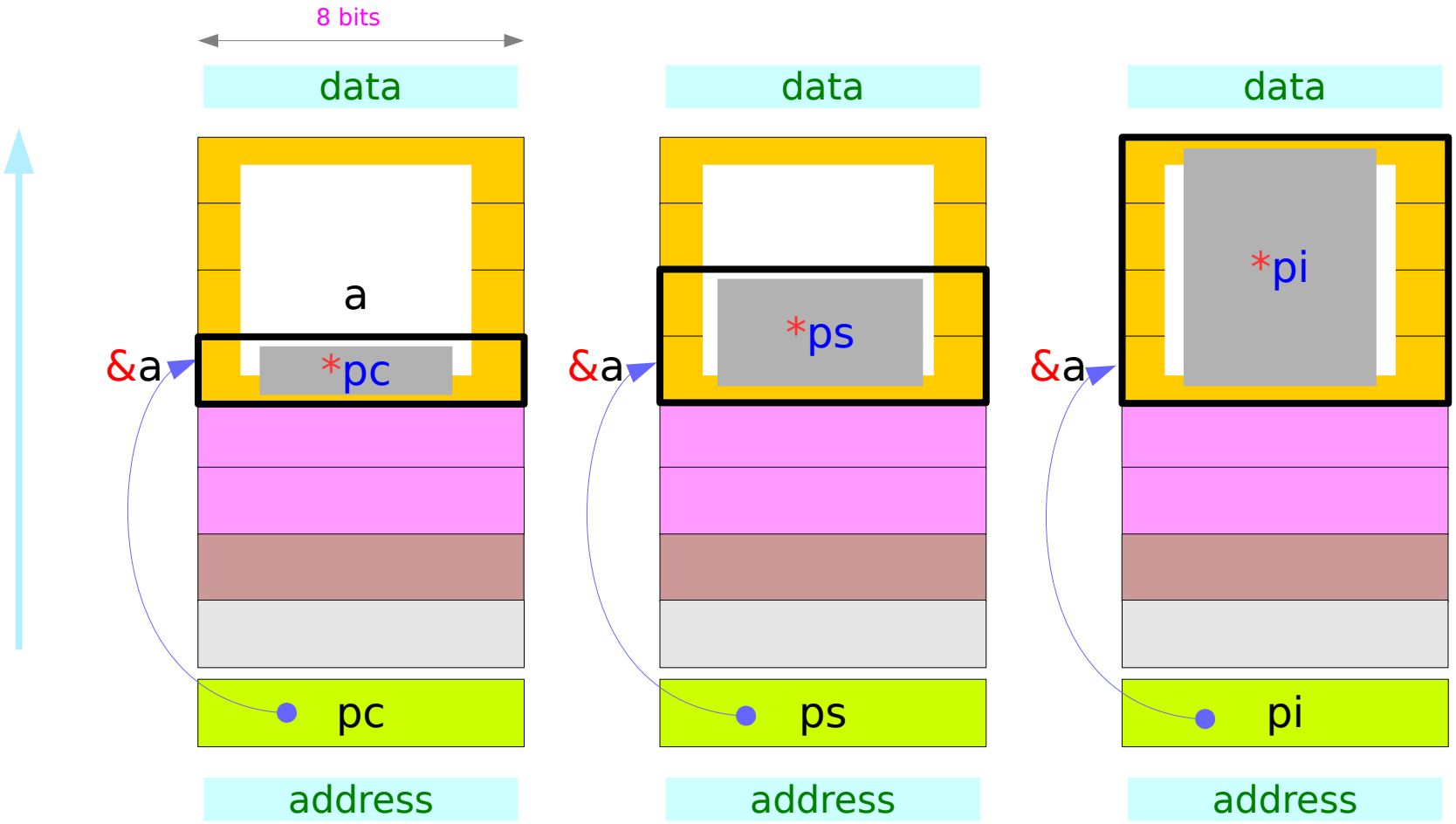
Pointer Variable Assignment




```
char   *   pc;
short  *   ps;
int    *   pi;

int    a;
short  b;
char   c;
```

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

Pointer Type Casting

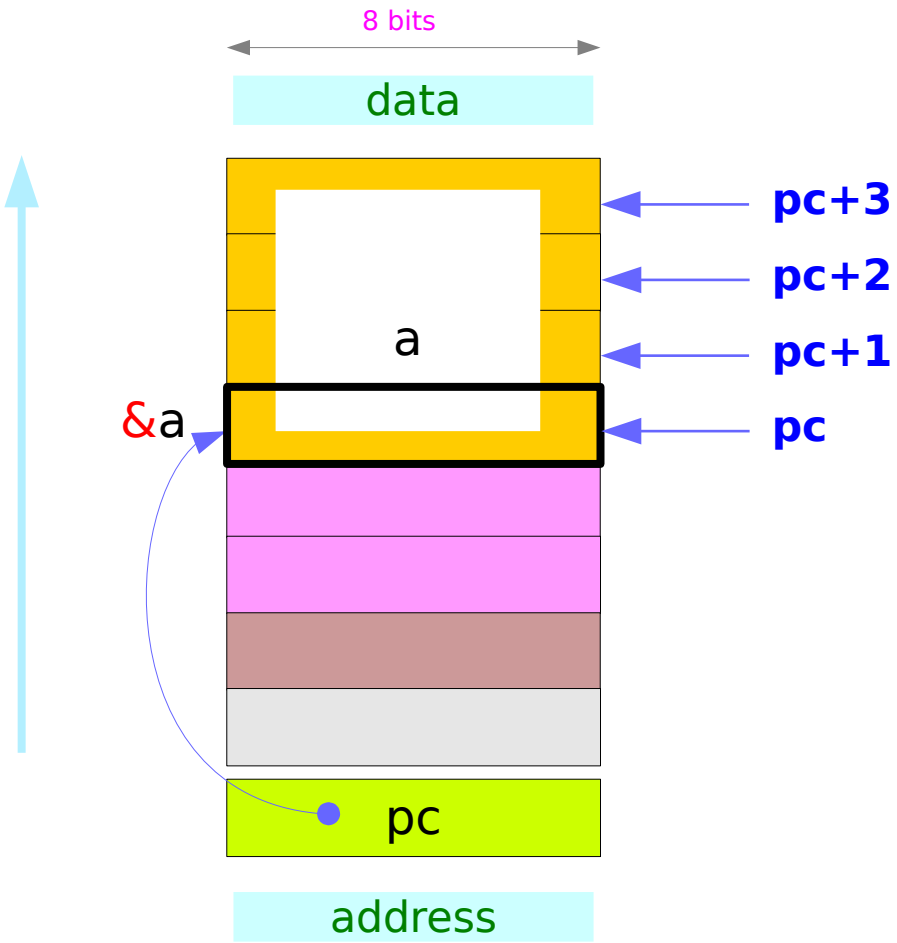


char *pc;
pc = (char *) &a

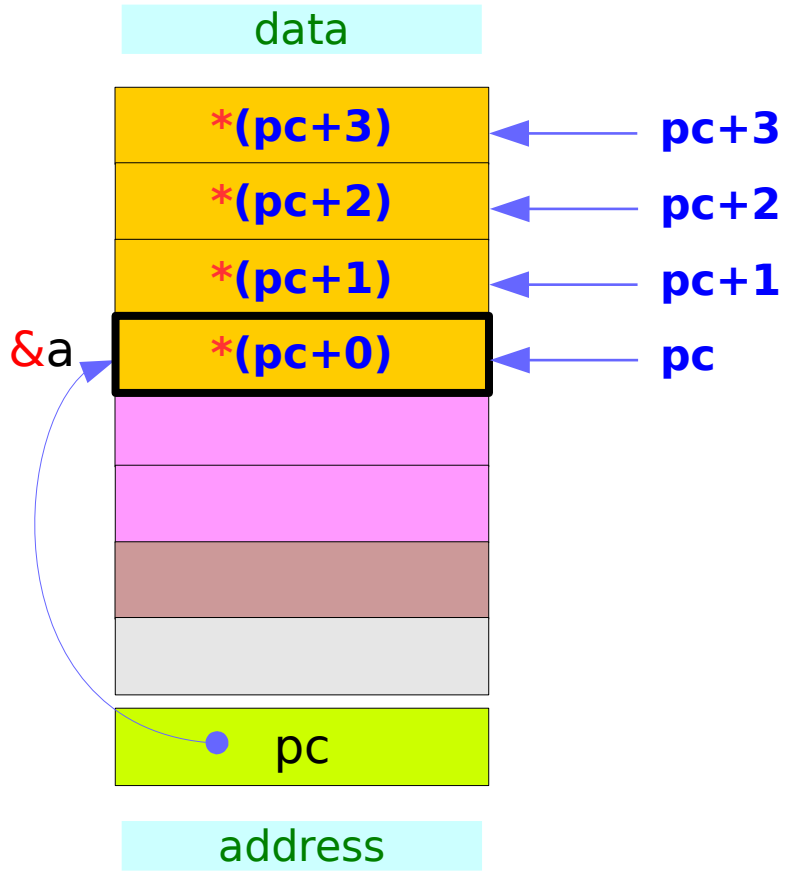
short *ps;
ps = (short *) &a

int *pi;
pi = (int *) &a

Accessing bytes of a variable



char *pc;
pc = (char *) &a

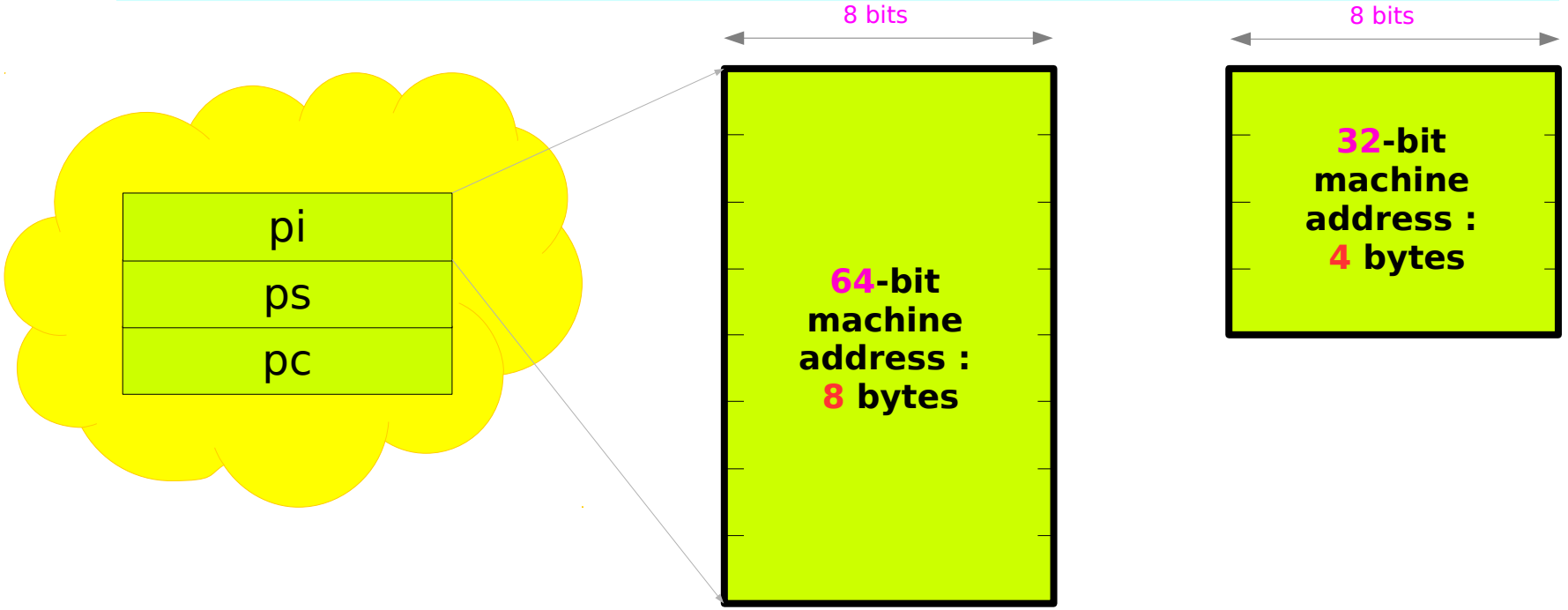


char *pc;
pc = (char *) &a

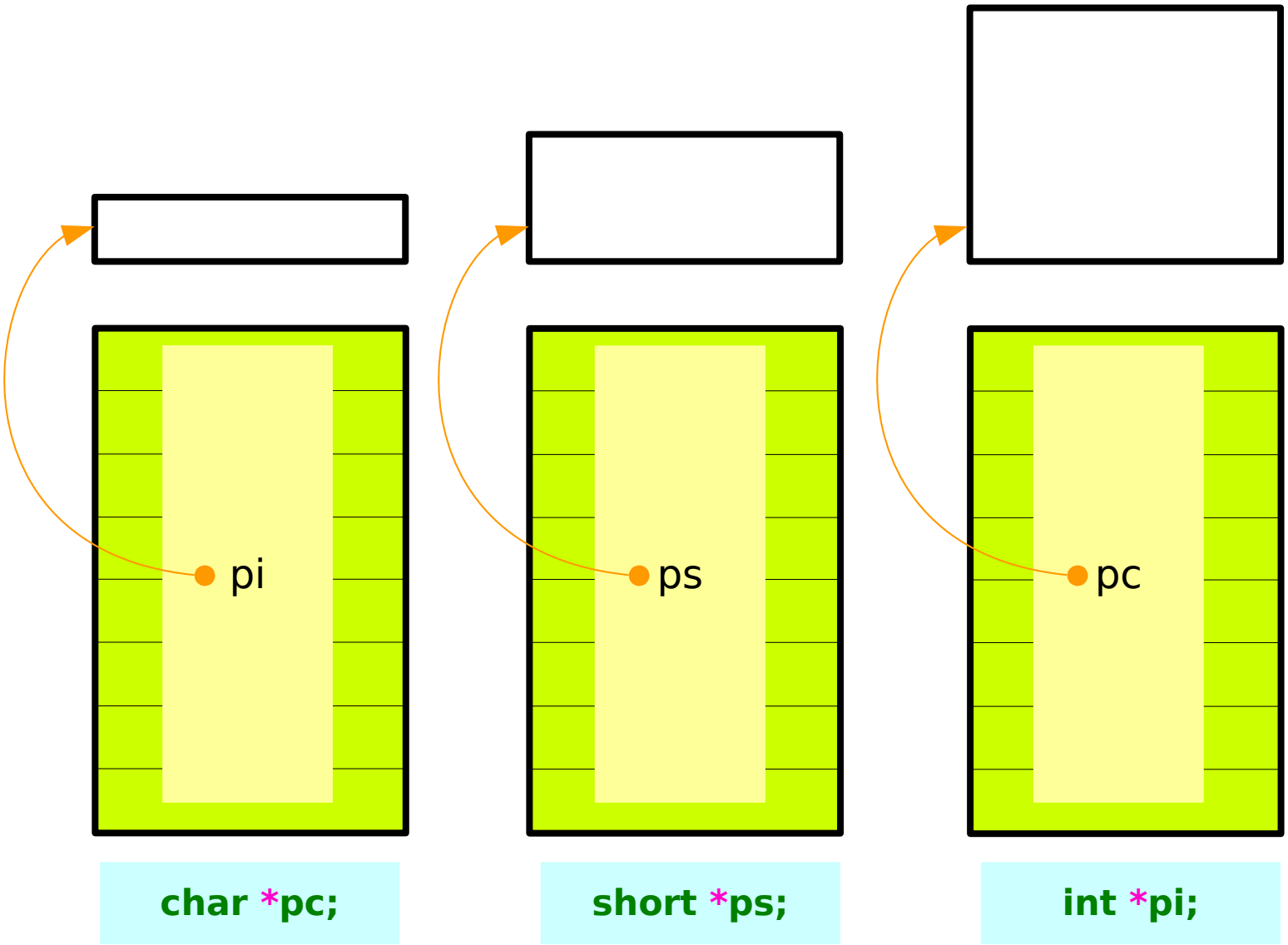
32-bit and 64-bit Address

32-bit machine : address : 4 bytes

64-bit machine : address : 8 bytes

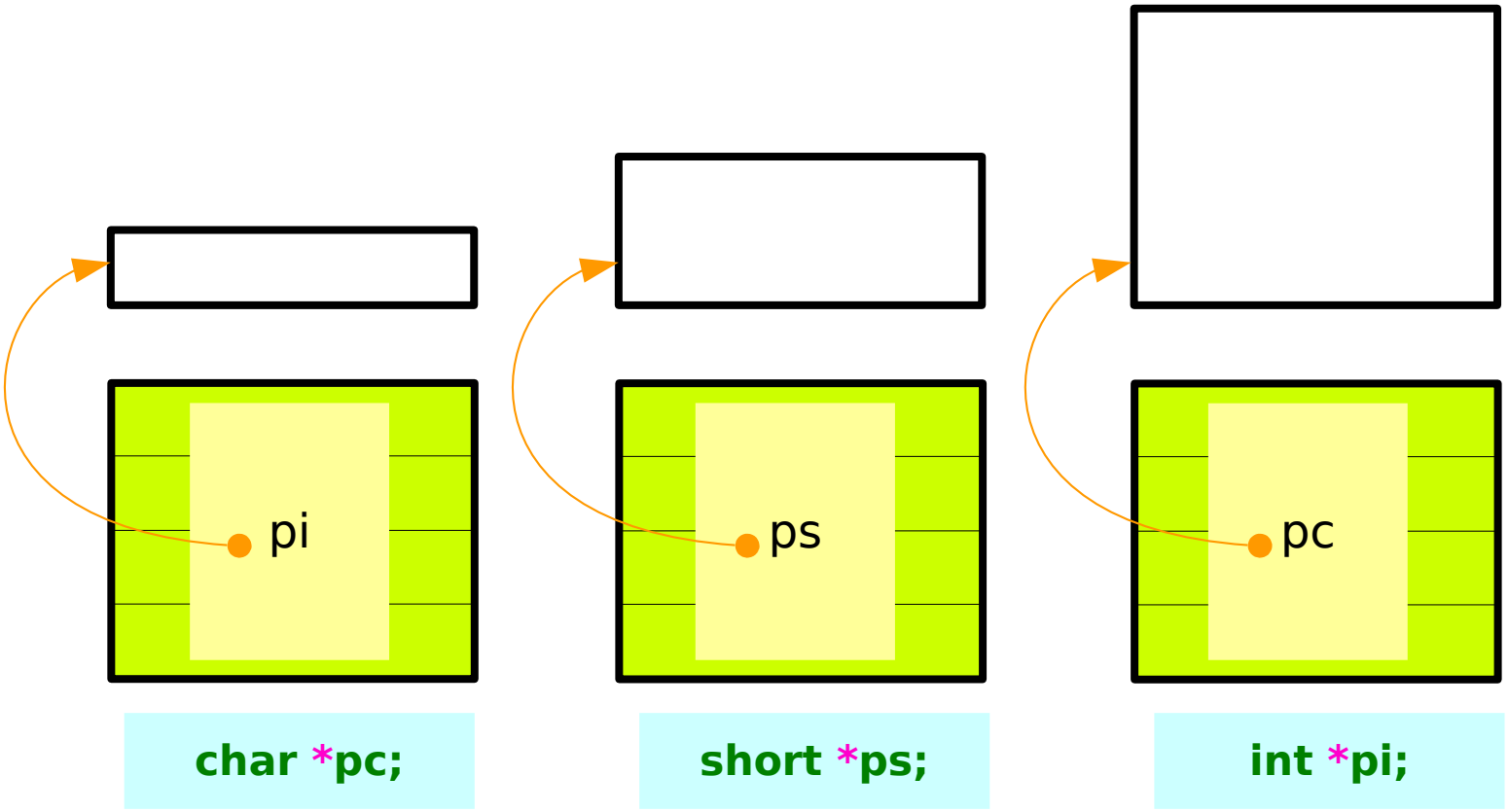


64-bit machine : 8-byte address

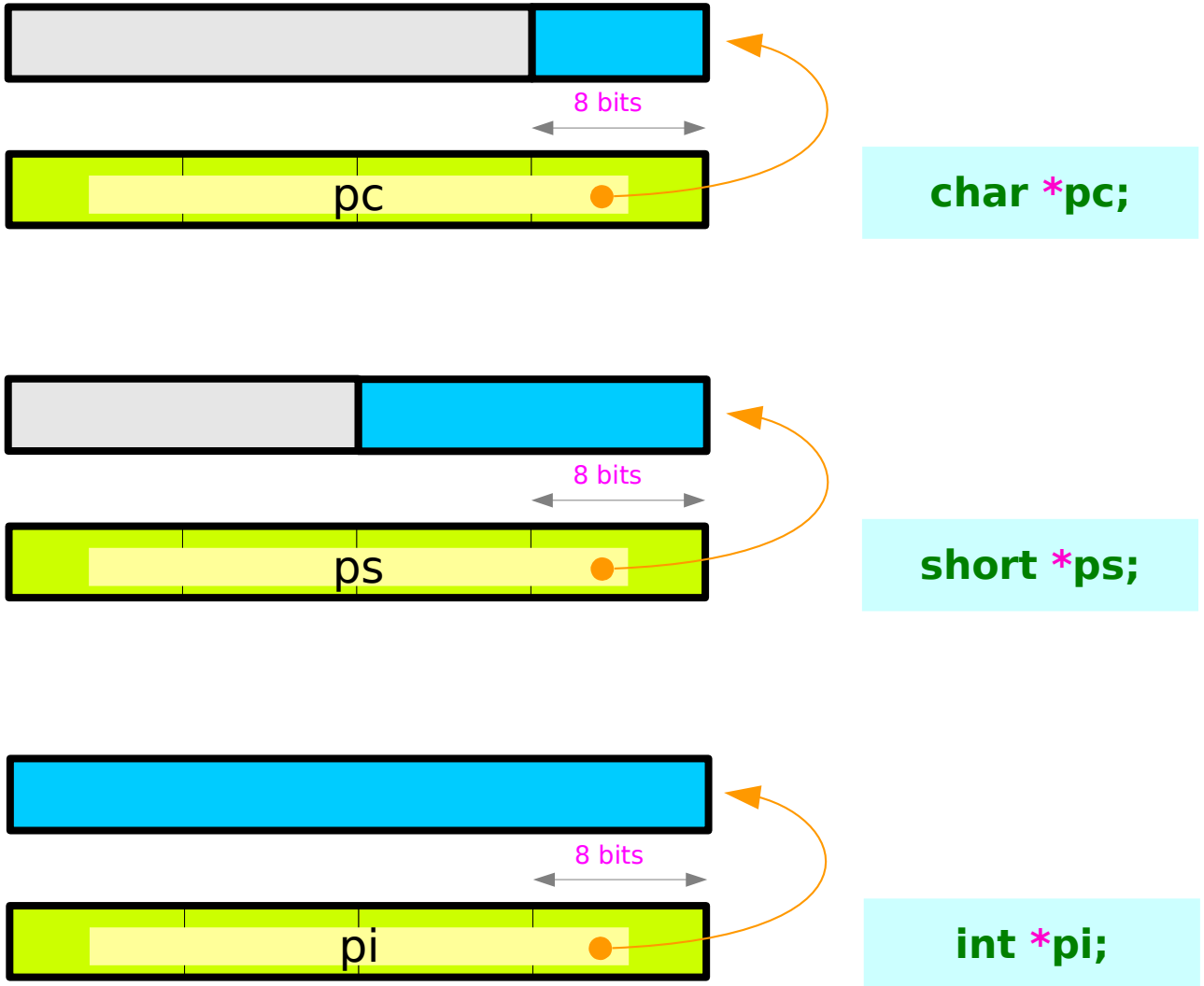


64-bit machine : 8-byte address & data buses

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