Type (1A)

Young Won Lim
2/17/18

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.

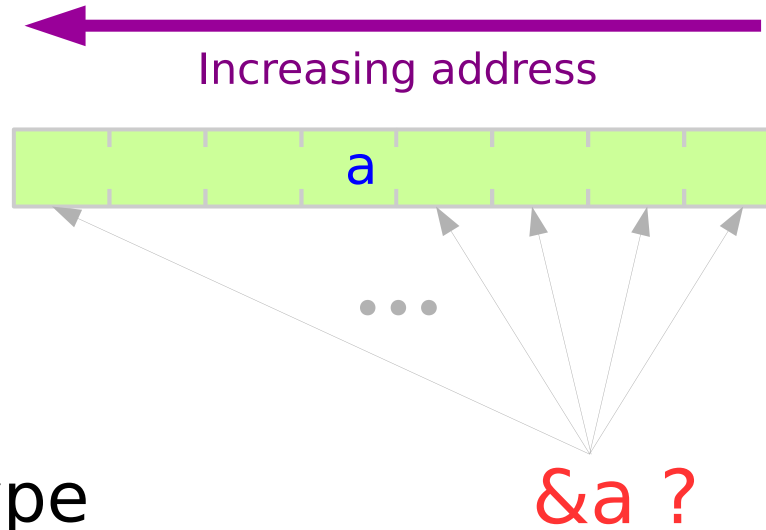
This document was produced by using LibreOffice.

Byte Address
Little Endian
Big Endian

Byte Address

long a;

8-byte size data type



Numbers in Positional Notation

long a = 0x1020304050607080; 8 (bytes)

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

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

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

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

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

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

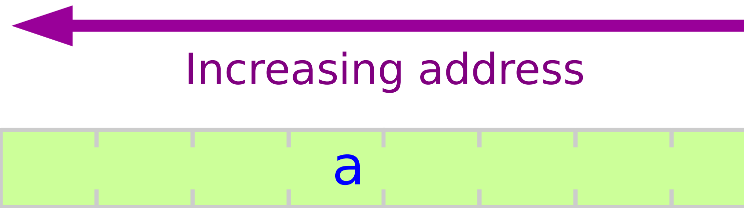
$a_2 = 0x60 \quad \cdots \quad 16^2$

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

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

Little / Big Endian Ordering of Bytes

long a;

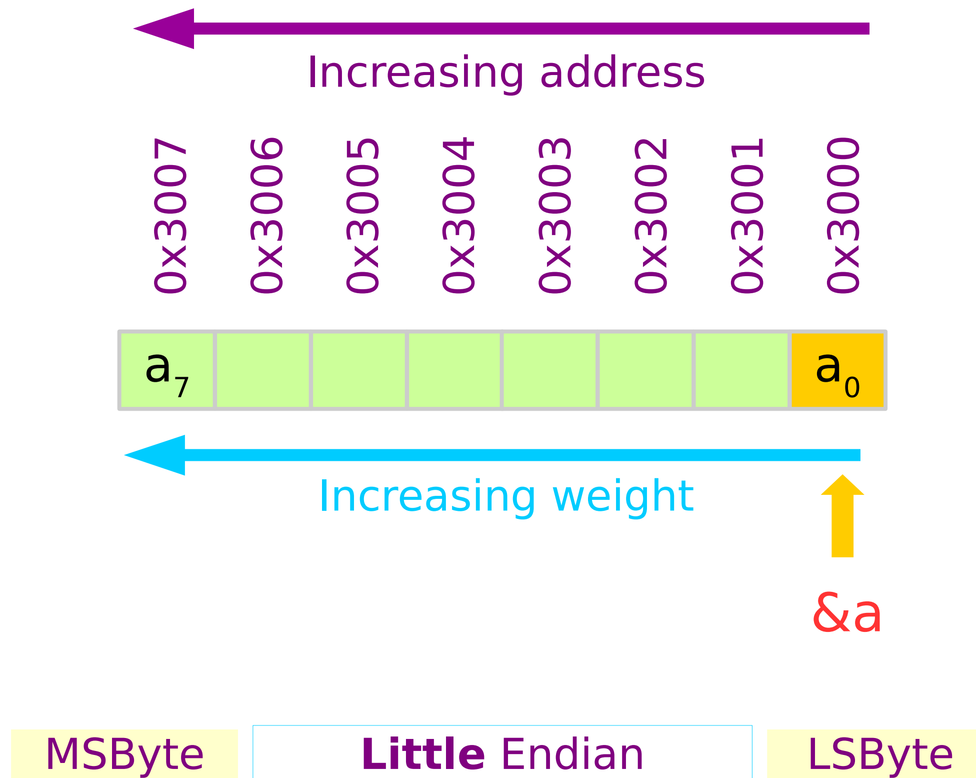


$a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$

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

Little Endian Byte Address Example

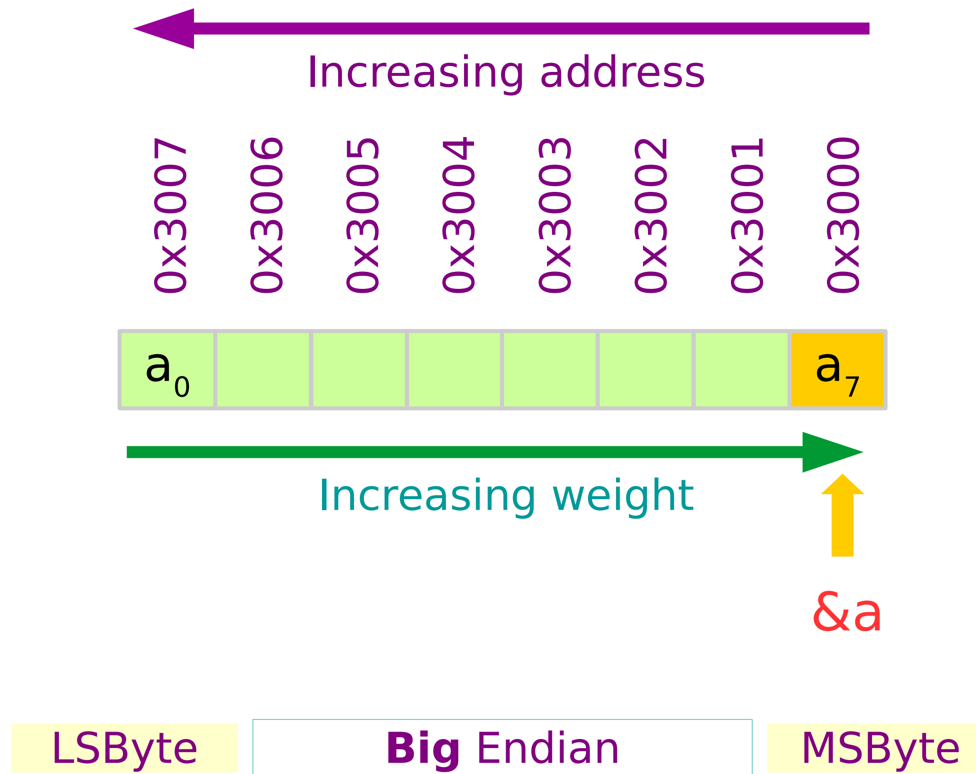
long a;



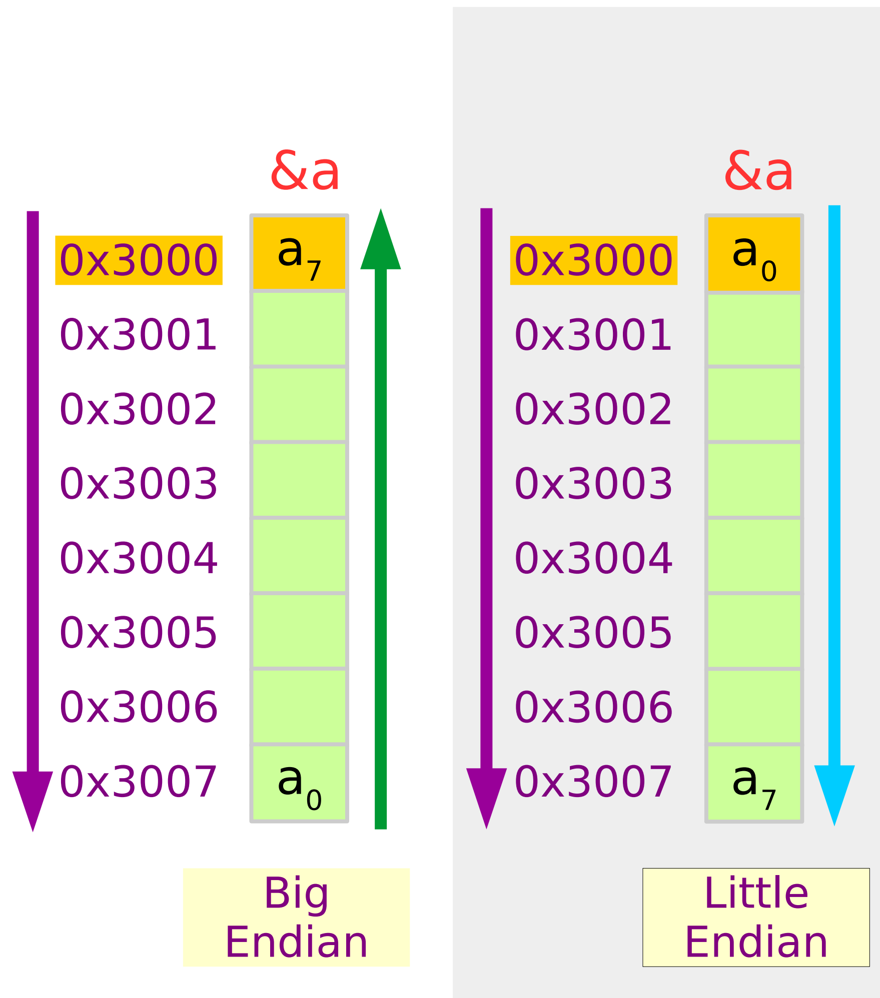
MSByte **Little** Endian LSByte

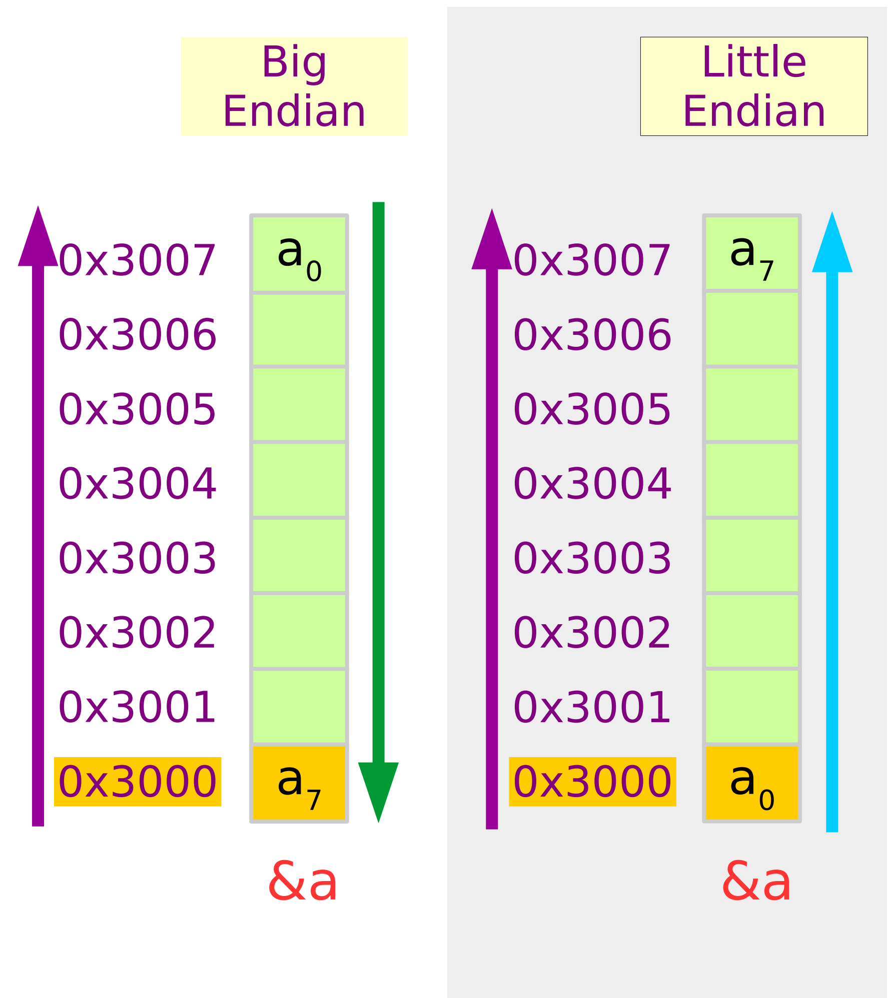
Big Endian Byte Address Example

long a;



Representations of Endianness

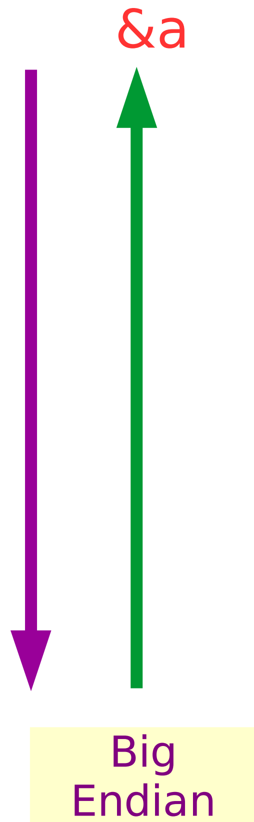




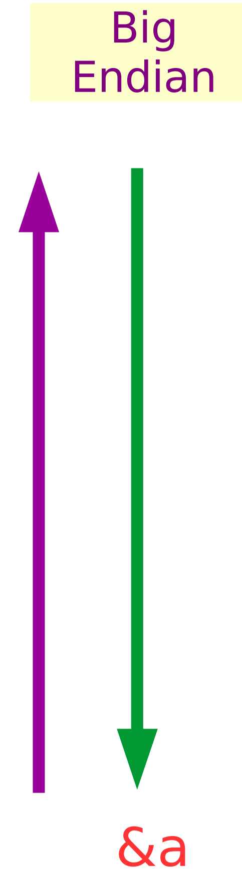
https://stackoverflow.com/questions/15620673/which-bit-is-the-address-of-an-integer

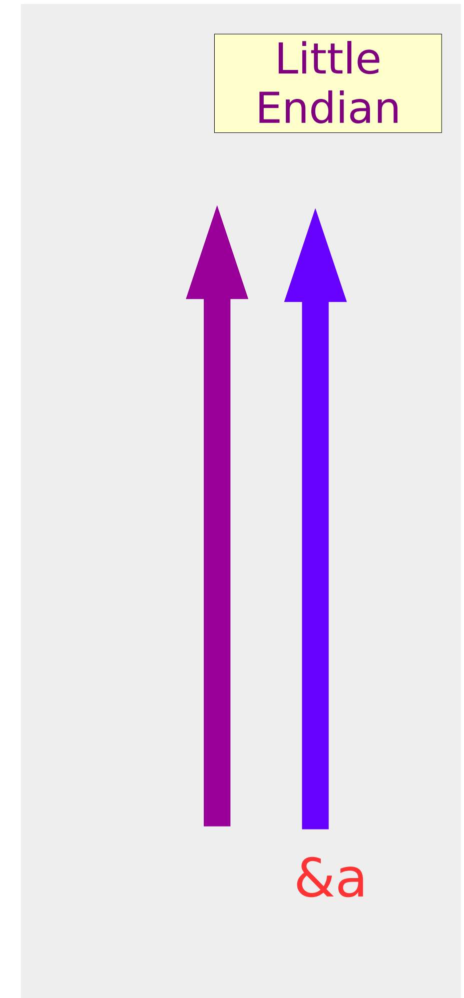
Increasing address, Increasing byte weight

downward, increasing address



upward, increasing address





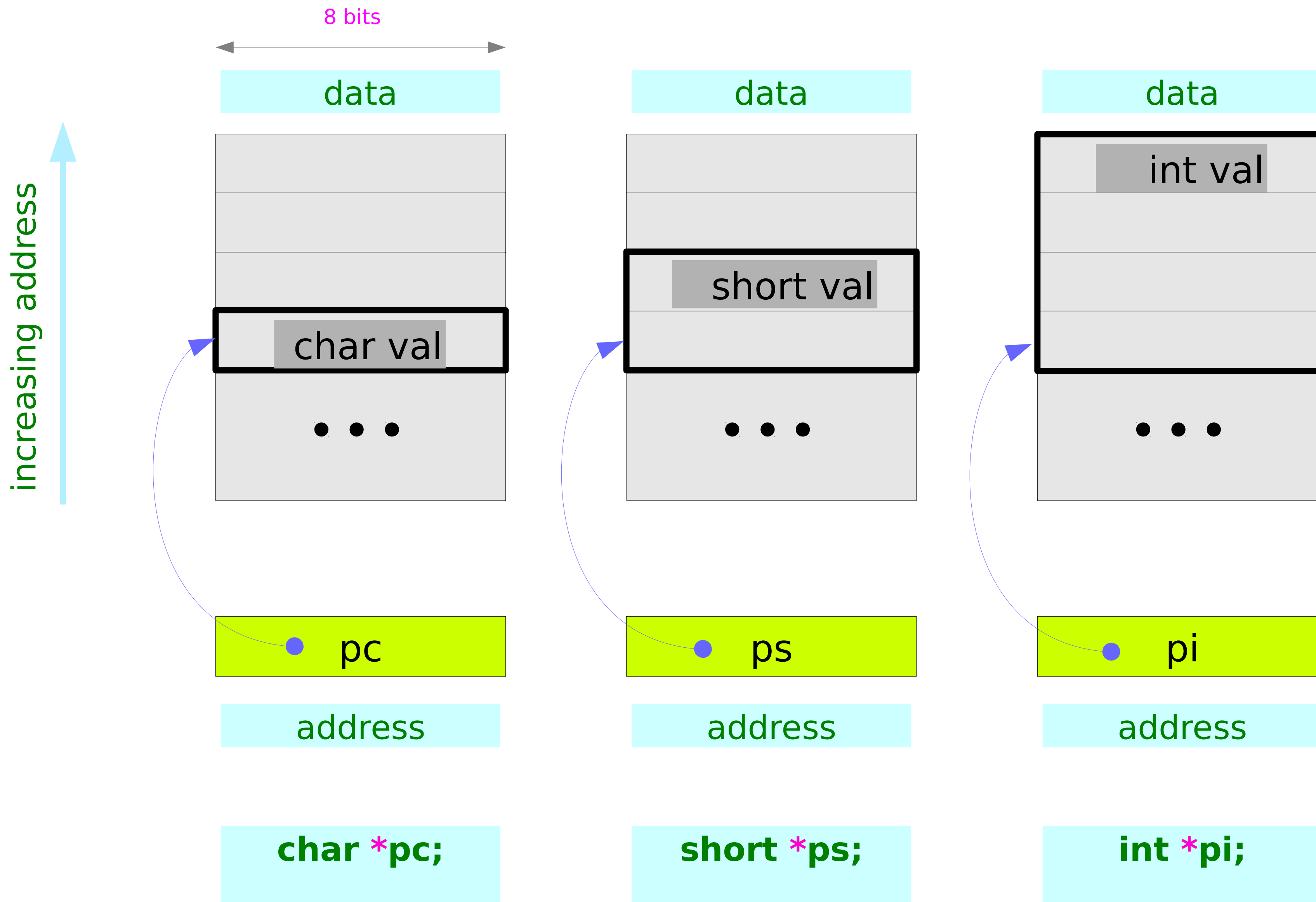
https://stackoverflow.com/questions/15620673/which-bit-is-the-address-of-an-integer

Little / Big Endian Processors

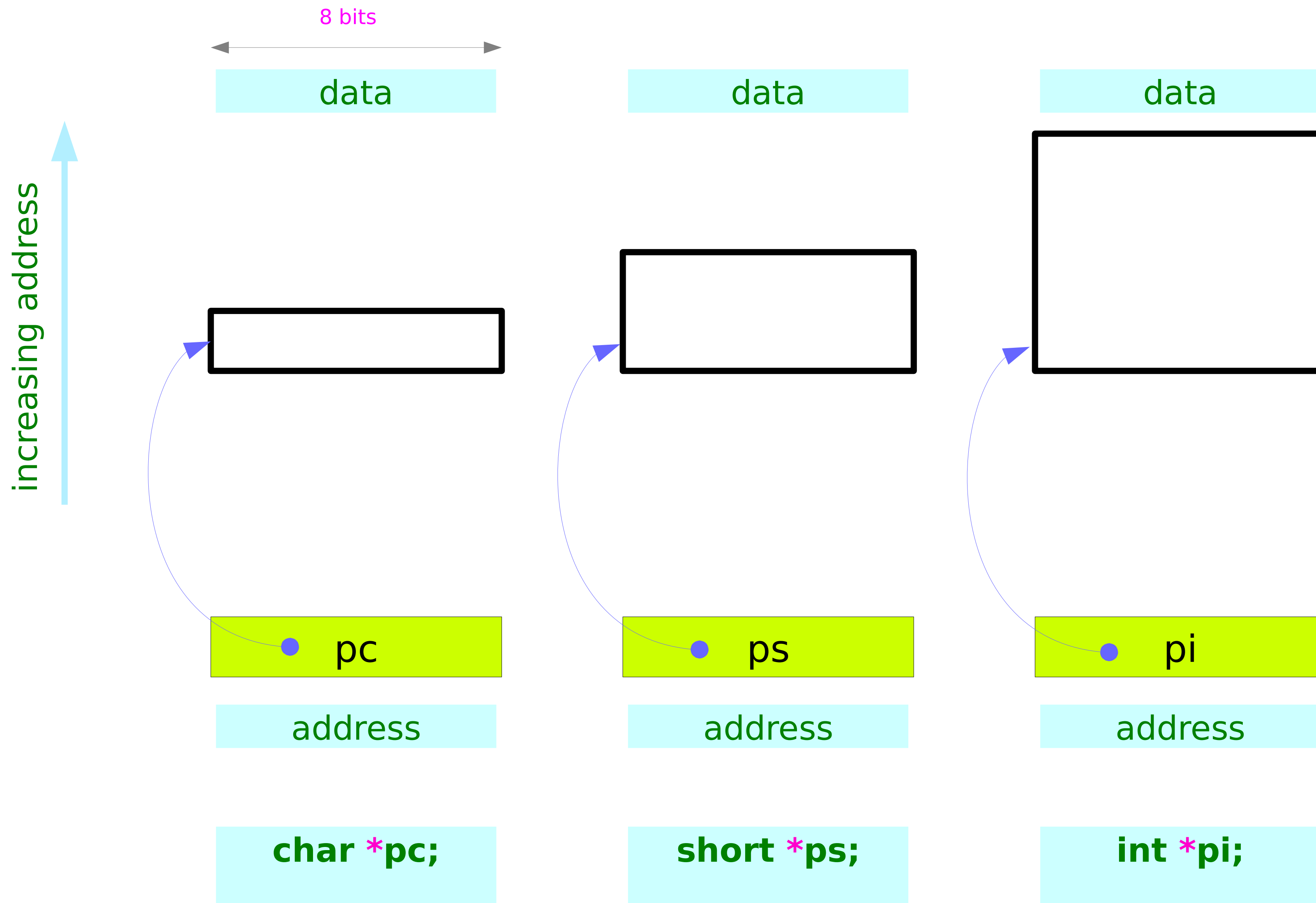
Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	(Big/Little) Endian
ARM	(Big/Little) Endian
IA-64 (64 bit)	(Big/Little) Endian
MIPS	(Big/Little)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

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

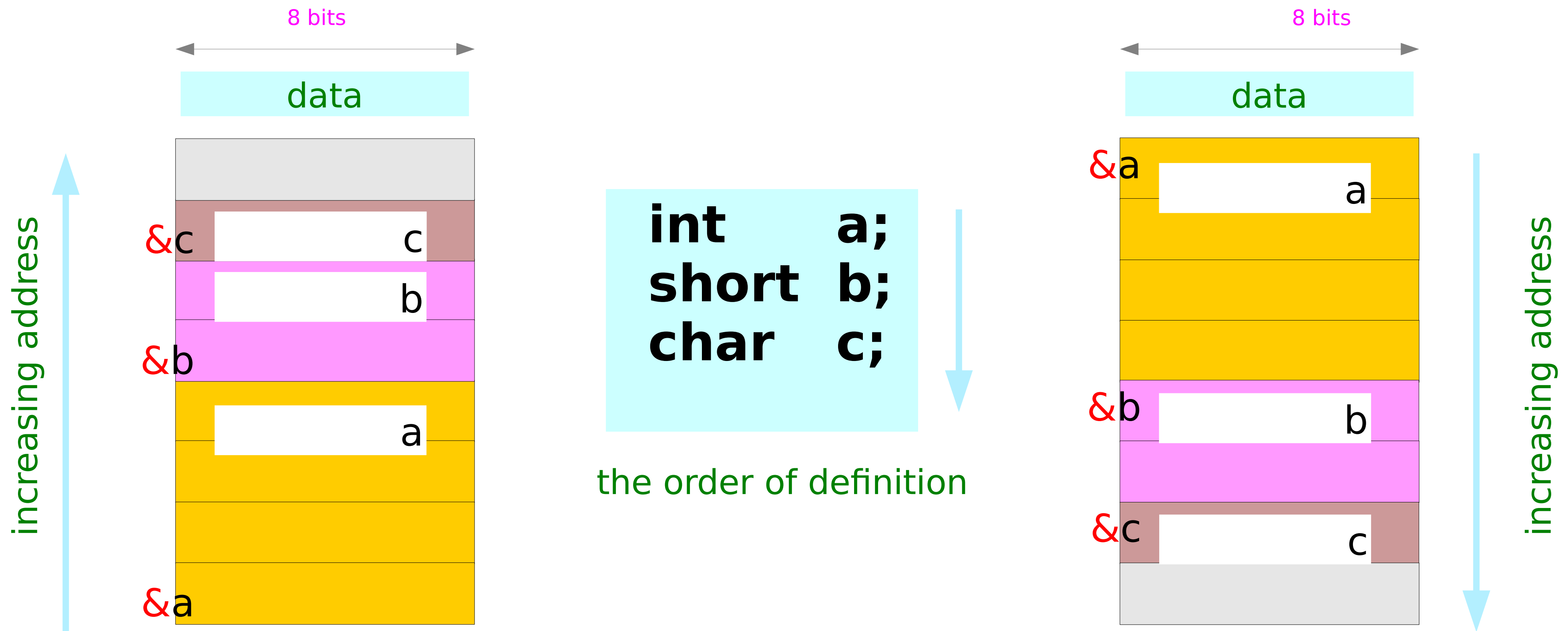
Pointer Types and Associated Data



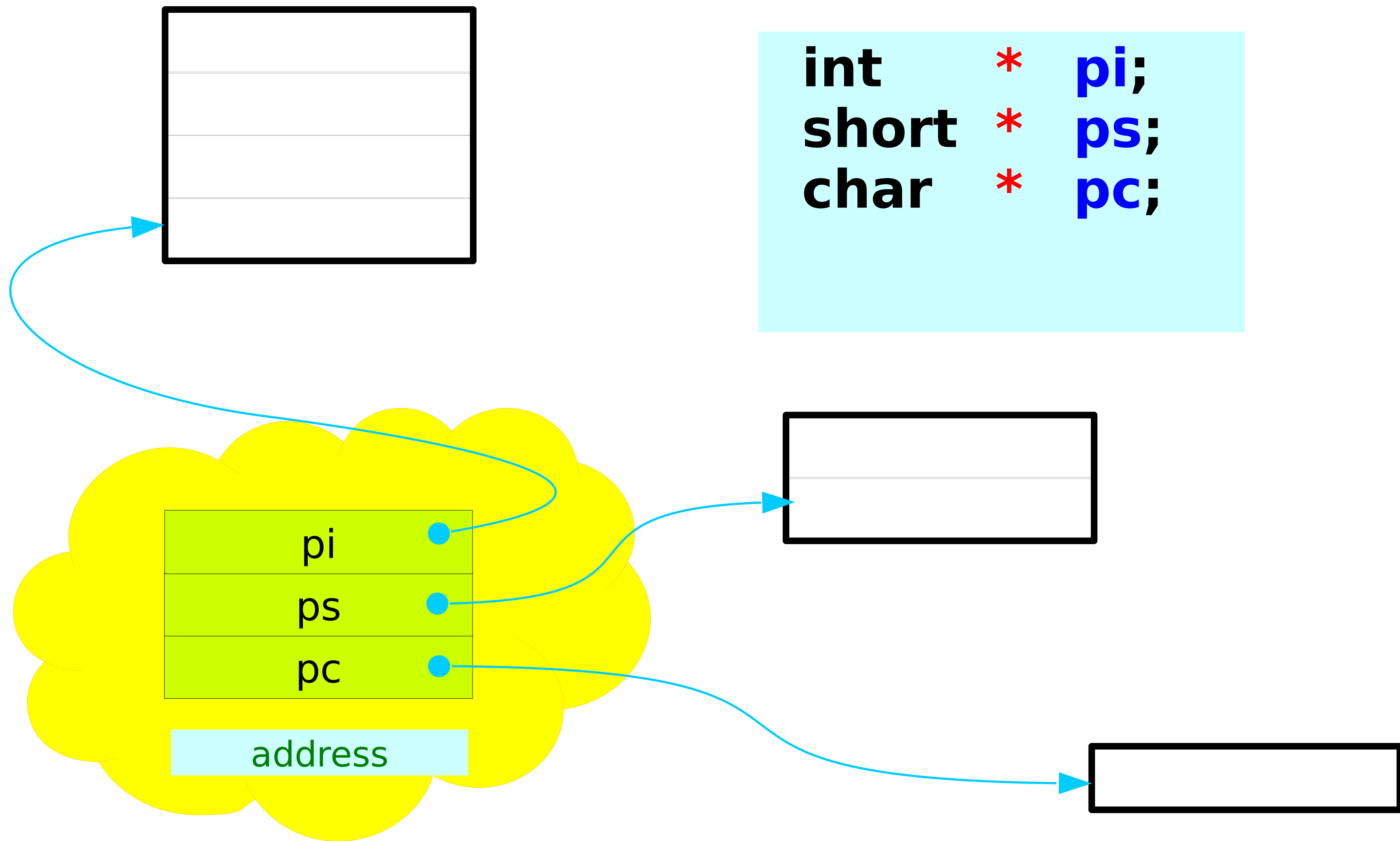
Pointer Types



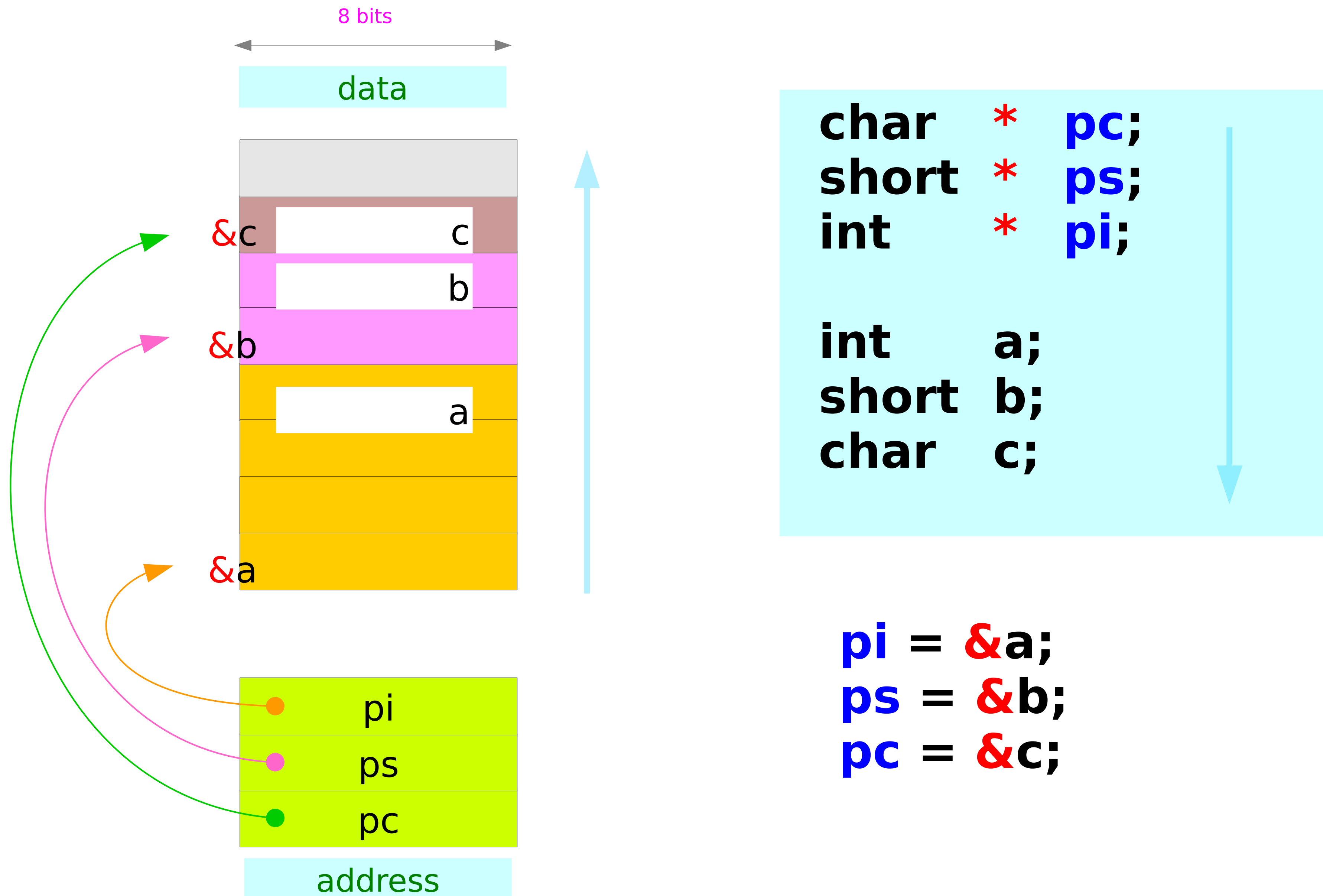
Little Endian Example



int *, short *, char * type variables



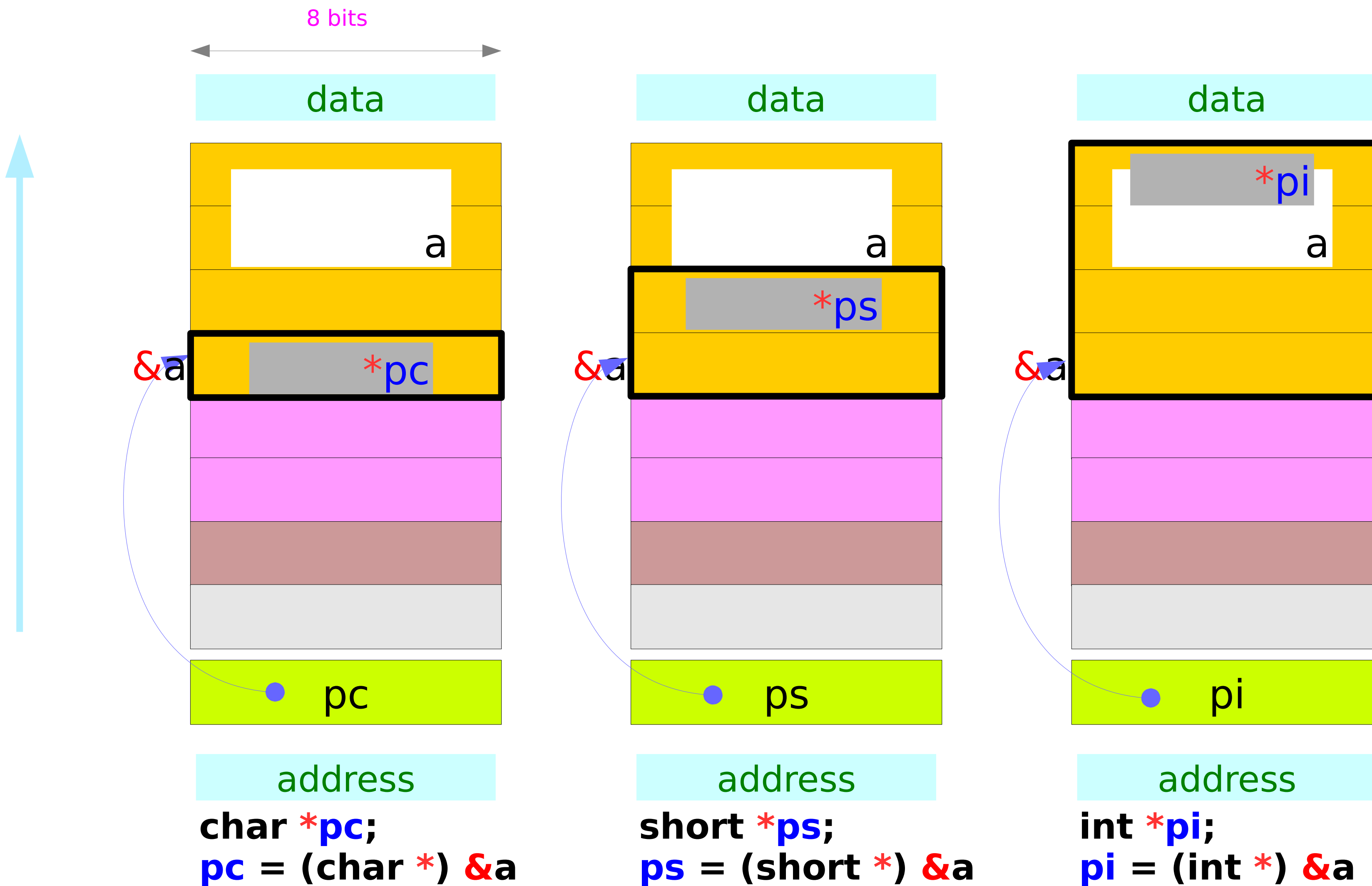
Pointer Variable Assignment




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

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

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

Pointer Type Casting

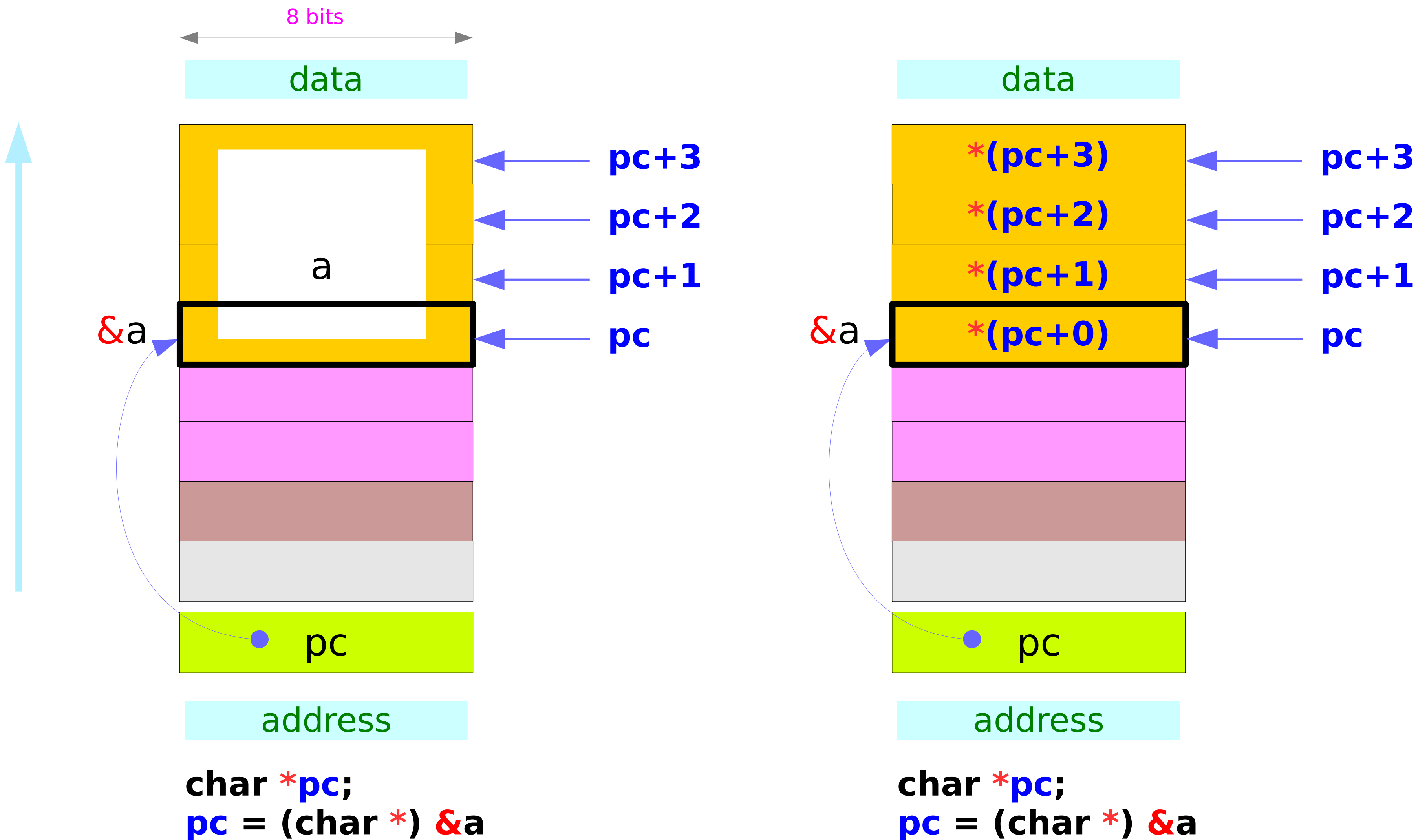


char *pc;
pc = (char *) &a

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

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

Accessing bytes of a variable



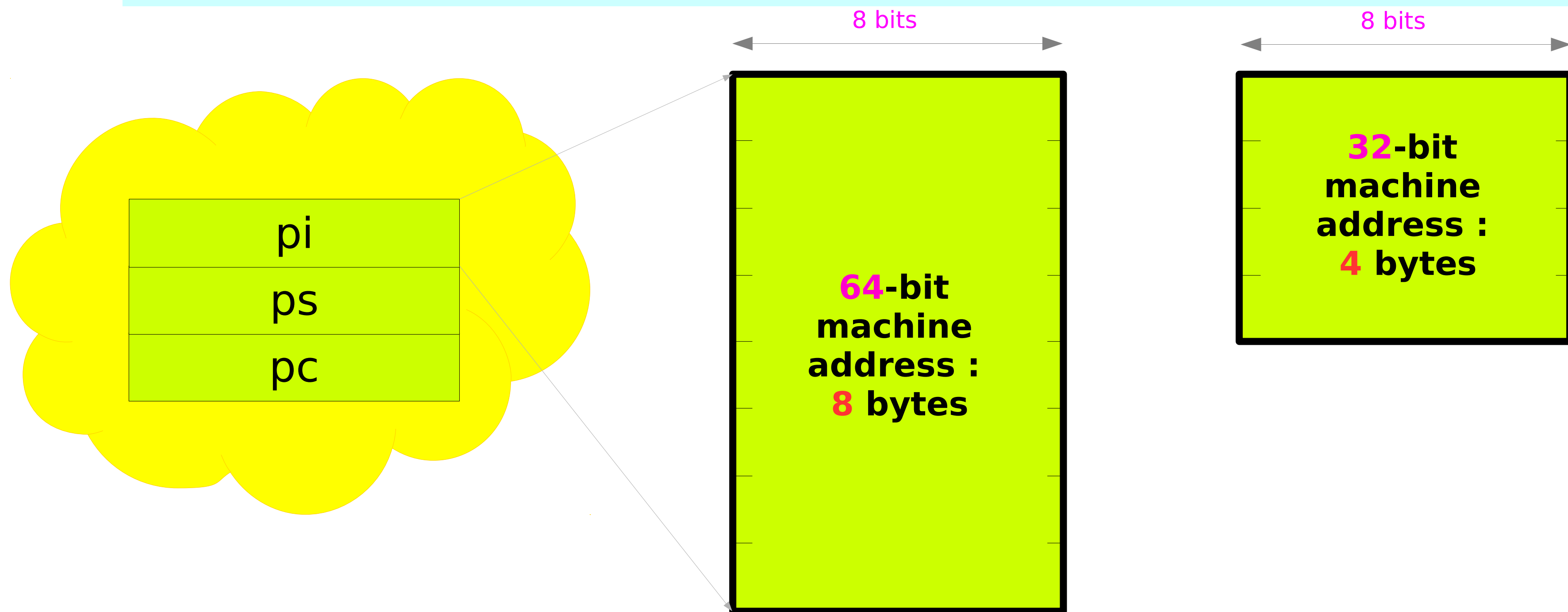

```
char *pc;
pc = (char *) &a
```

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

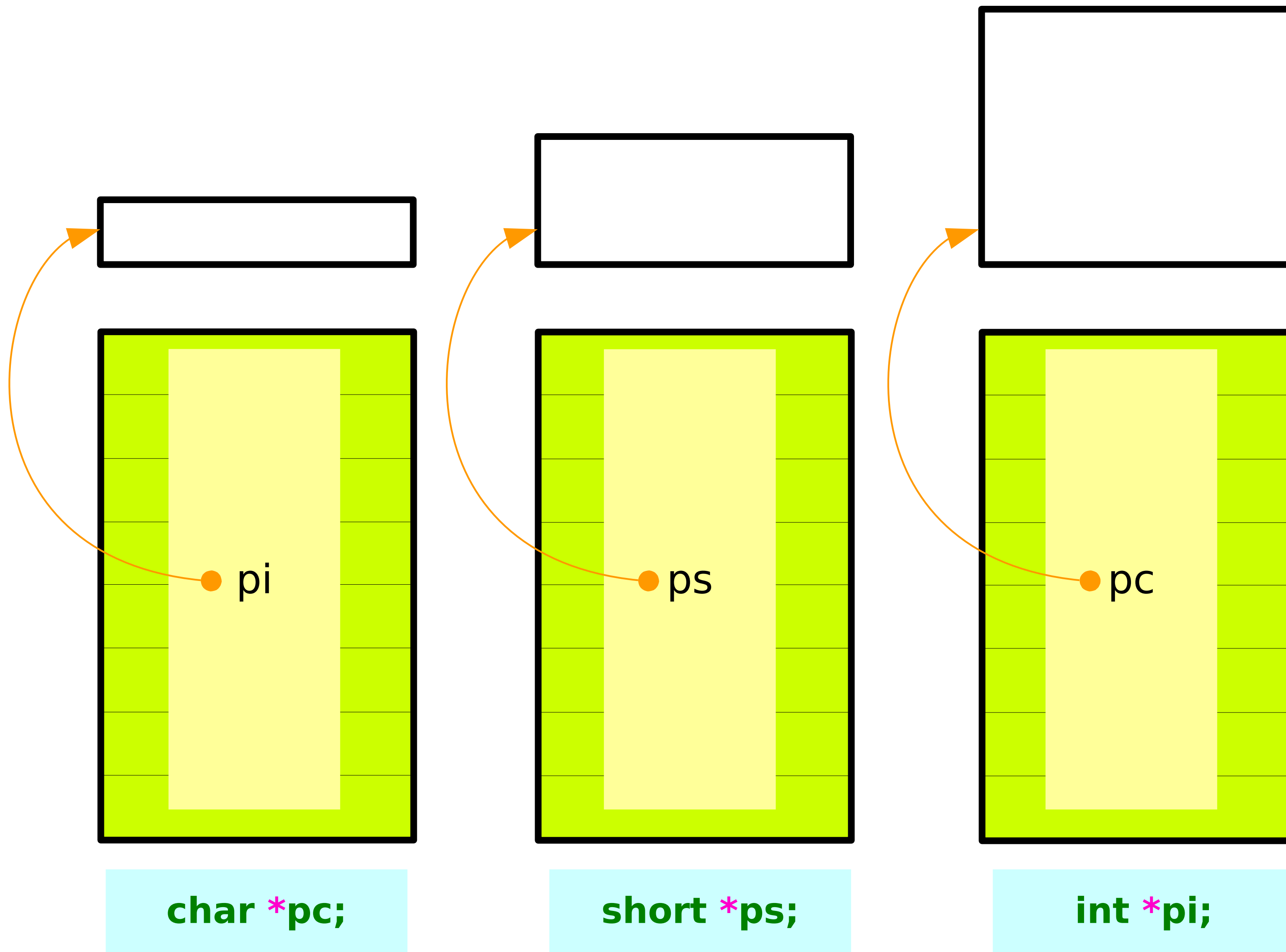
32-bit and 64-bit Address

32-bit machine : address : 4 bytes

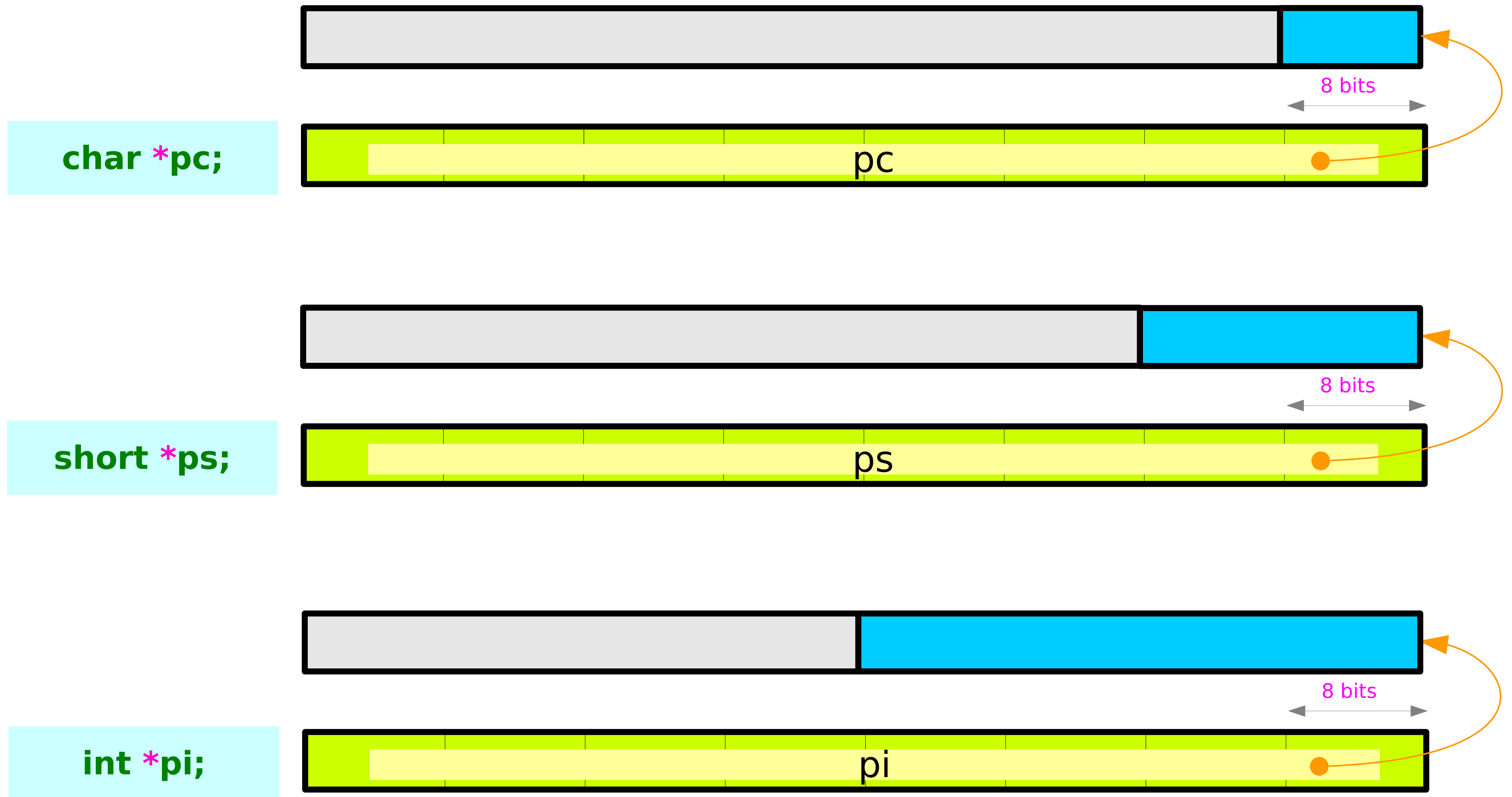
64-bit machine : address : 8 bytes



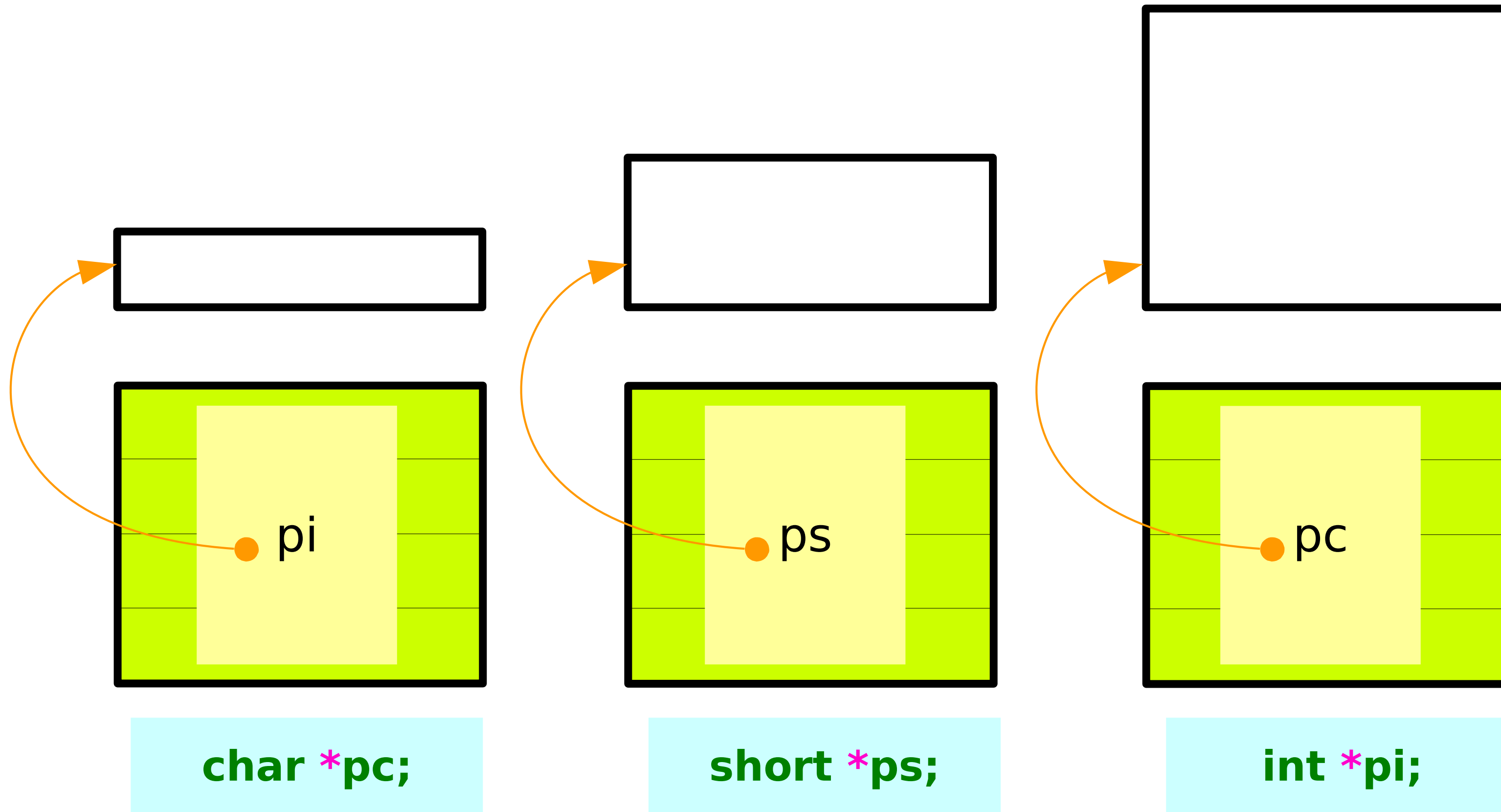
64-bit machine : 8-byte address



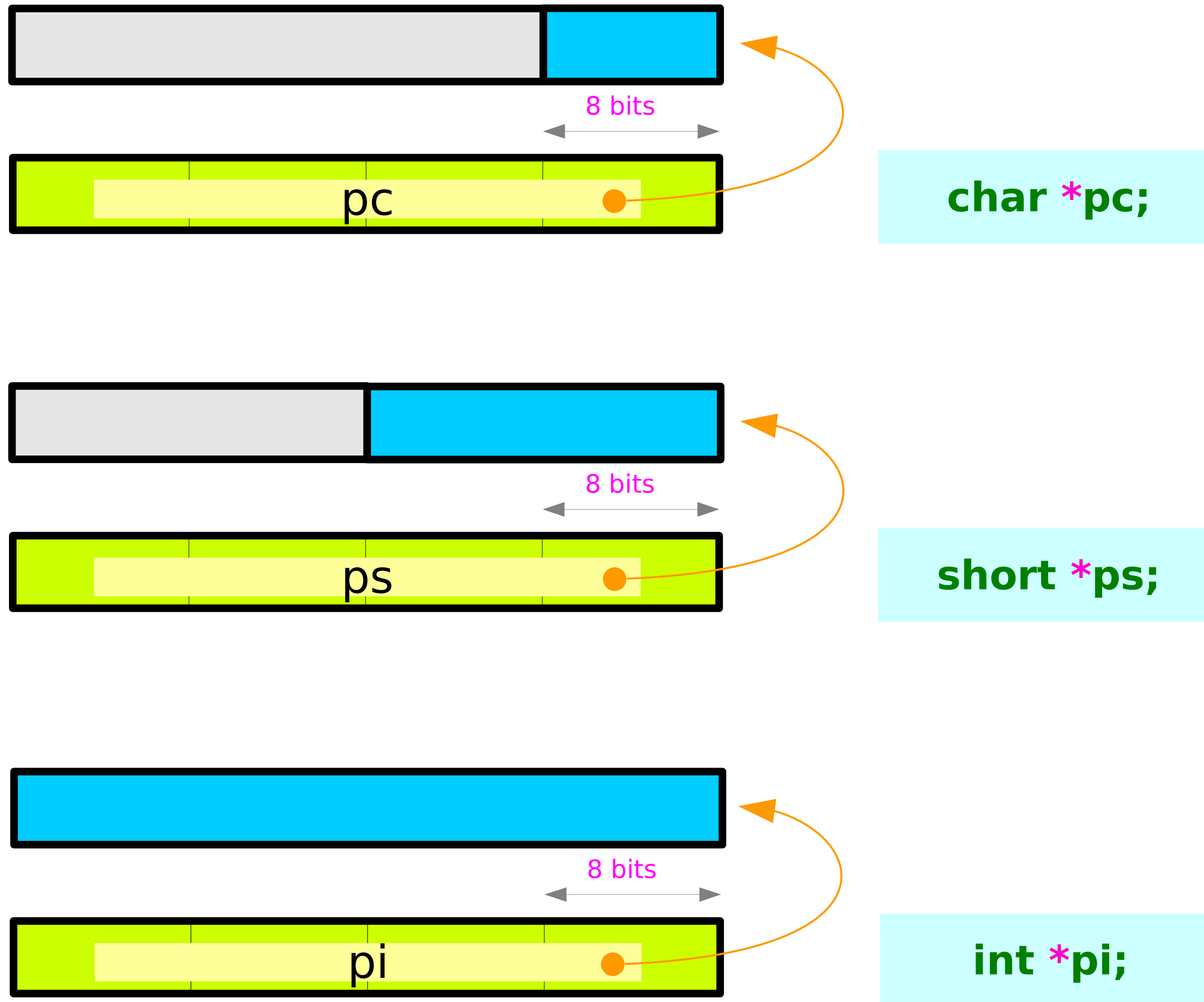
64-bit machine : 8-byte address & data buses



32-bit machine : 4-byte address



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



char *pc;

short *ps;

int *pi;

Memory Alignment (1) - allocation of variables

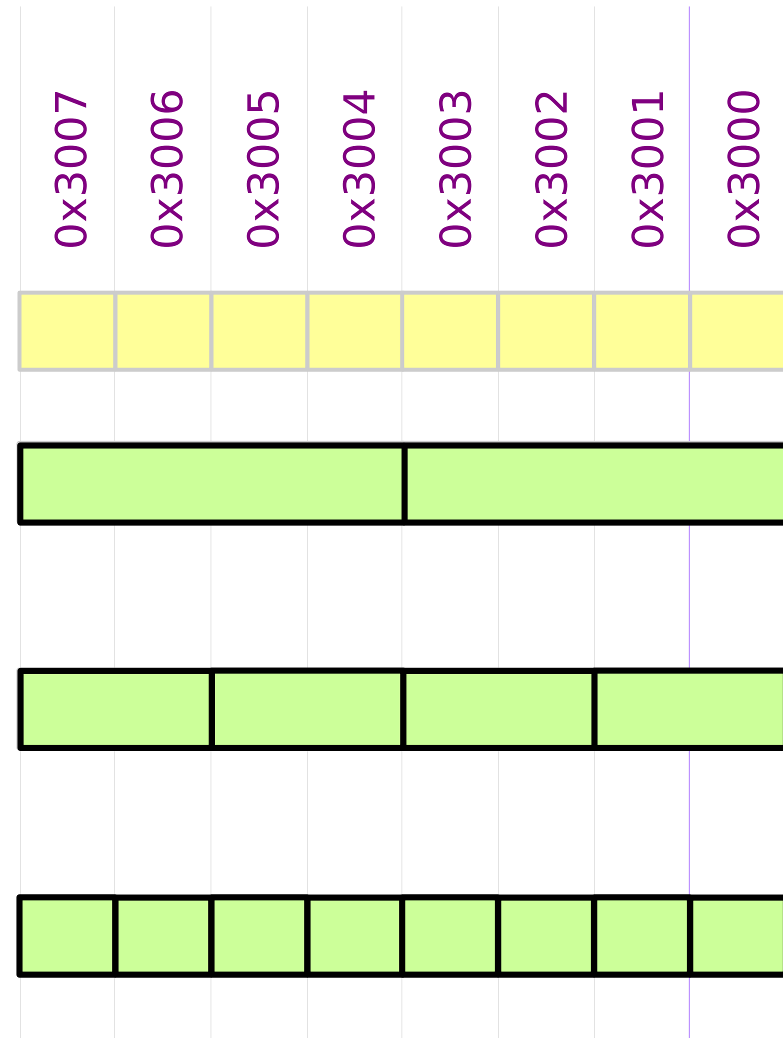
enforced by compilers

efficient memory access

int a;

short b;

char c;



Memory Alignment (2) – integer multiple addresses

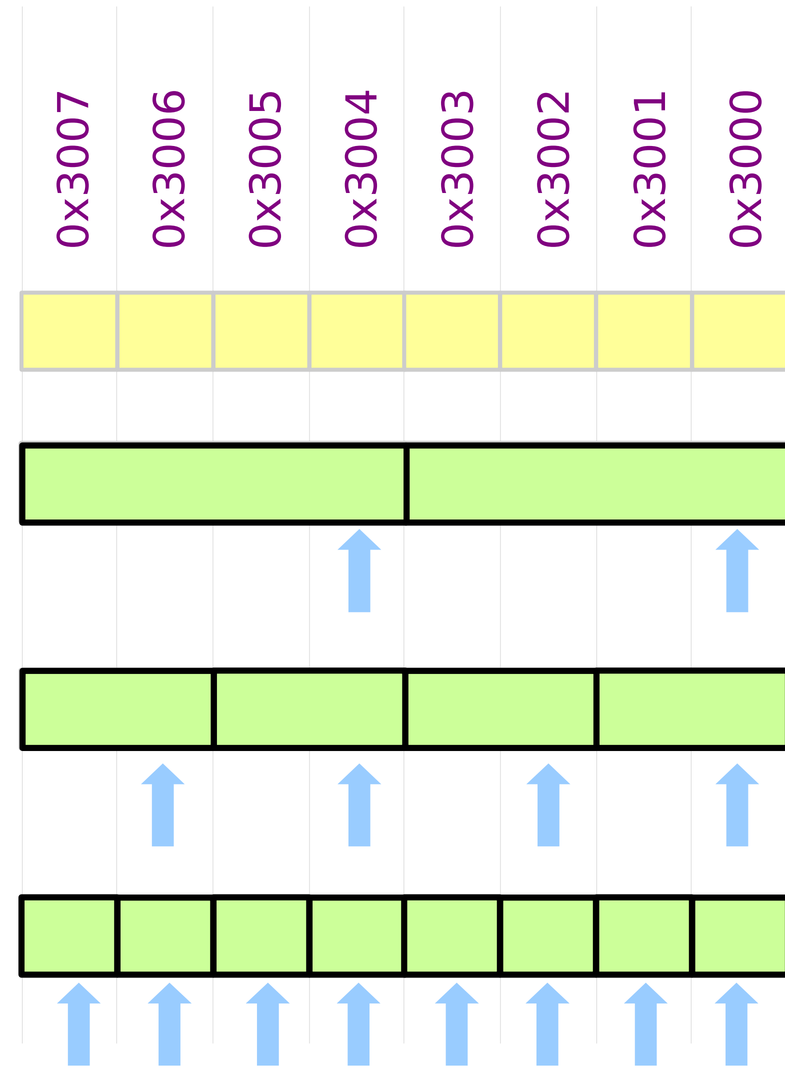
Memory Alignment:
the data **address** is a
multiple of the data **size**.

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

$integer\ \ addresses = \quad 4 \cdot k$

$short\ \ addresses = \quad 2 \cdot k$

$character\ \ addresses = \ 1 \cdot k$



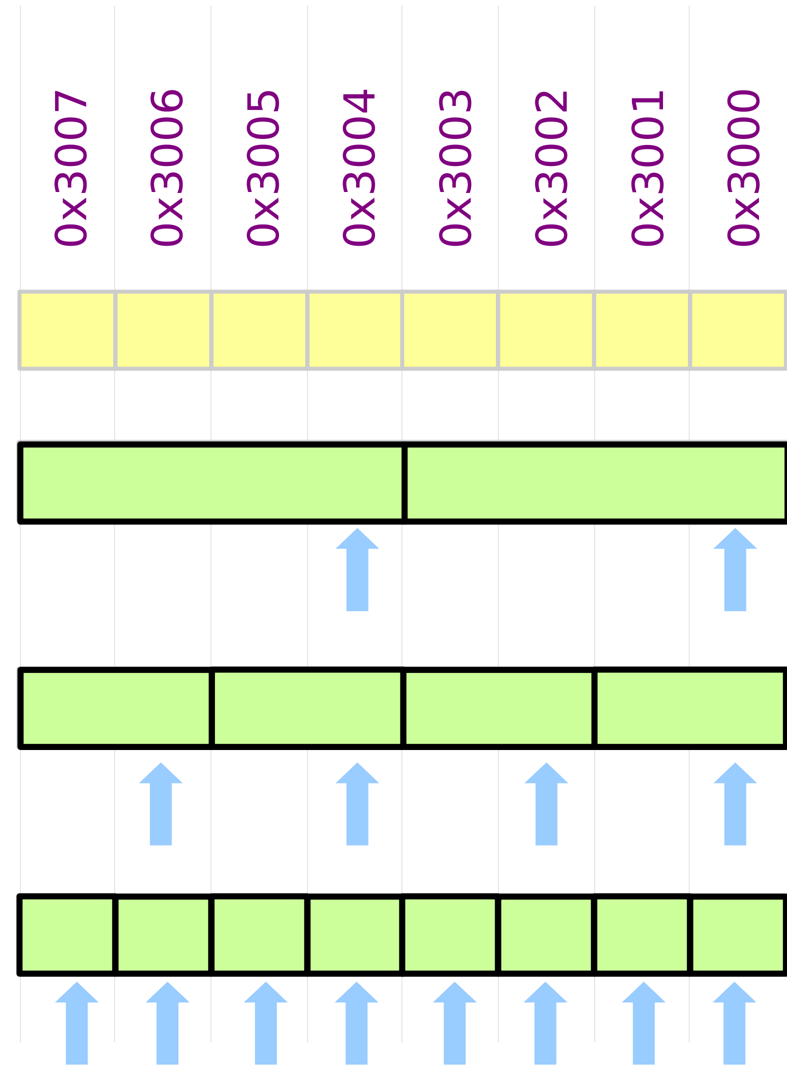
Memory Alignment (3) – pointed addresses

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

$4 \cdot k$

$2 \cdot k$

$1 \cdot k$



Memory Alignment (4) – non-pointed addresses

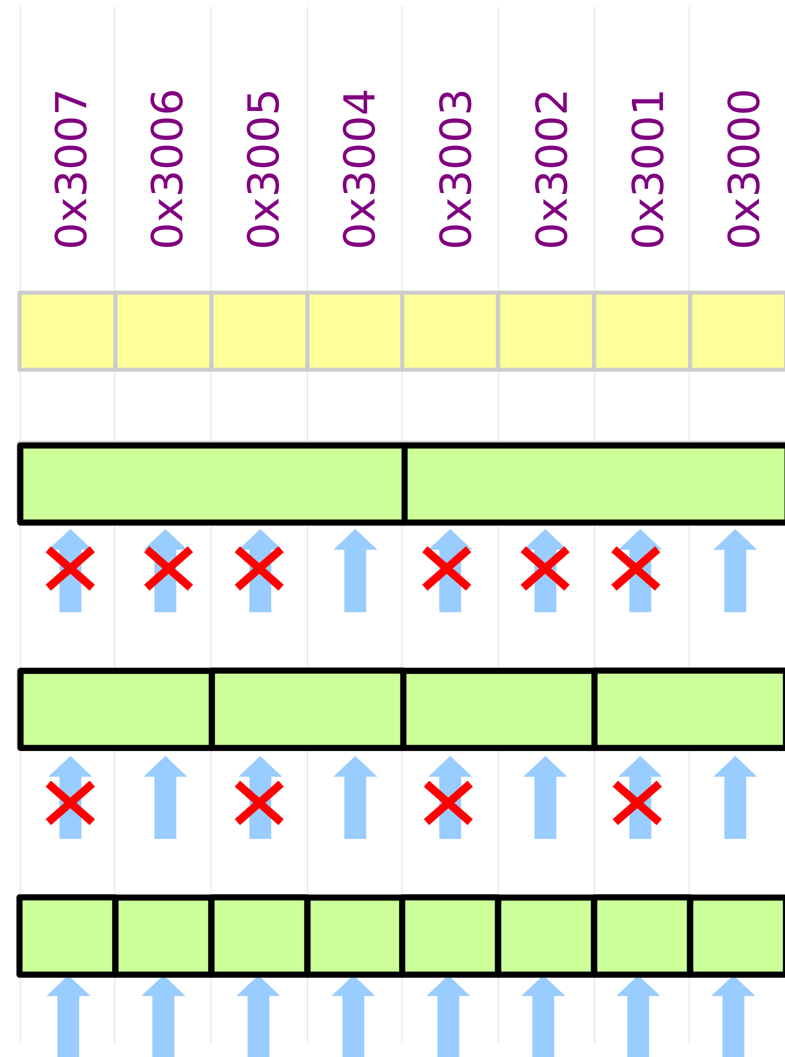
int *p;

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

short *q;

$2 \cdot k + 1$

char *r;

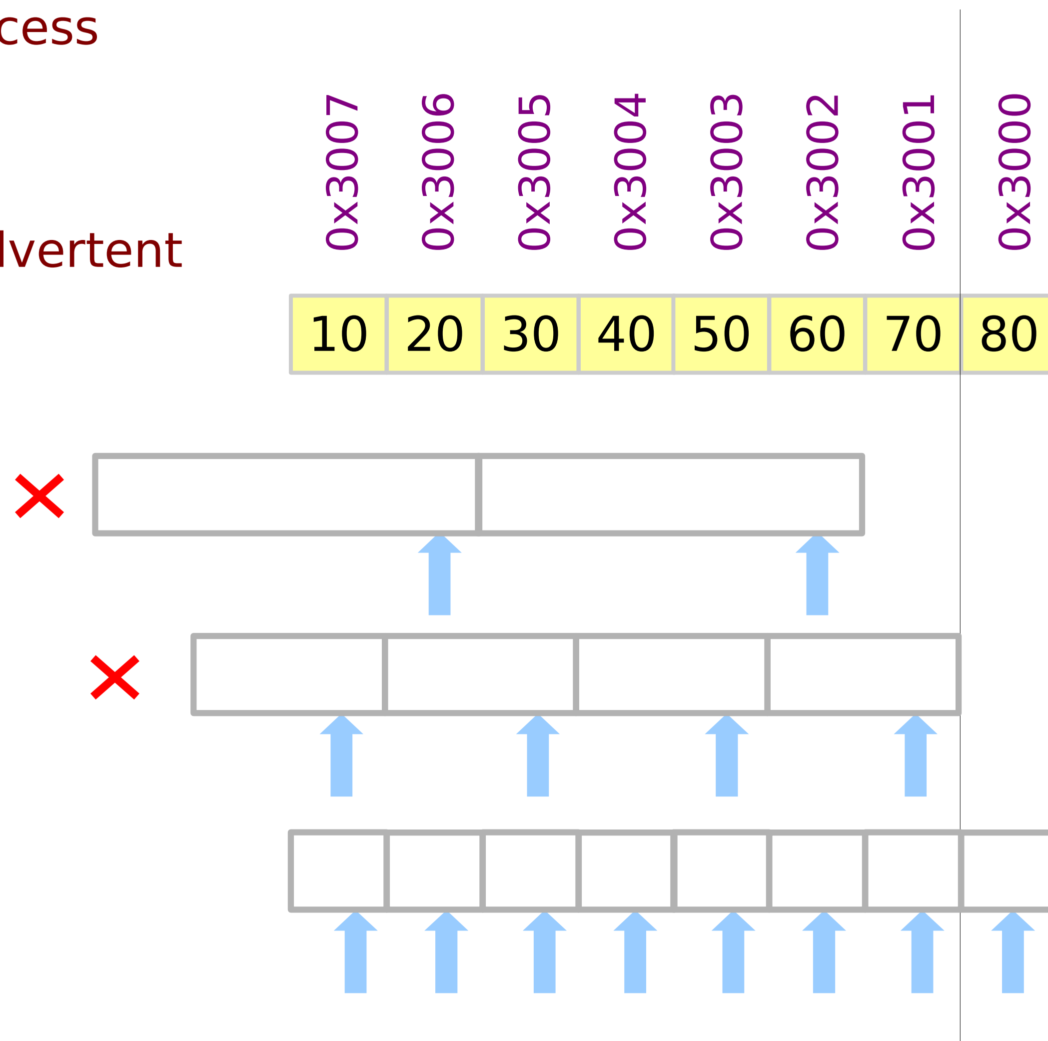


Memory Alignment (5) – broken alignment

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;
```

Unsigned Char Addition

0xFF	Signed: -1 Unsigned: 255
0xFD	Signed: -3 Unsigned: 253
0xFC	Signed: -4 Unsigned: 252

```
Signed     (-1)
Signed     (-3)
----------------
Signed     (-4)
```

```
Unsigned: 255
Unsigned: 253
          508
      -   256
----------------
Unsigned  252
```

```
Unsigned
   Wrap around
   Modulo 256
```

Converting Signed Numbers

```
Signed (op) Unsigned = Unsigned (op) Unsigned

Unsigned (op) Signed = Unsigned (op) Unsigned
```

Mixed Operation Examples

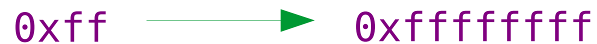
```
0xFF    Signed:   i=  -1
        Unsigned: m= 255

0xFD    Signed:   j=  -3
        Unsigned: n= 253

0xFC    Signed:   -4
        Unsigned: 252
```

```
m+n = 255+253-256   = 252
m-n = 255-253       = 2
i+j = -1-3          = -4
i-j = -1+3          = 2

m+j = 255+253-256   = 252
m-j = 255-253       = 2
i+n = 255+253-256   = 252
i-n = 255-253       = 2

(m > 0) = (255>0)   = 1
(i > 0) = (-1 > 0)  = 0
(m > n) = (255>253) = 1
(i > j) = (-1 > -3) = 1
(m < 256)=(255<0)   = 0
(i < 256)=(-1<256)  = 1
```

%u conversion (32-bit)

```
signed char m, n, p
m=%d: 15
n=%d: -1
p=%d: 14
m=%u: 15
n=%u: 4294967295
p=%u: 14
```

Promotion to 4-byte default integer
But with a sign extension




```c
#include <stdio.h>

int main(void) {
  char m, n, p;

  m = 0x0f;
  n = 0xff;
  p = m + n;

  printf("signed char m, n, p\n");
  printf("m=%%d: %d \n", m);
  printf("n=%%d: %d \n", n);
  printf("p=%%d: %d \n", p);
  printf("m=%%u: %u \n", m);
  printf("n=%%u: %u \n", n);
  printf("p=%%u: %u \n", p);
}
```

Void and Function Prototypes

void func (void);

No return value

No function parameters

The void type comprises an empty set of values; it is an incomplete object type that cannot be completed.

void *

a pointer type that doesn't specify what it points to.

void func ( ) ; accepts a constant but unknown number of arguments

void func (...) ; accepts a variable number of arguments (*not ISO C*)

void (* x) (); pointer to a function returning no result

void * x (); function returning pointer to void

Ignoring Return Value

```
int func (void)
```

```
(void) func (void)
```

(void) type casting
to ignore the return int value

Void Pointer

void * universal data pointer

- a pointer type that doesn't specify what it points to.
- can store an address to any non-function data type
- implicitly converted to any other pointer type on assignment
- must use an explicit cast if dereferenced inline.

Dereferencing Void Pointers

```
#include <stdio.h>

void fint   (void *a) { printf("%d\n",* (int *)    a ); }
void fchar (void *a) { printf("%c\n", * (char *) a ); }
void ffloat (void *a) { printf("%f\n", * (float *)  a ); }

void main(void) {
  int       a = 100;
  char      b = 'B';
  float     c = 3.14;

  fint      (&a);
  fchar     (&b);
  ffloat    (&c);
}
```

dereferencing the void pointer without type-casting
not possible.

void indicates the absence of type
cannot dereference or assign to.

```
void fint   (void *a) { printf("%d\n", (int)    *a ); }
void fchar (void *a) { printf("%c\n",  (char) *a ); }
void ffloat (void *a) { printf("%f\n",  (float) *a ); }
```

Pointer Arithmetic and Void Pointers

```
#include <stdio.h>

void func (void *a) {
   int *p = a;

      printf("%d\n", * p++);
      printf("%d\n", * p++);
      printf("%d\n", * p++);
      printf("%d\n", * p++);
      printf("%d\n", * p++);
}

void main(void) {
  int   a[5] = {10, 20, 30, 40, 50};

  func (a);

}
```

```
void func (void *a) {


      printf("%d\n", * (int *) a++);
      printf("%d\n", * (int *) a++);
      printf("%d\n", * (int *) a++);
      printf("%d\n", * (int *) a++);
      printf("%d\n", * (int *) a++);
}
```

Pointer arithmetic is not possible
on pointers of void

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
[5] http://www.stackoverflow.com