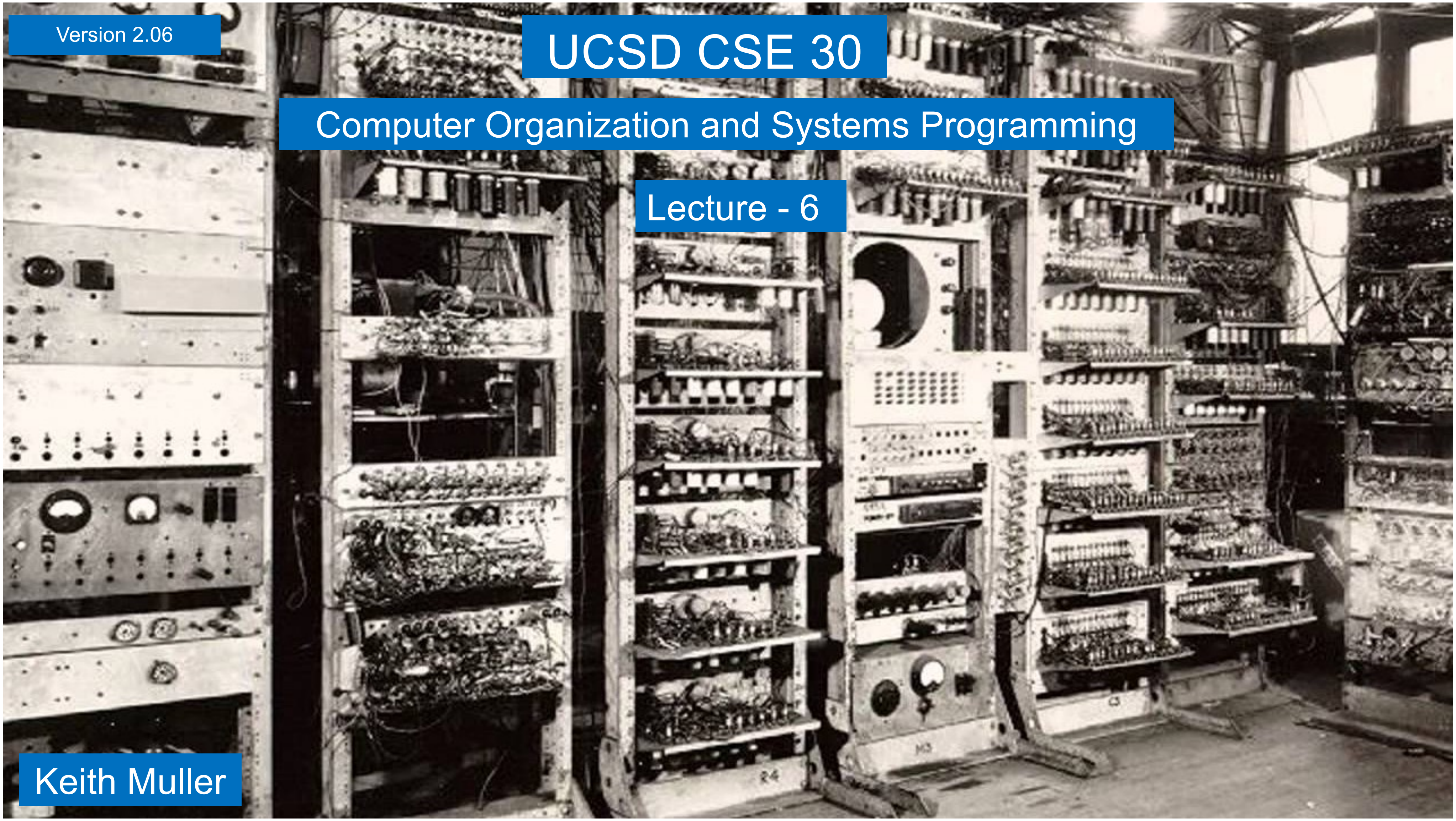
Version 2.06

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 6



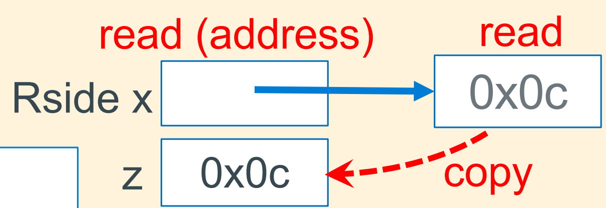
Keith Muller

Rside Indirection (or dereference) Operator: *

- Performs the following steps when the * is on the Rside:

1. read the contents of the variable to get an address
2. **read** and return the contents at that address
   - (requires two reads of memory on the Rside)

```
z = *x; // copy the contents of memory pointed at by x to z
```



Two reads here
(1) read to get an address
(2) read the address to get the value

x

Rside Indirection (or dereference) Operator: *

*Contents of **p** is the address of **i***

(p points at i)

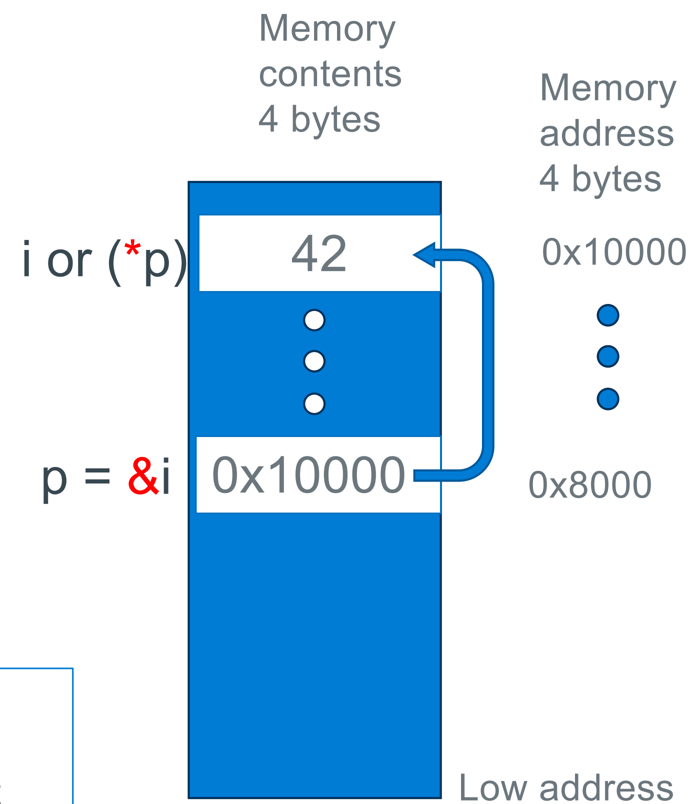
```
int i = 42;
int *p;

p = &i;
```

No reads here

```
printf("*p is %d\n", *p);
```

```
% ./a.out
*p is 42
```

Two reads here
(1) read to get an address
(2) read the address to get the value



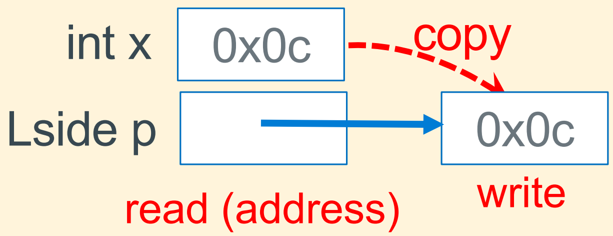
x

Lside Indirection Operator

Performs the following steps when the * is on the Lside:

1. read the contents of the variable to get an address
2. **write** the evaluation of the Rside expression to that address
   - (requires one read of memory and one write of memory on the Lside)

```
*p = x; // copy the value of x to the memory pointed at by p
```



x

Lside Indirection (or dereference) Operator: *

*Contents of **p** is the address of **i***

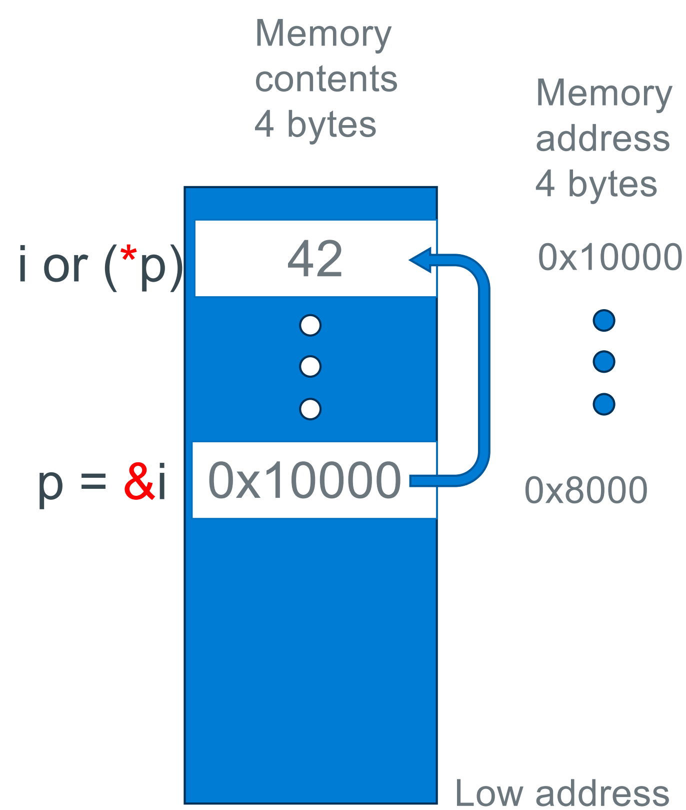
(p points at i)

No reads here
one write

One read here
(1) read to get
an address
(2) one write

```
int i;
int *p;

p = &i;
*p = 42;

printf("*p is %d\n", *p);
```

```
% ./a.out
*p is 42
```

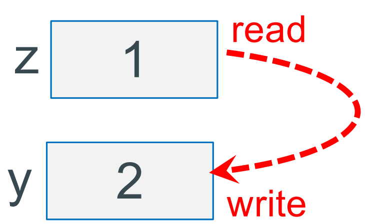


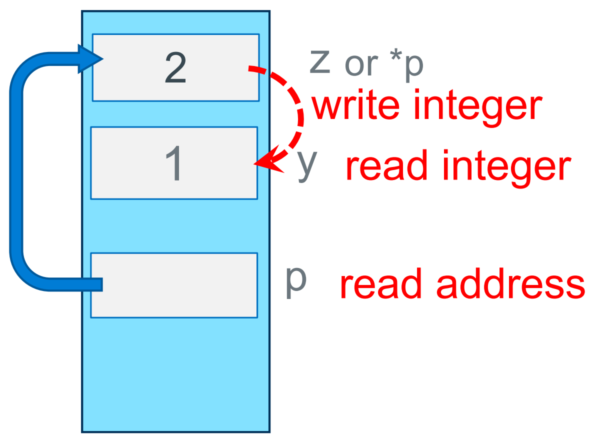
x

Each use of a * operator results in one additional read: Rside

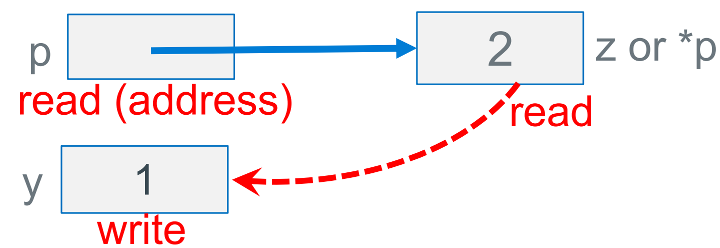
RULE: Each * when used as a dereference operator in a statement (either Lside or Rside) it causes an <u>additional</u> read to be performed

```
int z = 2;
int y = 1;

y = z; // one read
```






```
int z = 2;
int y = 1;
int *p;

p = &z;
y = *p; // two reads on rside
```

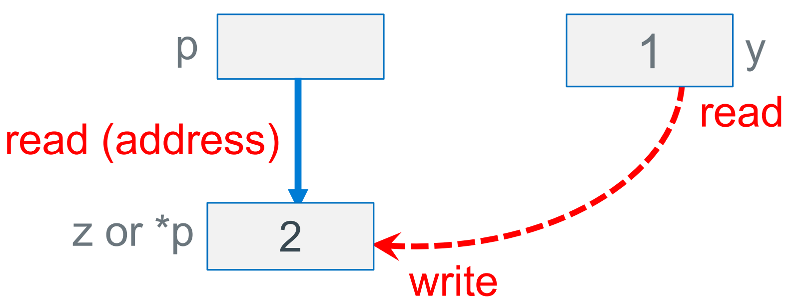


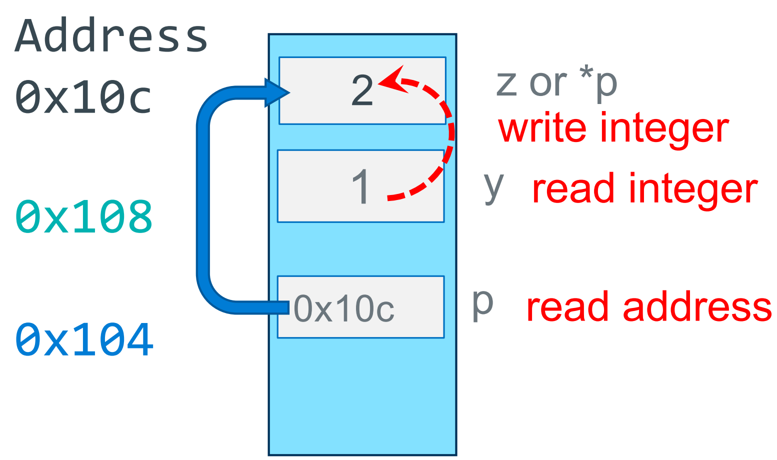
Aside: `y = *(&z); // same as y = z`

x

Each use of a * operator results in one additional read: Lside

```
int z = 2;
int y = 1;
int *p;

p = &z;
*p = y;     // one read on lside
```

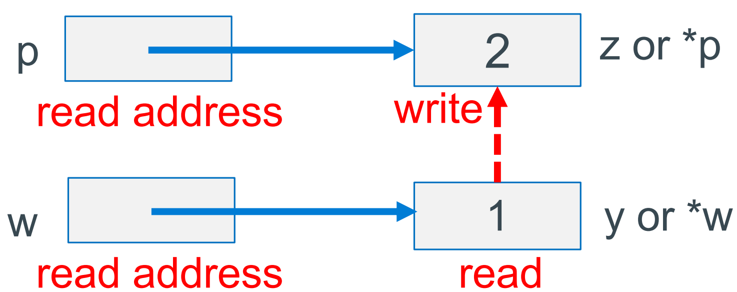


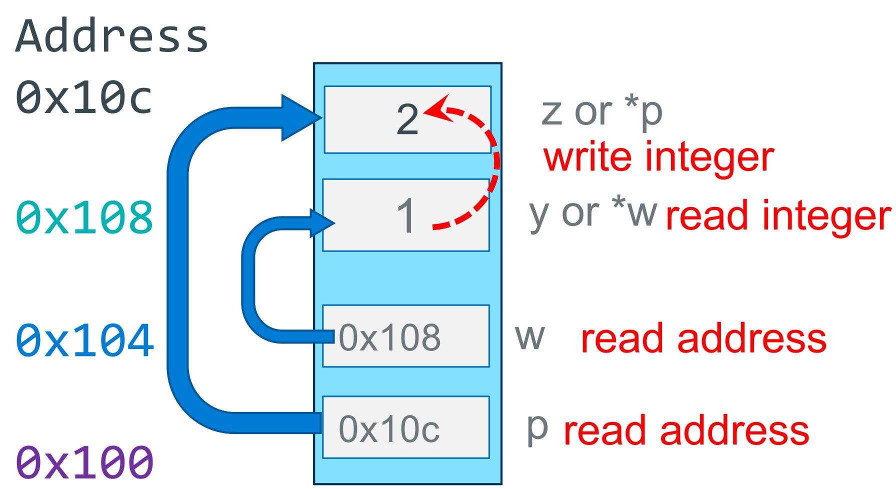


x

Each use of a * operator results in one additional read : both sides

```
int z = 2;
int y = 1;
int *w;
int *p;

p = &z;
w = &y;
*p = *w;
```



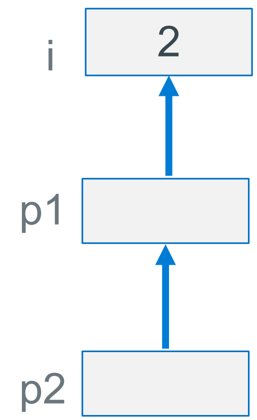


x

Pointer to Pointers (Double Indirection)

- Define a pointer to a pointer (p2 below)

```
int i = 2;
int *p1;
int **p2; // pointer to a pointer to an int

p1 = &i;
p2 = &p1;
printf("%d\n", (**p2) * (**p2));
```



- C allows any number of pointer indirections
  - more than two levels is very uncommon in real applications as it reduces readability and generates at lot of memory reads
- **RULE (important):** number of * **in the variable definition** tells you **how many reads it takes to get to the base type**

  #reads to base type = number of * (in the definition) + 1

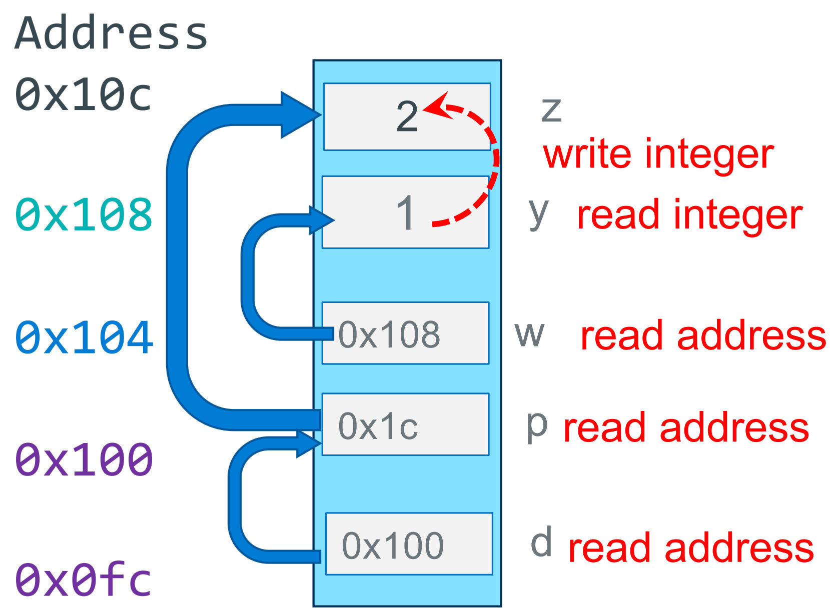
- Example:

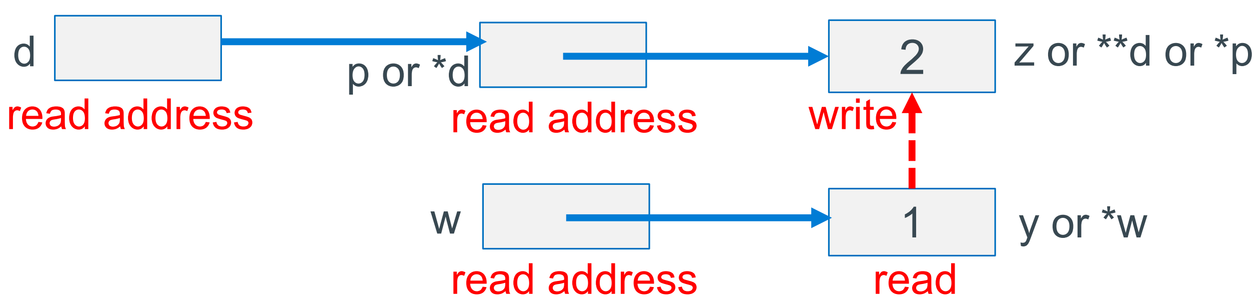
  `int **p2;      //` requires 3 reads to get to the int

x

Double Indirection: Lside

```
int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;
**d = *w;
```

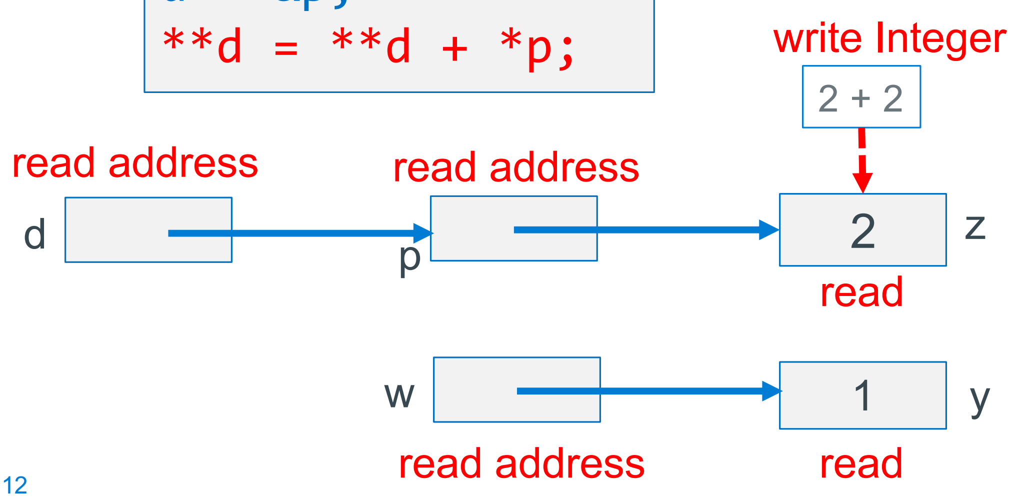


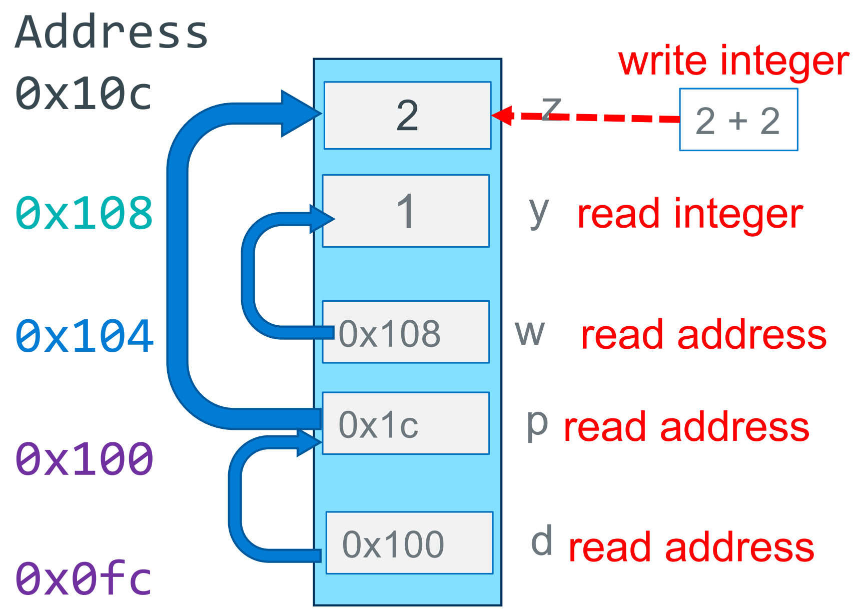


x

Double Indirection: Rside

```
int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;
**d = **d + *p;
```





Important Observation

**d on Lside is two reads

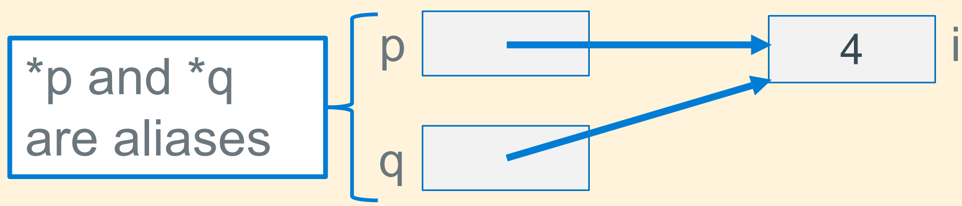
**d on Rside is three reads

x

What is Aliasing?

- Two or more variables are aliases of each other when they all reference the same memory (so different names, same memory location)
- **Example**: When one pointer is copied to another pointer it *creates an **alias***
- ***Side effect***: Changing one variables value (content) changes the value for other variables
  - Multiple variables all read and write the **same** memory location
  - Aliases occur either by accident (coding errors) or deliberate (careful: readability)

```
int i = 5;
int *p;
int *q;

p = &i;
q = p;    // *p & *q now aliases
*q = 4;   // changes i and *p
```



Result *p, *q and i all have the value of 4

x

Defining Arrays

Definition: `type name[count]`

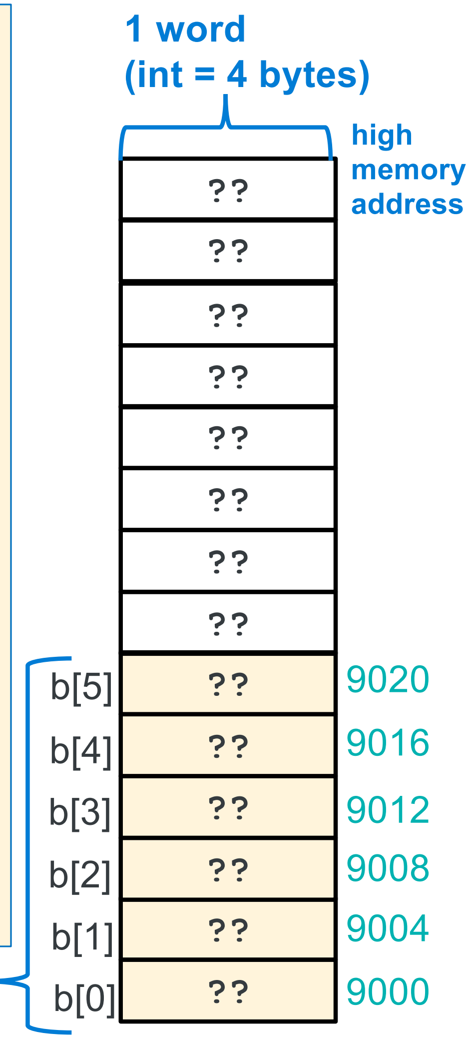
- ***"Compound"*** data type where each value in an array is an element of type
- Allocates **name** with a *fixed* count array elements of type **type**
- Allocates (`count * sizeof(type)`) bytes of ***contiguous memory***
- Common usage is to specify a compile-time constant for **`count`**

```
#define BSZ   6
int b[BSZ];
```

BSZ is a macro replaced by the C preprocessor

- Array **names are constants** and cannot be assigned (the name cannot appear on the Lside by itself)

```
int a [BSZ];
a = b;        // invalid does not copy the array
              // must copy arrays element by element
```




```
int b[6];
```


x

Array Initialization

- Initialization: `type name[count] = {val0,…,valN};`
  - **{ }** *(optional)* initialization list can *only* be used at **time** of **definition**
  - If no `count` supplied, `count` is determined by compiler using the number of array initializers

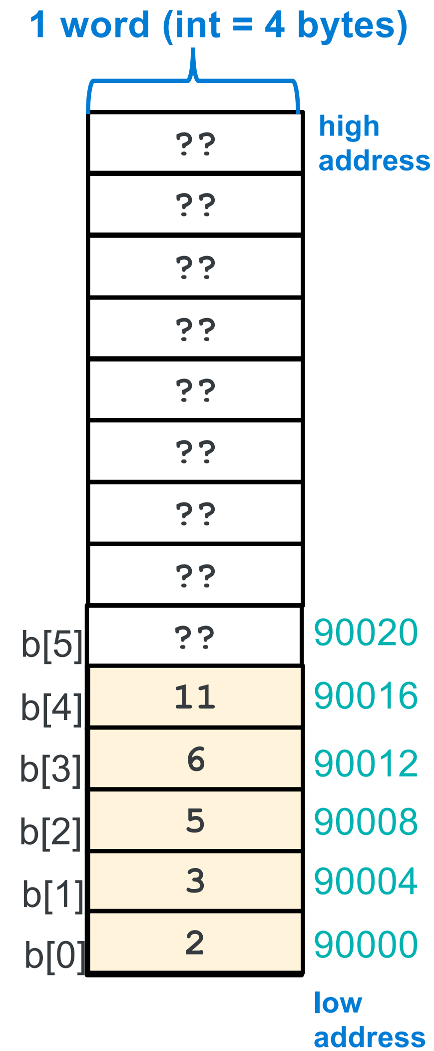
no initialization values given; then elements are initialized to 0

  - `int block[20] = {}; //only works with constant size arrays`
    - defines an **array of 20 integers** each element filled with zeros
    - Performance comment: do not zero automatic arrays unless really needed!
  - When a **count** is given:
    - **extra** *initialization values* are **ignored**
    - **missing** *initialization values* are set to **zero**

```
int block[5] = {2, 3, 5, 6, 11, 13};
```

not needed and if used **may** truncate initialization list

6 initialization values given, **only 5 are used**



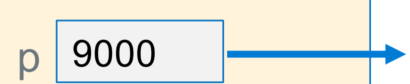
Accessing Arrays Using Indexing

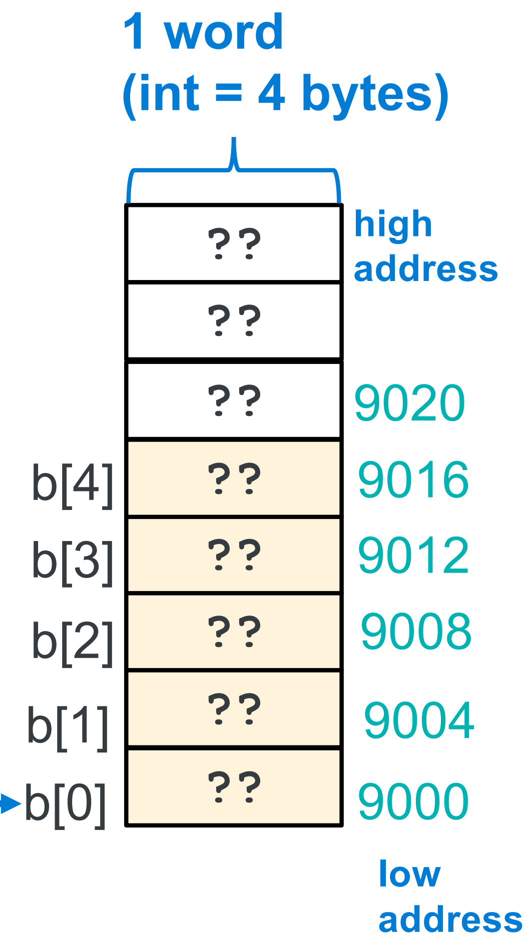
- `name[index]` selects the `index` element of the array
  - index **should be** unsigned
  - Elements range from: 0 to count – 1 (int x[count];)
- `name[index]` can be used as an assignment target or as a value in an expression

```
int a[2] = {1, 2};
a[0] = a[1];
```

- Array name (by itself with no [ ]) on the Rside evaluates to the address of the first element of the array

```
int b[5];
int *p = b;
```





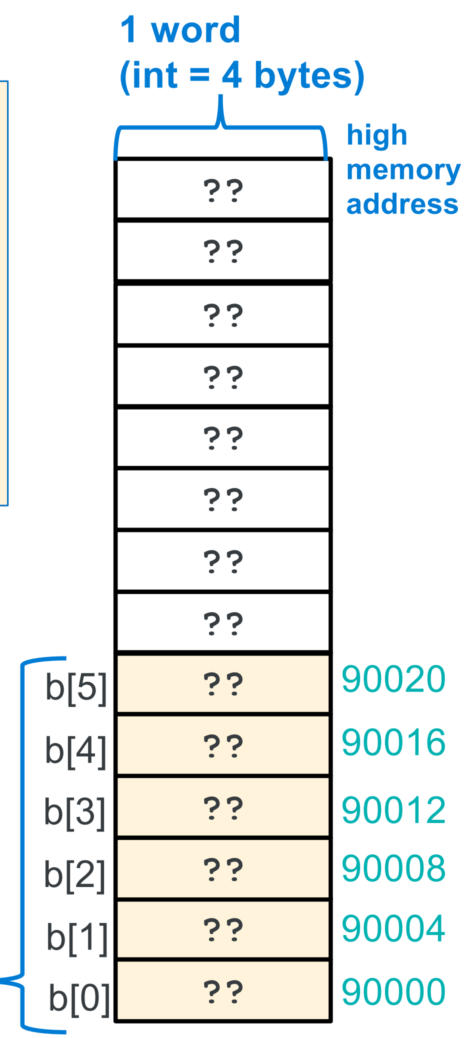
x

How many elements are in an array?

- **The number of elements of space allocated to an array (called element count) and indirectly the total size in bytes of an array** is <u>not stored anywhere</u>!!!!!!
- An **array name** is just the **address of the first element in a block of contiguous memory**
    - So, an array does not know its own size!

```
#define SZ 6
int block[SZ];        // you specify the array has SZ elements
int indx;             // use when SZ is defined

for (indx = 0; indx < SZ; indx++)
      block[indx] = 0;
```

```
int b[6];
```



x

Determining Element Count: compile time calculation

- Programmatically determining the element count in a compiler calculated array

  `sizeof(array) / sizeof(of just one element in the array)`

- `sizeof(array)` **only works** when used in the SAME **scope** where the array variable was defined

```
#include <stddef.h>
int main()
{

    int block[] =
        {2, 3, 5, 6, 11, 13};    // automatic: compiler calculates array size

    int cnt = (int)(sizeof(block) / sizeof(block[0]));    // in this case cnt = 6

    for (int indx = 0; indx < cnt; indx++)
        block[indx] = 0;
```


x

Pointers and Arrays - 1

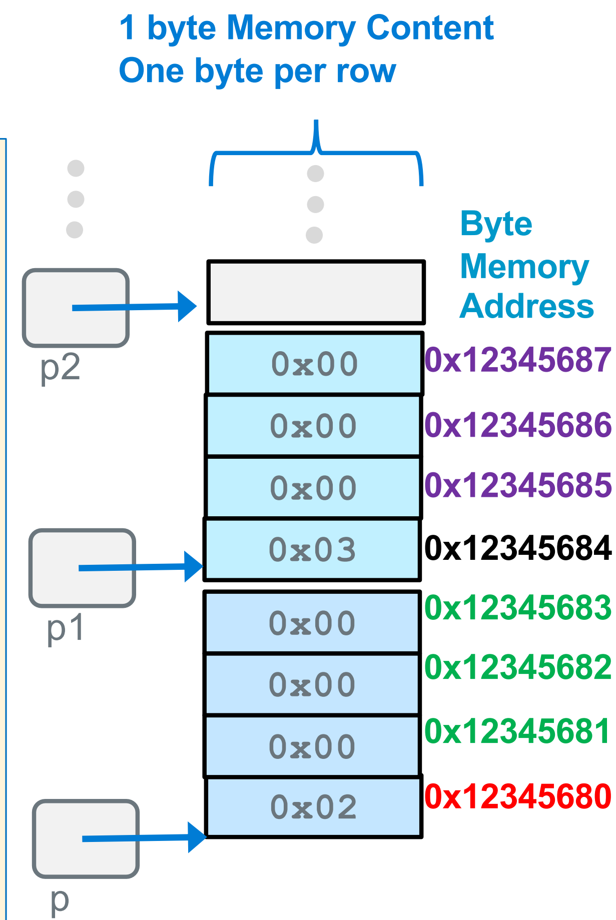
- A few slides back we stated: Array name (by itself) on the Rside evaluates to the address of the first element of the array

```
int buf[] = {2, 3, 5, 6, 11};
```

- Array indexing syntax ([ ]) an operator that performs *pointer arithmetic*

- **buf and &buf[0]** on the **Rside are equivalent**, ***both evaluate*** to the address of the first array element

```
int *p = buf;          // or int *p = &buf[0];
int *p1 = &buf[1];
int *p2 = &buf[2];
int *p3 = &buf[3];
```



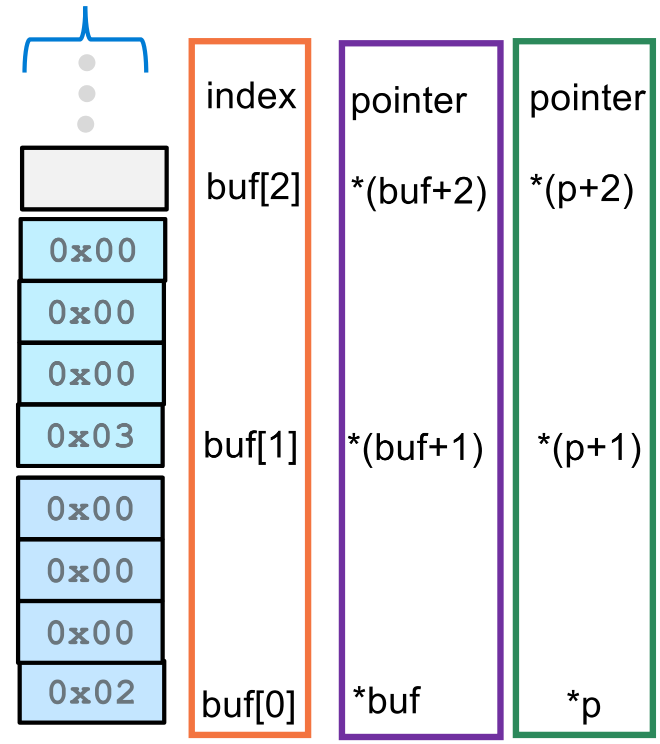
x

Pointers and Arrays - 2

- When p is a pointer, the actual evaluation of the address:
  - `(p+1)` **depends on the base type** the pointer p points at
- `(p+1)` adds `1 x sizeof(what p points at)` bytes to `p`
  - `++p` is equivalent to `p = p + 1`
- Using pointer arithmetic to find array elements:
  - Address of the second element `&buf[1]` is `(buf + 1)`
  - It can be referenced as `*(buf + 1)` **or** `buf[1]`

```c
int buf[] = {2, 3, 5, 6, 11};
int *p;
p = buf;

*p = *p + 10;                  // {12, 3, 5, 6, 11}
*(p + 1) = *(buf + 1) + 10;    // {12, 13, 5, 6, 11}
```

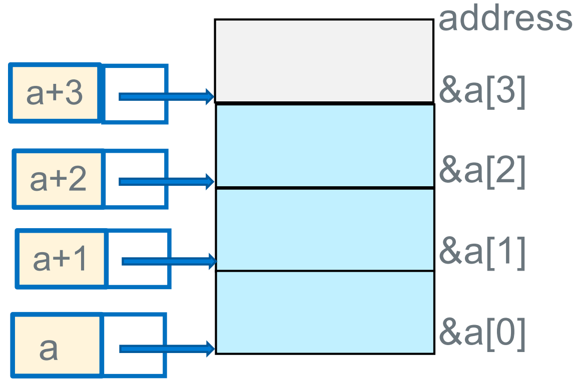
1 byte Memory Content
One byte per row



x

Pointer Arithmetic In Use – C's Performance Focus

```
char a[] = {'A', 'B', 'C'};
```



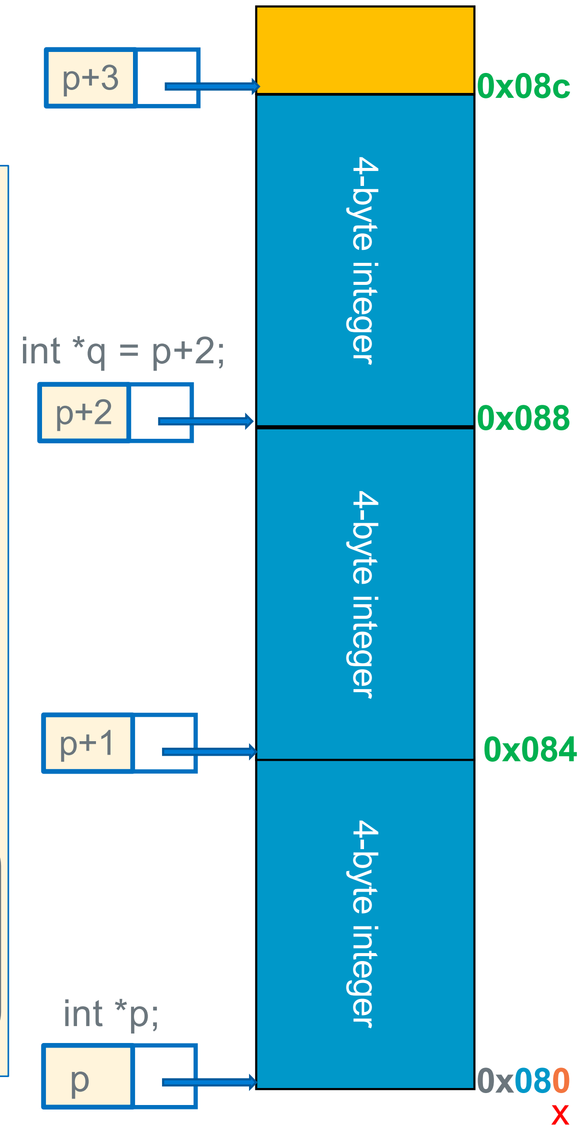
- **Alert!:** C performance focus **<u>does</u>** **<u>not</u>** perform any array "bounds checking"
- **Performance by Design**: *bound checking **<u>slows down execution</u>** of a properly written program*
- Example: array **a** of **length i**, C **<u>does</u>** **<u>not</u>** **<u>verify</u>** that **a[j] or *(a + j)** is valid (does not check: 0 ≤ j < i)
    - C simply *"translates"* and accesses the memory specified from: `a[j]` to be `*(a + j)` which may be *outside the bounds* of the array
    - OS only ***"faults"*** for an incorrect <u>access</u> to memory (read-only or <u>not</u> assigned to your process)
        - It does not fault for out of bound indexes or out of scope
- **Lack of bound checking** is a common source of **errors** and **bugs** and is a common criticism of C

x

Pointer Arithmetic

- **You <u>cannot</u> add two pointers** *(what is the reason?)*
- A pointer q <u>can be subtracted</u> from another pointer p when the pointers are the same type – best done only within arrays!
- The value of **`(p-q)`** is the number of **elements between** the two pointers
  - Using memory address arithmetic (p and q Rside are both byte addresses):
  - Notice that it is sizeof(*p) below: it is what **p points at** and not **sizeof(p) which is the size of the pointer**!

```
distance in elements = (p - q) / sizeof(*p)

(p + 3) - p = 3 = (0x08c - 0x080)/4 = 3
```



Pointer Comparisons

- Pointers (**same type**) can be compared with the comparison operators:

  `<, <=, ==, !=, >=, >`

```c
int numb[] = {9, 8, 1, 9, 5};
int *end;
int *a;
end = numb + (int) (sizeof(numb)/sizeof(*numb));
a = numb;
while (a < end)  // compares two pointers (address)
      /* rest of code including doing an a++ */
```

size of what numb points at!

- Invalid, Undefined, or **risky** pointer arithmetic (some examples)
  - Add, multiply, divide on two pointers
  - Subtract two pointers of different types or pointing at different arrays
  - Compare two pointers of different types
  - Subtract a pointer from an integer

X

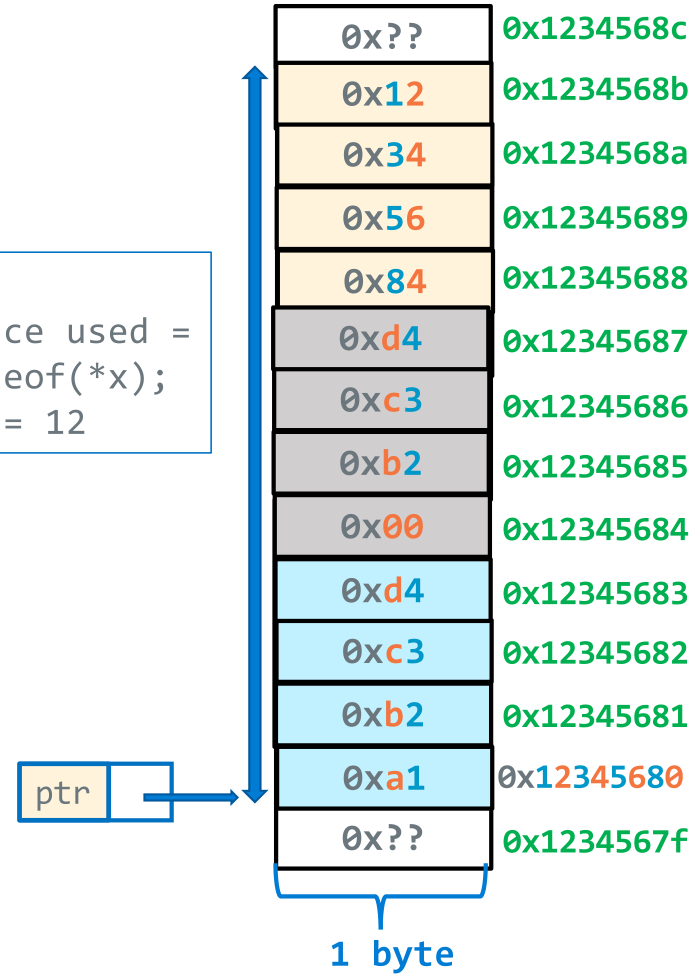
Using Pointers to Traverse an array

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));

for (int j = 0; j < cnt; j++)
    printf("%#x\n", x[j]);
}
```

```
cnt   = 3;
actual space used =
 cnt * sizeof(*x);
           = 12
```

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));
int *ptr = x;                    // or &x[0]

for (int j = 0; j < cnt; j++)
    printf("%#x\n", *(ptr + j));
}
```

Brute force translation to pointers



Fast Ways to Traverse an Array: Use a Limit Pointer

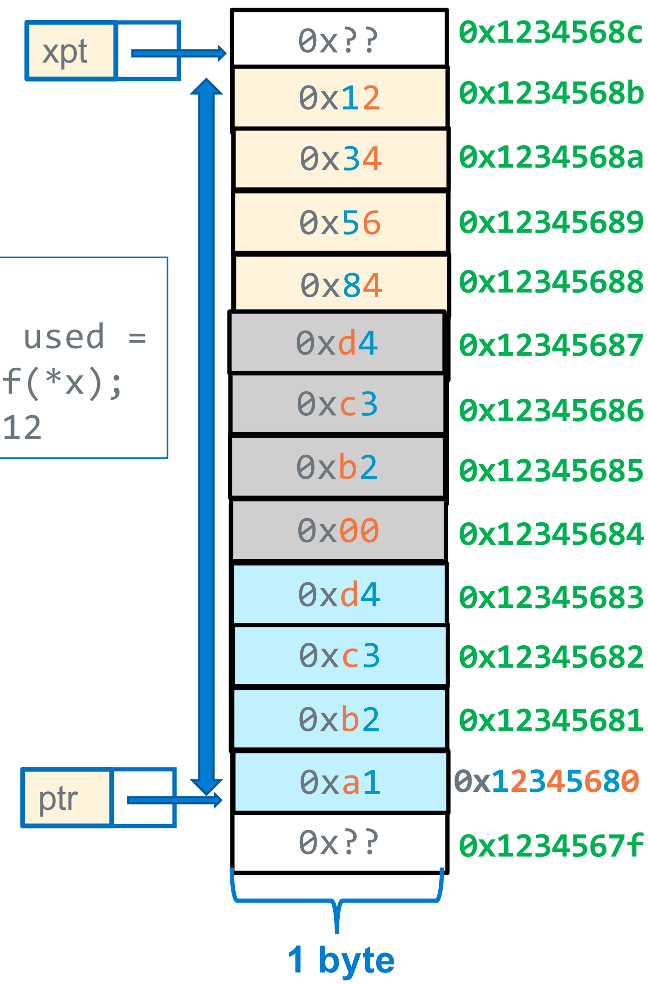
```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));

int *ptr;
int *xptr;
ptr = x; //or &x[0]
xptr = ptr + cnt;
```

xpt is a **loop limit pointer**
it points **1 element past**
the end of the array

```
while (ptr < xptr) {
    printf("%#x\n", *ptr);
    ptr++;
}
```

```
cnt  = 3;
actual space used =
 cnt * sizeof(*x);
          = 12
```

```
 % ./a.out
0xd4c3b2a1
0xd4c3b200
0x12345684
```



x

C Precedence and Pointers

- ++ -- pre and post increment combined with pointers can create code that is complex, hard to read and difficult to maintain
- Use () to help readability

common	With Parentheses	Meaning
`*p++`	`*(p++)`	(1)The Rvalue is the object that p points at (2)increment pointer p to next element ++ is higher than *
`(*p)++`		(1)Rvalue is the object that p points at (2)increment the object
`*++p`	`*(++p)`	(1)Increment pointer p first to the next element (2)Rvalue is the object that the incremented pointer points at
`++*p`	`++(*p)`	Rvalue is the incremented value of the object that p points at

Operator	*Description*	*Associativity*
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
?:	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

X