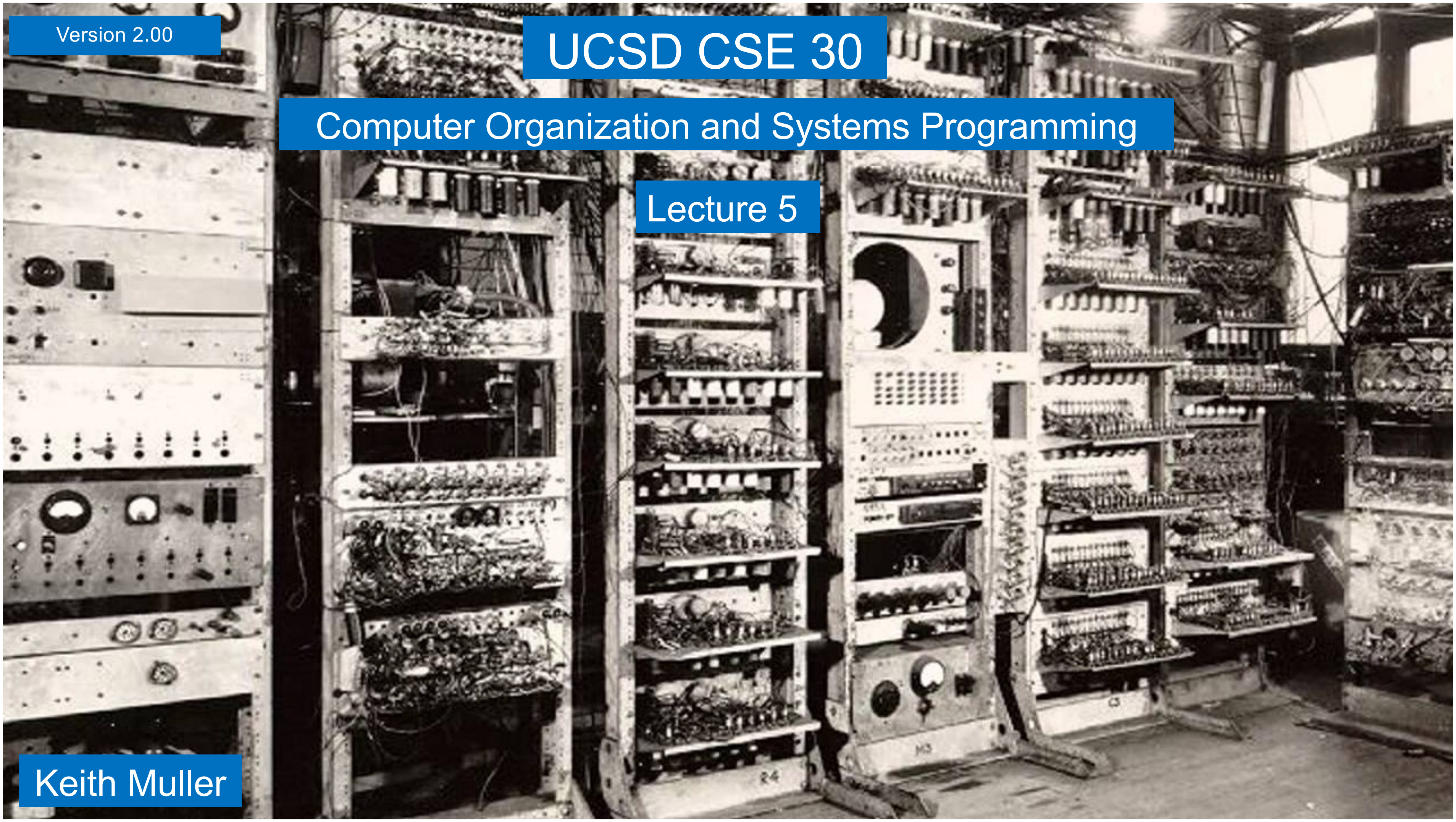


Version 2.00

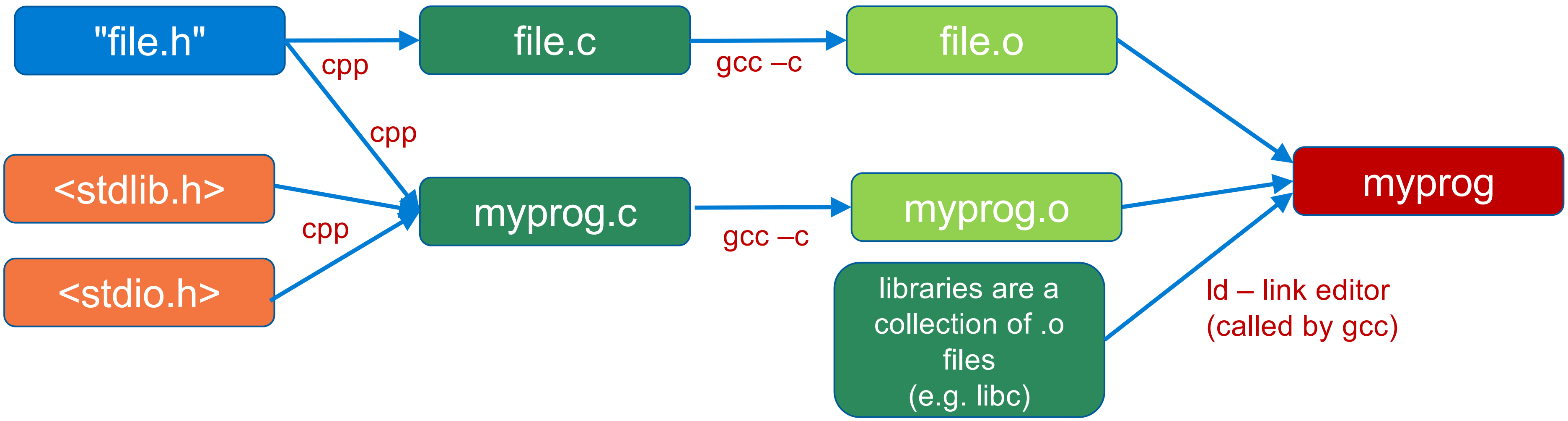
UCSD CSE 30

Computer Organization and Systems Programming

Lecture 5

Keith Muller

Compiling Multi-File Programs (assembly steps not shown)



1. compile each .c file independently to a .o object file this requires you use the –c flag to gcc to only compile and assemble and NOT to call the liner yet

```
gcc -Wall –Wextra –Werror –c file.c   # creates file.o
gcc -Wall –Wextra –Werror –c myprog.c   # creates myprog.o
```

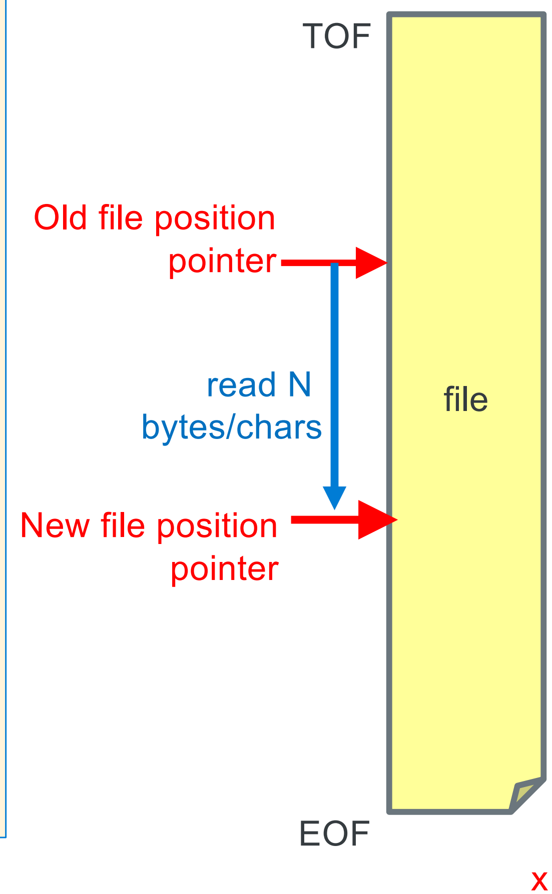
2. link all the .o objects files and libraries (aggregation of multiple .o files) to produce an executable file (gcc calls ld, the linker
   - The .o's in the libraries are automatically linked in as needed to produce an executable file

```
gcc -Wall –Wextra myprog.o file.o –o myprog
```


x

C standard I/O Library (stdio) File I/O
File Position Pointer and EOF

- Read/write functions in the standard I/O library *advances* the ***file position pointer*** from the ***top of a file*** (before the 1st byte if any) towards the ***end of the file*** **after each call** to a read/write function
  - **Side effect of call:** file position pointer moves towards the **end of file** by number of bytes read/written
- **standard I/O File position pointer** indicates where in the file (byte distance from the top of the file) the next read/write I/O will occur
- Performing a sequence of read/write operations (without using any other stdio functions to move the file pointer between the read/write calls) performs what is called **Sequential I/O** (sequential read & sequential write)
- EOF condition state may be set after a **read operation**
  - After the last byte is read in a file, additional reads results in a **function return value** of EOF
  - **EOF signals** no more data is available to be read
  - EOF is **NOT a character in the file**, but a condition state on the stream
  - EOF is usually a #define EOF -1 macro located in the file stdio.h (later in course)



C Library Function API : Simple Character I/O – Used in PA3

Operation	Usage Examples
Write a char	`int status; int c;` `status = putchar(c);          /* Writes to screen stdout */`
Read a char	`int c;` `c = getchar();                /* Reads from keyboard stdin */`

```
#include <stdio.h>  // import the public interface
```

`int putchar(int c);`

- writes c (demoted to a char) **to** `stdout`
- returns either: c on success *OR* EOF (a macro often defined as -1) on failure
- see % man 3 putchar

`int getchar(void);`

- returns the next input character (if present) **promoted to an int** read **from** `stdin`
- see % man 3 getchar

- Make sure you use int variables with putchar() and putchar()
- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition (**-1) which is outside the range of valid characters

Why is character I/O using an int?

Answer: Needs to indicate an EOF (-1) condition that is not a valid char

X

Character I/O (Also the Primary loop in PA3)

```
// copy stdin to stdout one char at a time
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int c;


   while ((c = getchar()) != EOF) {
       (void)putchar(c);    // ignore return value
   }

   return EXIT_SUCCESS;
}
```

Always check return code to handle EOF
EOF is a macro integer in stdio.h

```
% ./a.out
thIS is a TeSt
thIS is a TeSt
^d
%
%./a.out < a > b
```

Typed on keyboard

Printed by program

Typed on keyboard

Copies file a to file b

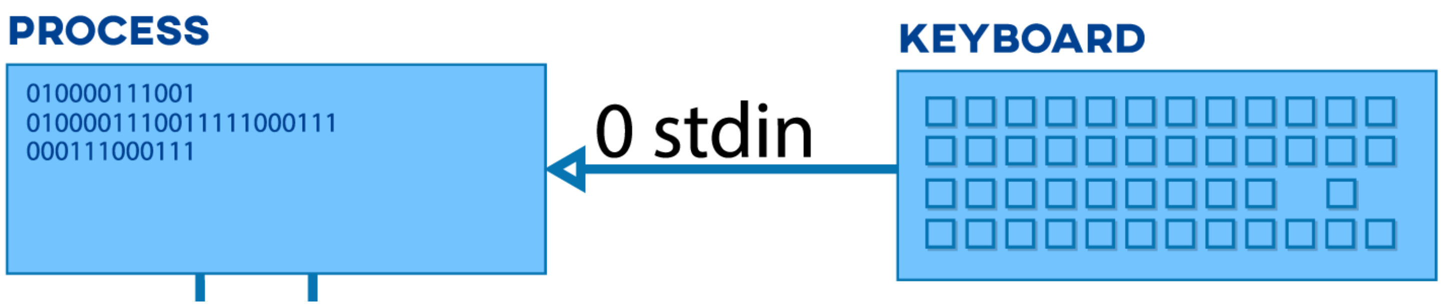
Always check return codes unless you do not need it

Sometimes you may see a (void) cast which indicates ***ignoring the return value is deliberate*** this is often required by many coding standards

Make sure you use int variable with getchar() and putchar()!

X

stdio File I/O – Working with a Keyboard



How do I signal EOF?

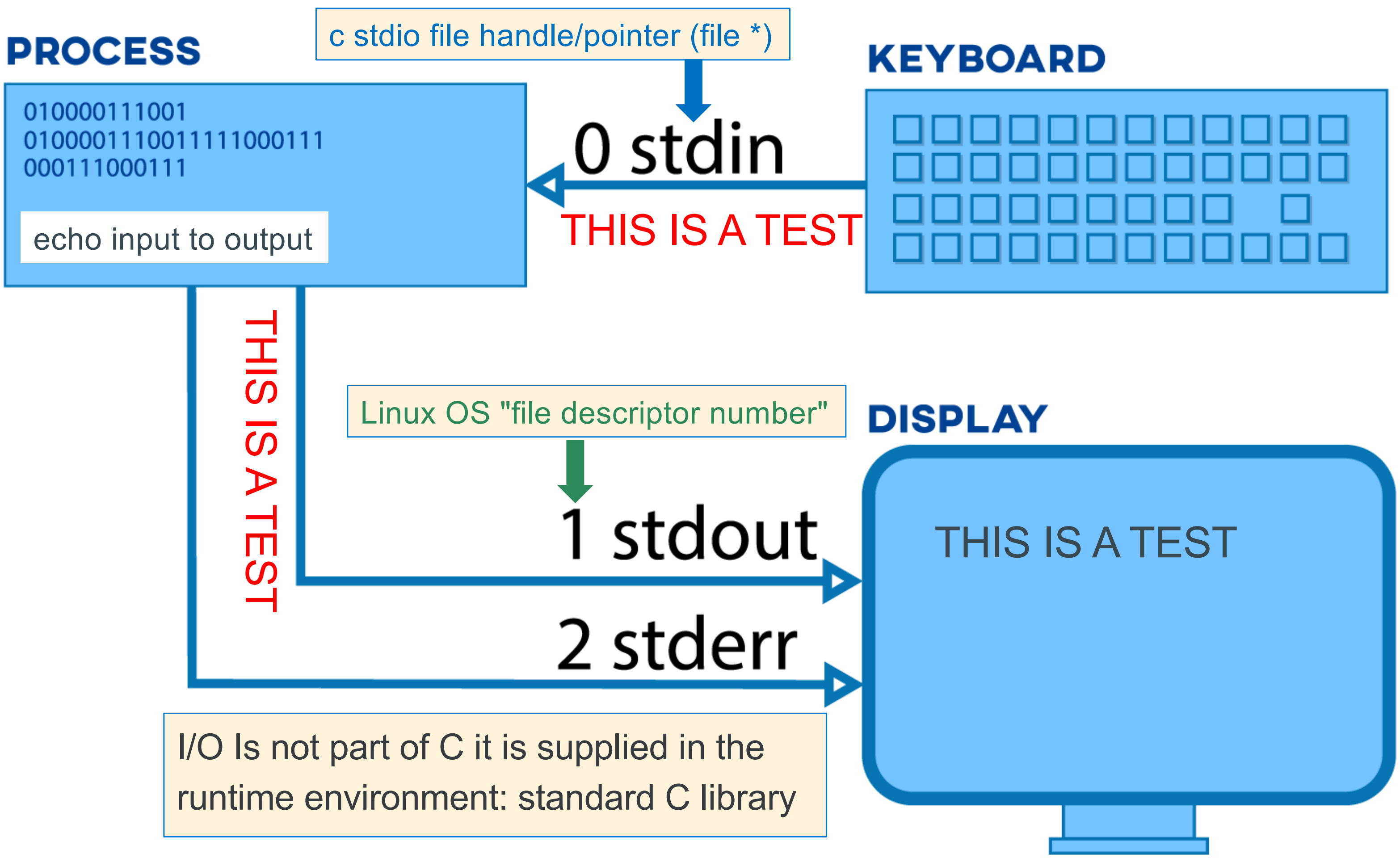
- How can you have an EOF **when reading from a keyboard**?
- stdio I/O library functions **designed** to work primarily on **files**
  - With keyboard devices the semantics of *file operations* needs to be *"simulated"*
- Example: when a program (or a shell) is reading the keyboard and is blocked waiting for input it is waiting for you to type a line
  - **This is NOT an EOF condition**
- To set an *EOF condition from the keyboard,* type on an input line all by itself:

  *two key combination (**ctrl key** and the **d key** at same time),* **followed by a return/enter**

  ctrl-d *often shown in slides etc. as* ***^d***

x

Linux/Unix Process and Standard I/O (CSE 15L)



x

C Library Function: Simple Formatted Printing

Task	Example Function Calls

Write formatted data	`int status;`
	`status = fprintf(stderr, "%d\n", i);`
	`status = printf("%d\n", i);      /* Writes to stdout */`

```
#include <stdio.h>  // import the public interface

int fprintf(FILE *file, const char *format, ...);
```

- Write chars to the file identified by **file** (stdout, stderr are already open)
- Convert values to chars, as directed by **format**
- Return count of chars successfully written
- **Format** is the output specifications enclosed in a "string"
- Returns a negative value if an error occurs

```
int printf(const char *format, ...); // *format - later in course
```

- Equivalent to `fprintf(stdout, format, ...);`

- Type % `man 3 printf` for more information on *format*

x

Some Formatted Output Conversion Examples

- Conversion specifications example
  - **%d** conversion specifier for **int** variables
  - **%c** conversion specifier for **char** variables
  - many more conversion specifiers (online manual: `% man printf` and the textbooks)

```
int i = 10;
char z = 'i';
char a[] = " Hello\n";

printf("%c = %d,%s", z, i, a); // write to stdout
fprintf(stderr, "This is an error message to stderr\n");
```

  - Output

```
i = 10, Hello
This is an error message to stderr
```


x

Conditional Statements (`if`, `while`, `do...while`, `for`)

- **C conditional test expressions**: **0 (NULL) is FALSE**, **any non-0 value is TRUE**
- **C comparison operators (==, !=, >, etc.) evaluate to either 0 (false) or 1 (true)**
- Legal in Java and in C:

```
i = 0;
if (i == 5)
   statement1;
else
   statement2;
```

Which statement is executed after the if statement test?

- Illegal in Java, but **legal** in C (often a typo!):

```
i = 0;
if (i = 5)
   statement1;
else
   statement2;
```

Assignment operators evaluate to the value that is assigned, so….
Which statement is executed after the if statement test?

x

Program Flow – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal** evaluation

```
if ((x == 5) || (y > 3))  // if x == 5 then y > 3 is not evaluated
```

- **Each** expression argument is evaluated **in sequence** from left to right including any side effects (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated *(for performance)*

x

Program Flow – Short Circuit or Minimal Evaluation

```
if ((a != 0) && func(b))     // if a is 0, func(b) is not called
    do_something();
```

```
// if (((x > 0) && (c == 'Q')) evaluates to non zero (true)
// then (b == 3) is not tested

while (((x > 0) && (c == 'Q')) || (b == 3)) {  // c short circuit
    x = x / 2;
    if (x == 0) {
       return 0;
    }
}
```


x

Be Careful with the comma , sequence operator

- Sequence Operator ,

  expr1,expr2

- Evaluates *expr1* first and then *expr2* evaluates to or returns *expr2*

```
for (i = 0, j = 0; i < 10; i++, j++)
   …
```

- Unexpected results with , operator (some compilers will warn)

```
    i = 64,323;         // i = 64 (assigns first)
    i = (64, 323);      // i = 323 (value of expression)
```


x

Review: Binary Numbering

- Binary is base 2
  - *adjective:* being in a state of one of two **mutually exclusive** conditions such as **on** or off, true or false, molten or frozen, presence or absence of a signal
  - From Late Latin *bīnārius* ("consisting of two")
- Two symbols:

  0 1
- Numbers in C that start with 0b are binary
- Example: What is 0b110 in base 10?
  - 0b110 = 110_2 = $(1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6_{10}$

powers of two

- A **bit** is a single binary digit
- A **byte** is an 8-bit value

$$Unsigned\ binary\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 2^i = b_{n-1}2^{N-1} + b_{n-2}2^{N-2} + \ldots + b_1 2^1 + b_0 2^0$$

x

Review: Hexadecimal Numbering

- hexadecimal is base 16
  - From “hexa” (Ancient Greek ἑξα-) ⇒ six
  - and from “decem” (Latin) ⇒ ten
- Sixteen symbols

  0 1 2 3 4 5 6 7 8 9 a b c d e f

- Numbers in C that start with 0x are hexadecimal numbers
  - $\mathbf{16_{10}} = 0x10_{16}$
- Example: What is 0xa5 in base 10?
  - $0xa5 = a5_{16} = (10 \times 16^1) + (5 \times 16^0) = 165_{10}$
- Hexadecimal numbers are very commonly used in programming to express binary values
  - Imagine the difficulty in correctly expressing a 64-bit binary value in your code

$$Unsigned\ Hex\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 16^i = b_{n-1}16^{N-1} + b_{n-2}16^{N-2} + \ldots + b_1 16^1 + b_0 16^0$$

x

Binary <---> Hexadecimal Equivalences

- Hex → Binary: $16^1 = 2^4$ 1 digit hex = 4 digits binary
  1. Replace hex digits with binary digits
  2. Drop leading zeros
  - Example: 0x2d to binary
    - 0x2 is 0b00**10,** 0xd is 0b1101
    - Drop two leading zeros, answer is 0b**10**1101
- Binary → Hex: $2^4 = 16^1$
  1. **Pad** with enough leading zeros until number of digits is a multiple of 4
  2. **Replace** each **group of 4** with the **HEX equivalent**
  - Example: 0b101101
    - **Pad on the left** to: 0b 0010 1101
    - Replace to get: 0x2d

x

Number Base Overview (as written in C)

- Decimal is base 10 and Hexadecimal is base 16,
- **Hex digits** have 16 values 0 - 9 a - f (written in C as 0x0 – 0xf)
- No standard prefix in C for binary (most use hex)
  - gcc (compiler) allows 0b prefix others might not

Hex digit	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111

Hex digit	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
Decimal value	8	9	10	11	12	13	14	15
Binary value	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111

x

Hex to Binary (group 4 bits per digit from the right)

- Each Hex digit is 4 bits in base 2 $16^1 = 2^4$

$$0x\ \underbrace{f}_{1111}\ \underbrace{a}_{1010}\ \underbrace{5}_{0101}\ \underbrace{3}_{0011}$$

$$\boxed{0b1111101001010011}$$

binary start with a 0b in C

x

Binary to Hex (group 4 bits per digit from the right)

- 4 binary bits is one Hex digit $2^4 = 16^1$

$$0b\ \underbrace{0110}_{6}\ \underbrace{1010}_{a}\ \underbrace{0011}_{3}\ \underbrace{1111}_{f}$$

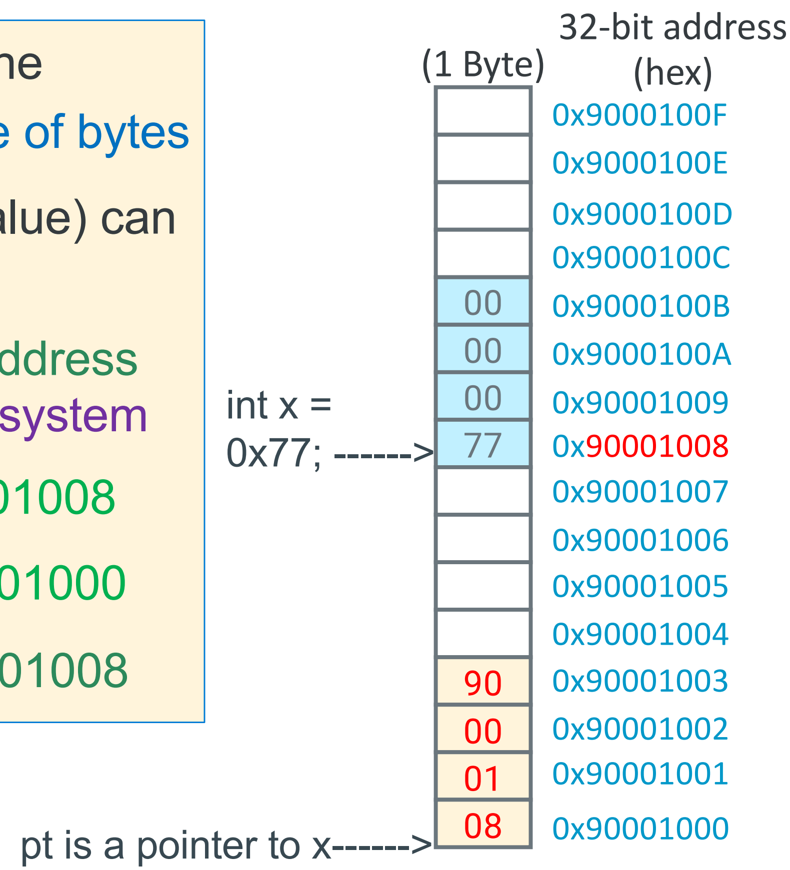
$$\boxed{0x6a3f}$$

hex start with 0x in C

x

Address and Pointers

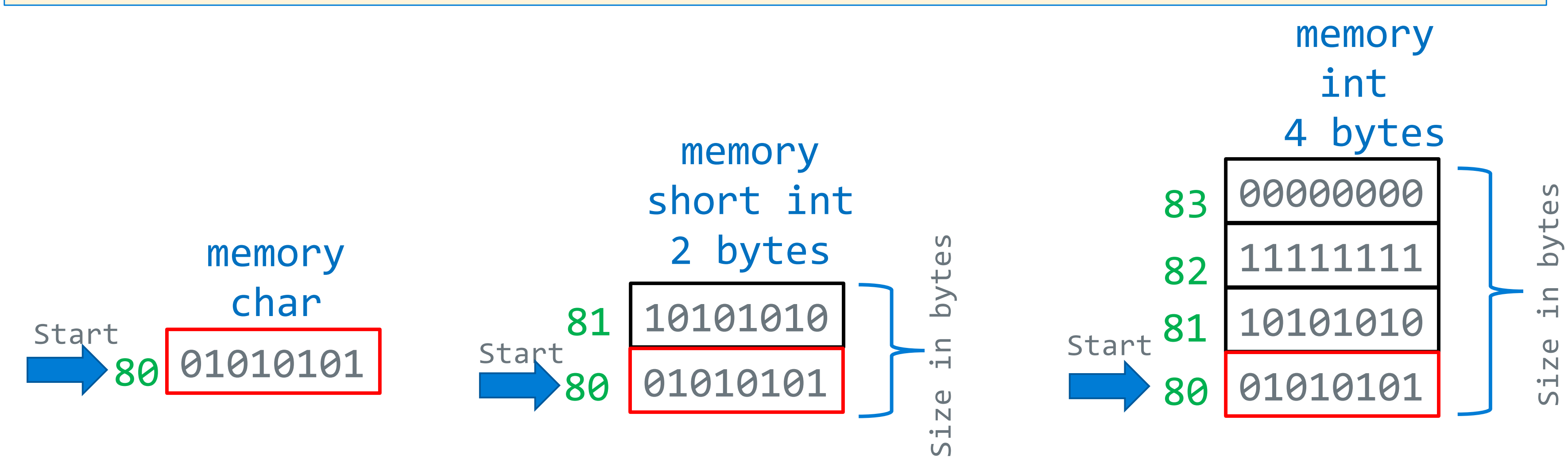
- An address refers to a location in memory, the lowest or first byte in a contiguous sequence of bytes
- A pointer is a variable whose contents (or value) can be properly used as an address
  - The value in a pointer *should* be a valid address allocated to the process by the operating system
- The variable x is at memory address 0x90001008
- The variable pt is at memory location 0x90001000
- The contents of pt is the address of x 0x90001008



x

Variables in Memory: Size and Address

- The number of **contiguous bytes** a variable uses is based on the *type* of the variable
  - Different variable types require different numbers of contiguous bytes
- ***Variable names*** map to a *starting address in memory*
- Example Below: Variables all starting at address 0x80, each box is a byte



x

Variables: Size

- **Integer types**
  - `char`, `int`
- **Floating Point**
  - `float`, `double`
- Modifiers for each base type
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]
  - `const`: variable read only
- **char type**
  - One byte in a byte addressable memory
  - **Signed** vs **Unsigned** Char implementations
  - **Be careful** char is unsigned on arm and signed on other HW like intel

C Data Type	AArch-32 contiguous Bytes	AArch-64 contiguous Bytes	printf specification
char (arm unsigned)	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	8	16	%Lf
pointer *	4	8	%p

size of a pointer is the word size

x

sizeof(): Variable Size (number of bytes) *Operator*

```
#include <stddef.h>
/* size_t type may vary by system but is always unsigned */
```

sizeof() operator returns a value of type size_t:

the number of bytes used to store a variable or variable type

```
size_t size = sizeof(variable_type);
              or
size_t size = sizeof(variable_name); // preferred!
```

- The argument to `sizeof()` is often an expression:

```
size = sizeof(int * 10);
```

  - reads as:
    - number of bytes required to store **10 integers (an array of [10])**

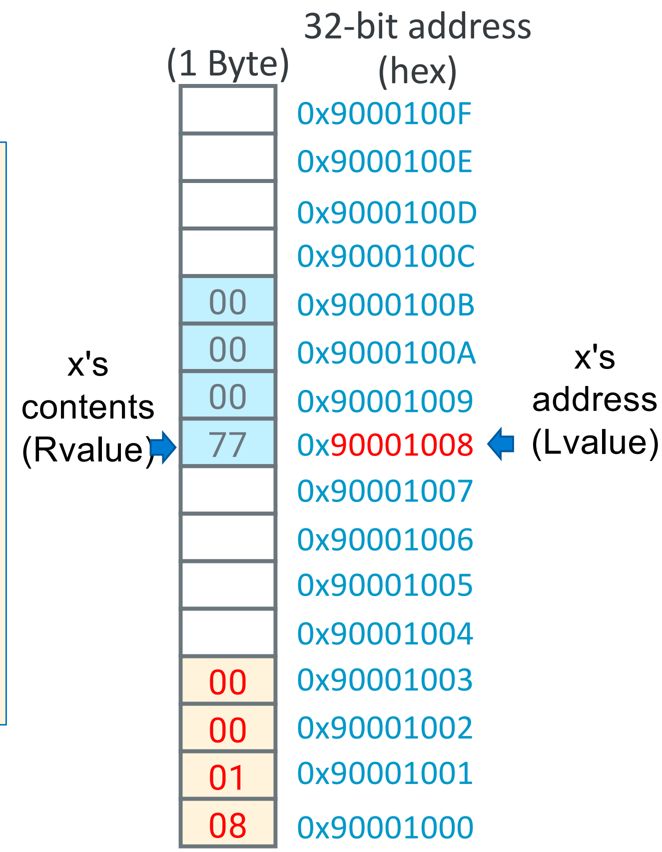
x

Memory Addresses & Memory Content

Variable names in a C statement evaluation

```
x = x + 1;     // Lvalue = Rvalue
```

- **Lvalue:** when on the left side (Lside or Left value) of the = sign
  - address where it is stored in memory – a constant
  - Address assigned to a variable cannot be changed at runtime
  - Does not require a memory read
- **Rvalue:** when on the right side (Rside or Right value) of an = sign
  - contents or value stored in the variable (at its memory address)
  - requires a memory read to obtain contents



x

Memory Addresses & Memory Content

One memory write required

One memory read required

```
y = 42;
x = y;          // Lvalue = Rvalue
```

y	42
x	42

copy

- **x** on left side (**Lside**) of the assignment operator = evaluates to:
  - **Address** of the memory assigned to the x – this is x's **Lvalue**
- **y** on right side (**Rside**) of the assignment operator = evaluates to:
  - **Contents** of the memory assigned to the variable y (type determines length – number of bytes) - this is y's **Rvalue**
- So, x = y; is:

Read memory at y (**Rvalue**); write it to memory at x's address (**Lvalue**)

x

Introduction: Address Operator: &

- Unary ***address operator*** (&) produces the **address** of where an identifier is in memory
  - Assigned address to g
- Example this might print:

  ***value** of g is: 42*

  ***address** of g is: 0x71a0a0*

  (the address will vary)

```
int main(void)
{
    int g = 42;

    printf("value of g is: %d\n", g);
    printf("address of g is: %p\n", &g);
    return EXIT_SUCCESS;
}
```

- *Tip*: `printf()` format specifier to display an address/pointer (in hex) is "%p"

x

Introduction: Address Operator: &

- Requirement: **identifier must have a Lvalue**
  - Cannot be used with constants (e.g., 12) or expressions (e.g., x + y)
  - Example: **&12** does not have an *Lvalue*,
    - so, &12 is **not** a legal expression
- How can I get an address for use on the **Rside**?
  - **&var** (any variable identifier or name)
  - **function_name** (name of a function, not func());
    - &funct_name is equivalent
  - **array_name** (name of the array like array_name[5]);
    - &array_name is equivalent

x

Pointer Variables

- In C, there is a *variable* ***type*** for **storing an address**: a ***pointer***
  - **Contents** of a pointer is an **unsigned** (positive numbers) **memory address**

```
type *name;  // defines a pointer; name contains address of a variable of type
```

- A pointer is defined by placing a ***star (**or **asterisk) (*)*** ***before*** the identifier (name)
- You also must specify the type of variable to which the pointer points
- **Pointers are typed**! Why?
  - The compiler needs to know the size (sizeof()) of the data **you are pointing at** (number of consecutive bytes to access) to use the pointer
- When the **Rside of a variable** contains a **memory address**, (it **evaluates** to an **address**) the variable is called a **pointer variable**

x

Pointer Variables - 2

- A pointer **cannot** point at itself, why?

```
int *p = &p; /* is not legal – type mismatch */
```

  - p is defined as (int *), a pointer to an int, **but**
  - the type of &p is (int **), a pointer to a pointer to an int
- Pointer variables all use the **same amount of memory** no matter what they point at

```
   int *iptr;
   char *cptr;

   printf("iptr(%u) cptr(%u)\n", sizeof(iptr), sizeof(cptr));
```

- Above prints on a 32-raspberry pi

```
% ./example
iptr(4) cptr(4)
```


x

Defining Pointer Variables

- Assigning a value to a pointer:

```
int *p = &i;  /* p points at i (assign address i to p) */
```

- Is the same as writing the following definition and assignment statements

```
int *p;         /* p is defined (not initialized) */
p = &i;         /* p points at i (assign address i to p */
```

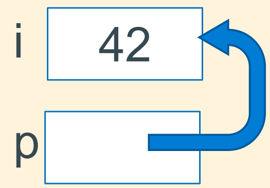
- The * is part of the definition of p and is not part of the variable name
  - The name of the variable is simply p, not *p
- C mostly ignores whitespace, so these three definitions are equivalent

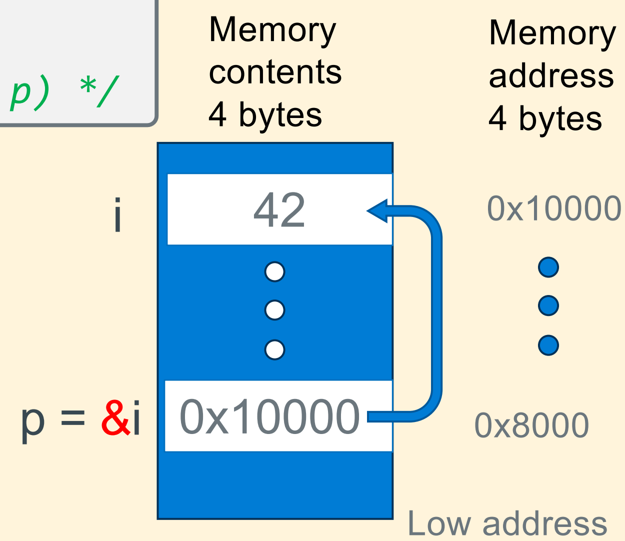
```
int  *p = &i;       /* Style A */
int * p = &i;       /* Style B */
int*  p = &i;       /* Style C */
```


x

Using Pointer Variables and the Address Operator & - 1

```
int i = 42;
int *p;  /* p contains the address of an integer */
p = &i;  /* p "points at" i (assign address of i to p) */
```





- Recommended: be careful when defining multiple pointers on the same line:

`int *p1, p2;` is not the same as: `int *p1, *p2;`

Use instead:

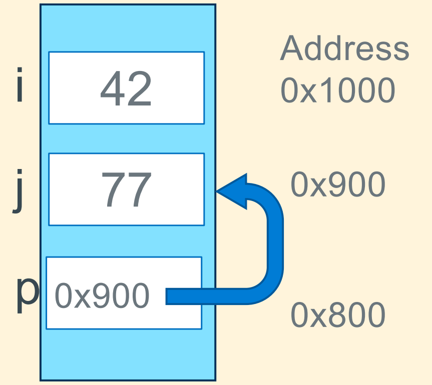
```
int *p1;
int *p2;
```


x

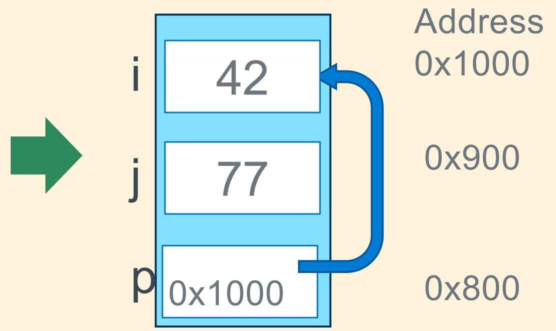
Using Pointer Variables and the Address Operator & - 2

- As with any variable, its value can be changed

```
p = &j;     /* p now points at j */
```




```
p = &i;     /* p now points at i */
```

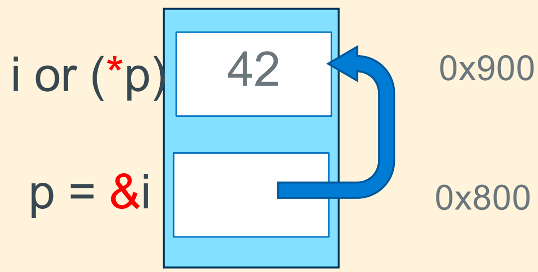


x

Indirection (or dereference) Operator: *

- The ***indirection operator*** (*) or the *dereference operator to a variable* is the **inverse** of the *address operator* (&)
- **address operator (&)** can be thought of as:

  "get the address of this box""



- **indirection operator (*)** can be thought of as:

  "follow the arrow to the next box and get its contents""

- **Indirection operator causes an additional read to occur**, when on either the Rside or Lside of a statement

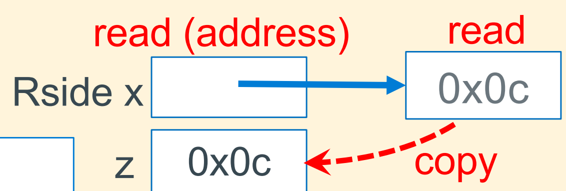
x

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)

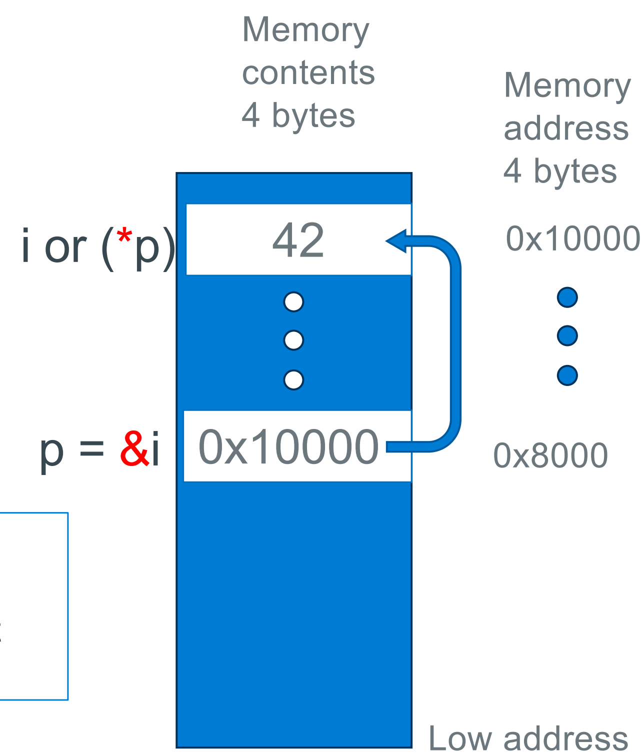
```
int i = 42;
int *p;
p = &i;
```

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

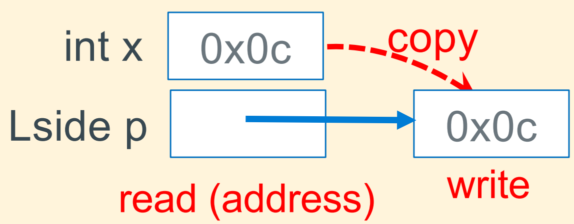


Lside Indirection Operator

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***

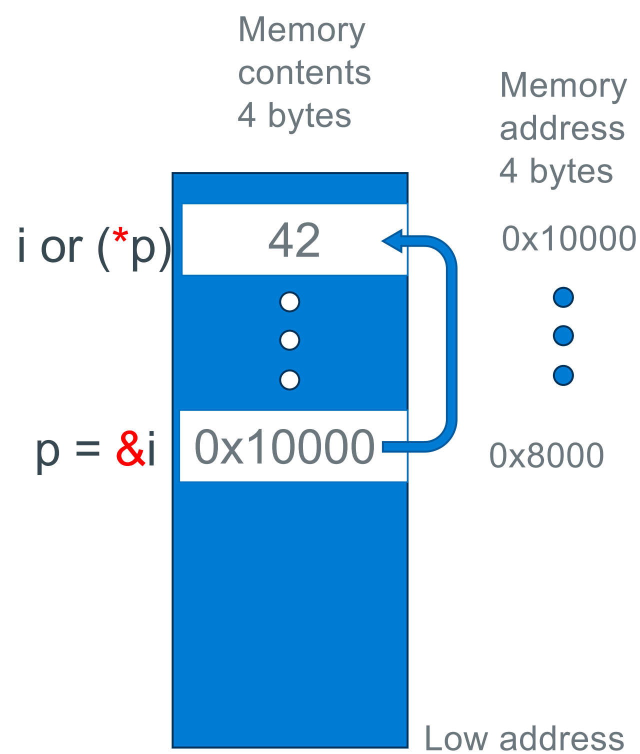
(p points at i)

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

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

No reads here
one write

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

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

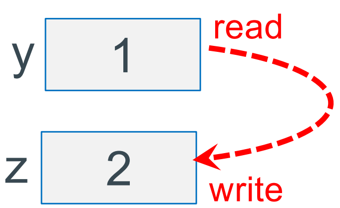


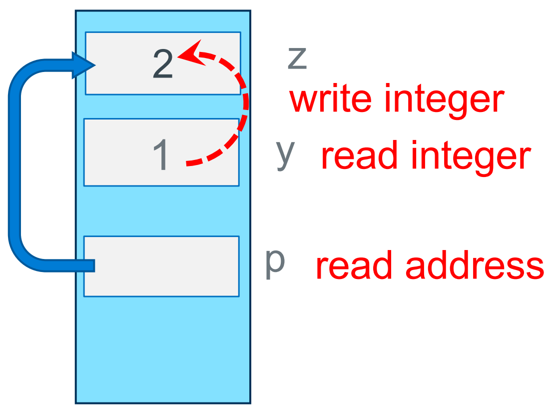
x

Each use of a * operator results in one additional read -1

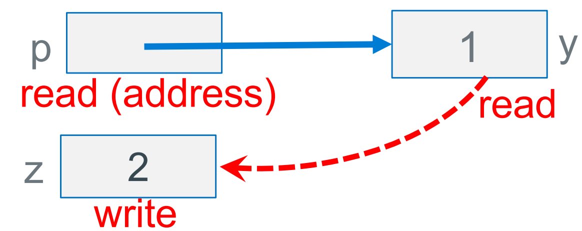
RULE: Each * when used as a dereference operator in a statement (either Lside or Rside) generates an additional read

```
int z = 2, y = 1;
z = y; // one read
```






```
int z = 2, y = 1;
int *p;
p = &y;
z = *x; // two reads on rside
```

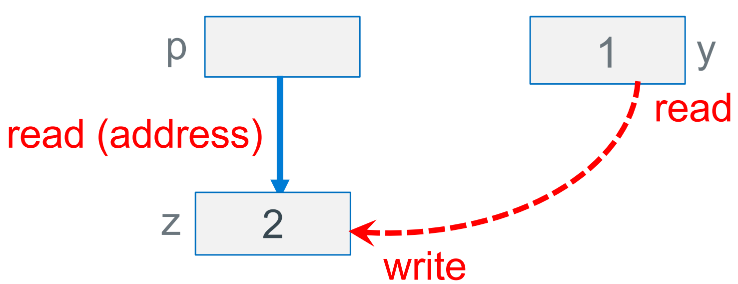


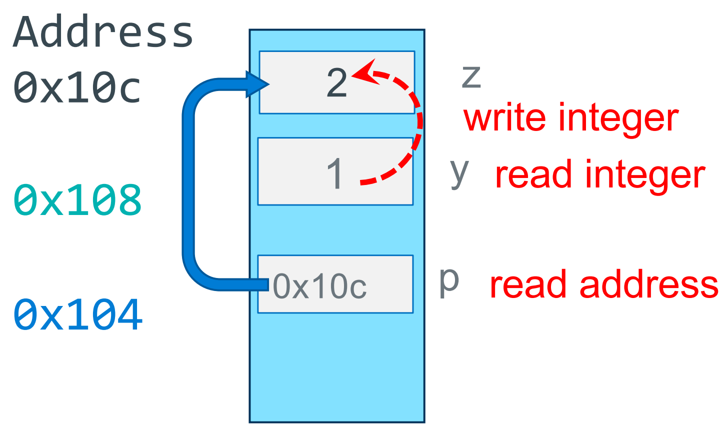
Aside: `z = *(&x); // same as z = x`

x

Each use of a * operator results in one additional read -2

```
int z = 2, y = 1;
int *x;
p = &z;
*p = y;     // one read on lside
```

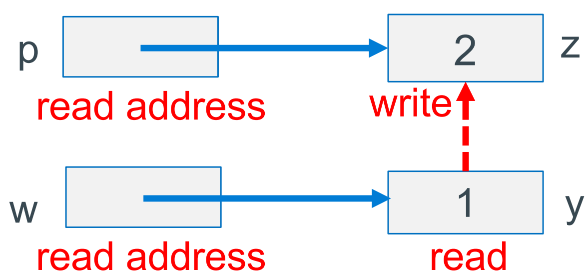


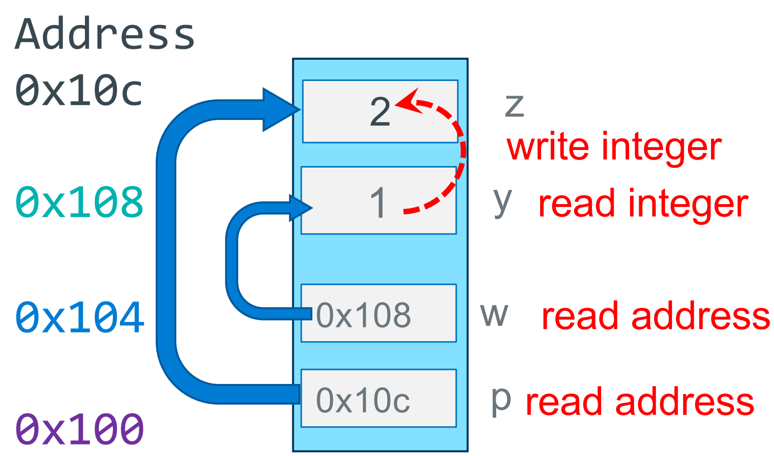


x

Each use of a * operator results in one additional read -2

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



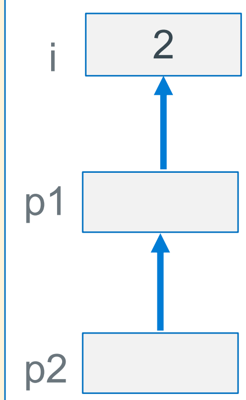


x

Pointer to Pointers (Double Indirection)

- Define a pointer to a pointer (p2 below)

```
int i = 2;
int *p1;
int **p2;
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 definition tells you how many reads it takes to get to the base type

  #reads to base type = number * + 1

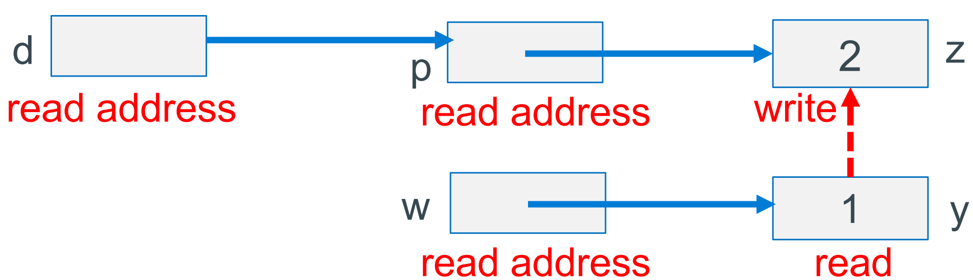
- Example:

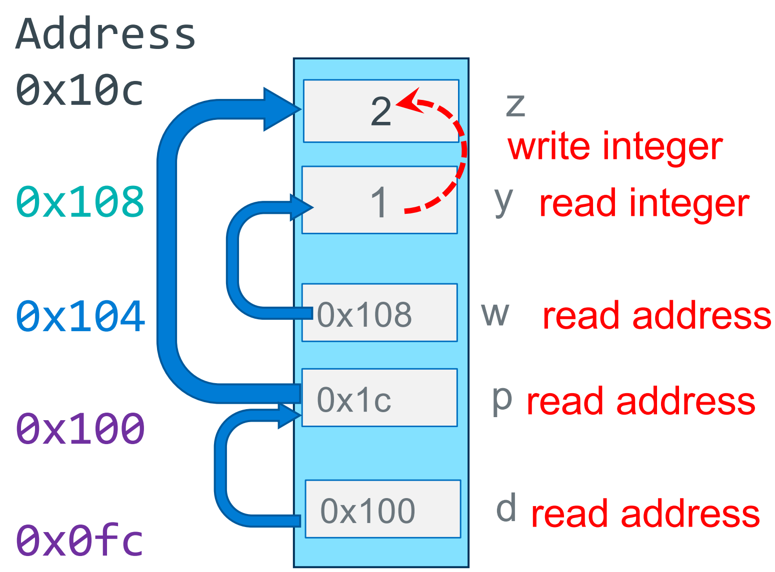
  `int **p2;` // requires 3 reads to get to the int

x

Double Indirection

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

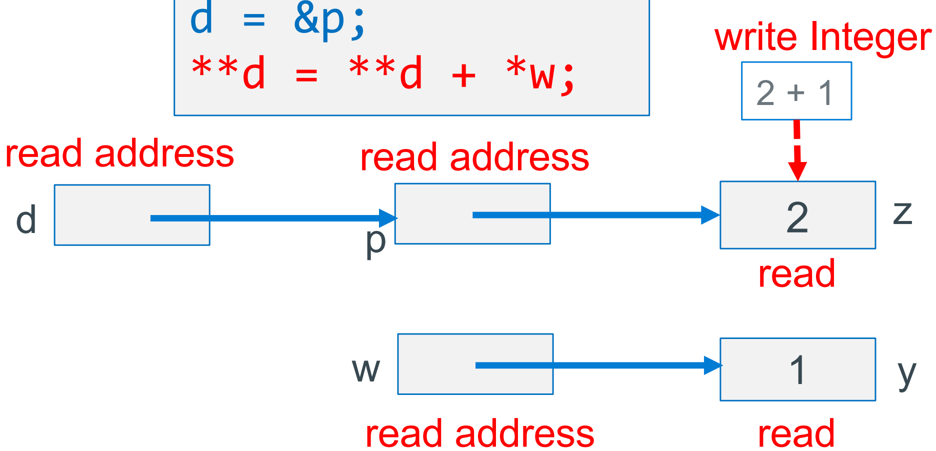


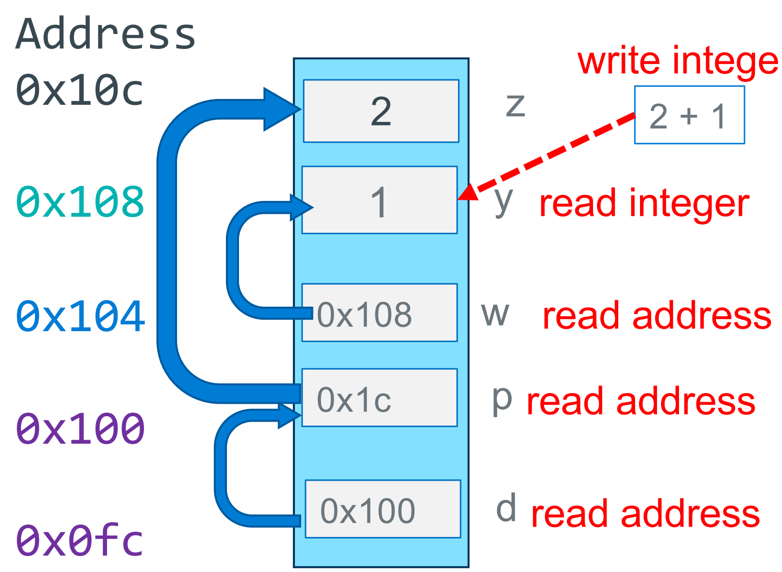


x

Double Indirection

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





Important Observe

**d on Lside is two reads

**d on Rside is three reads

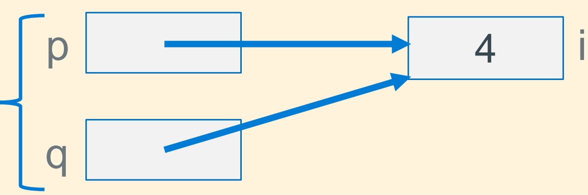
x

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 are aliases
*q = 4;  // changes i
```

*p and *q are aliases

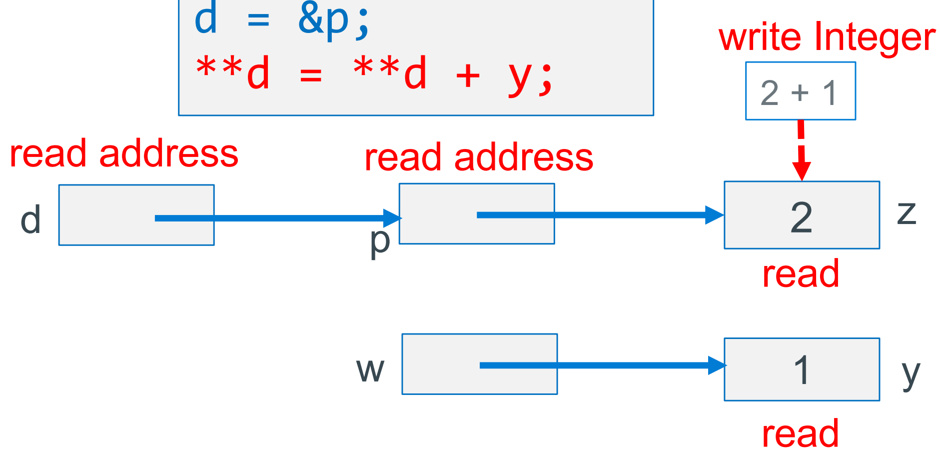


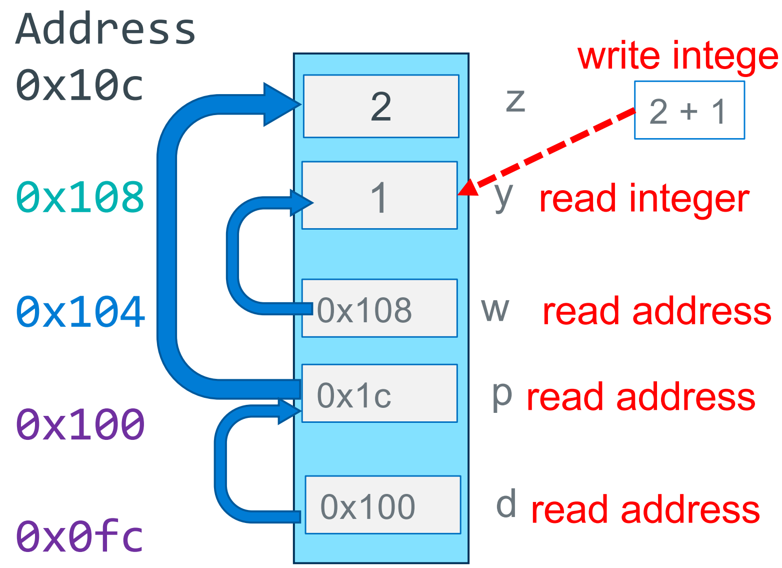
Result *p, *q and i all have the value of 4

x

Double Indirection Aliases

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





Important Observe
**d, *p and y are all aliases

x

The NULL Constant and Pointers

- **NULL is a constant** that **evaluates to zero (0)**
- You assign a pointer variable to contain NULL to indicate that the pointer does not point at anything
- A pointer variable with a value of NULL is called a "NULL pointer" (invalid address!)
- Memory location 0 (address is 0) is not a valid memory address in any C program
- Dereferencing NULL at runtime will cause a program fault (segmentation fault)!

```
p = NULL;
i = *p;                /* segmentation fault! */
*(int *)900000 = 25;   /* cast 900000 to a pointer */
                       /* if writeable address space, it works */
                       /* that memory location just changed */
```


x

Using the NULL Pointer

- Many functions return NULL to indicate an error has occurred

```
/* these are all equivalent */
int *p = NULL;
int *p = (int *)0;   // cast 0 to a pointer type
int *p = (void *)0;  // automatically gets converted to the correct type
```

- NULL is considered "false" when used in a Boolean context
  - **Remember: false expressions** in C are defined to be zero *or* NULL
- The following two are equivalent (the second one is preferred for readability):

```
if (p) ...
if (p != NULL) ...
```


x

Defining Arrays

<u>Definition</u>: `type name[count]`

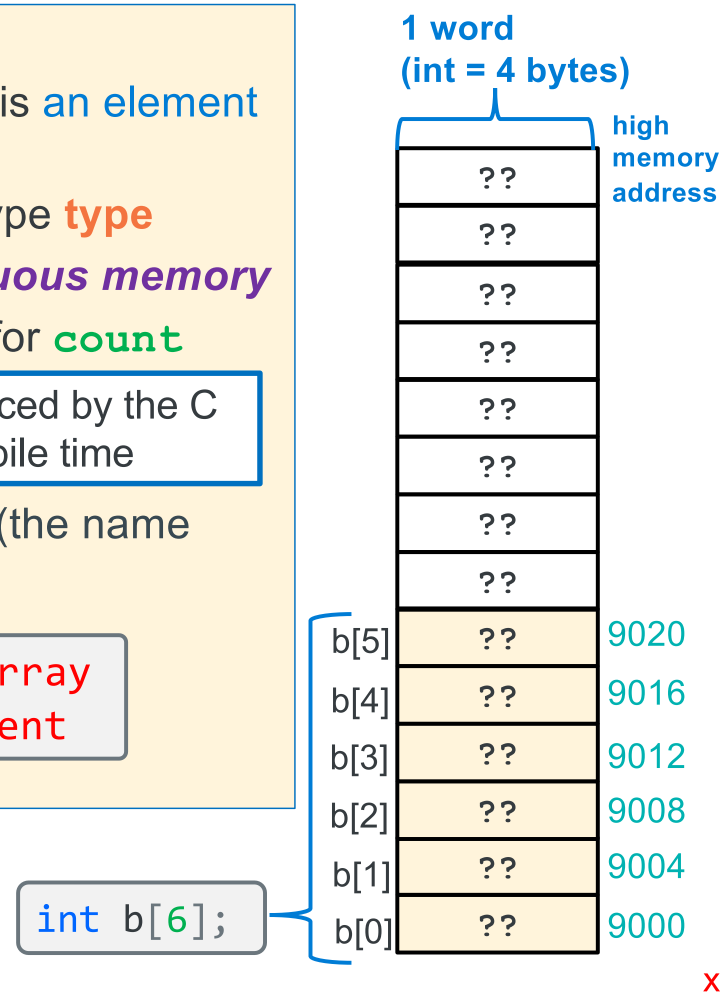
- ***"Compound"*** data type where each value in an array is an element of type
- Allocates **name** with a *fixed* count array elements of type **type**
- Allocates (`count * sizeof(type)`) bytes of ***contiguous memory***
- Common usage is to specify a compile-time constant for **`count`**

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

BSZ is a macro replaced by the C preprocessor at compile time

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

```
a = b;        // invalid does not copy the array
              // copy arrays element by element
```



Array Initialization

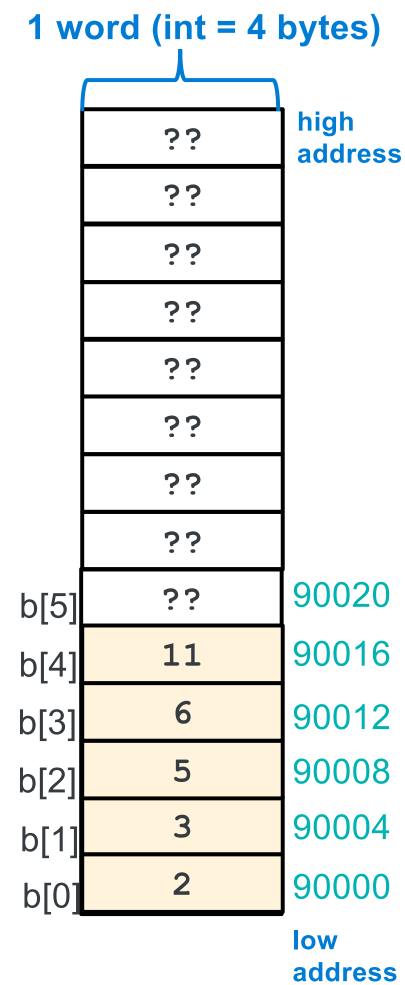
- Initialization: `type name[count] = {val0,…,valN};`
  - `{ }` *(optional)* initialization list can *only* be used at **time of definition**
  - If no `count` supplied, `count` is determined by compiler using the number of array initializers

    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**



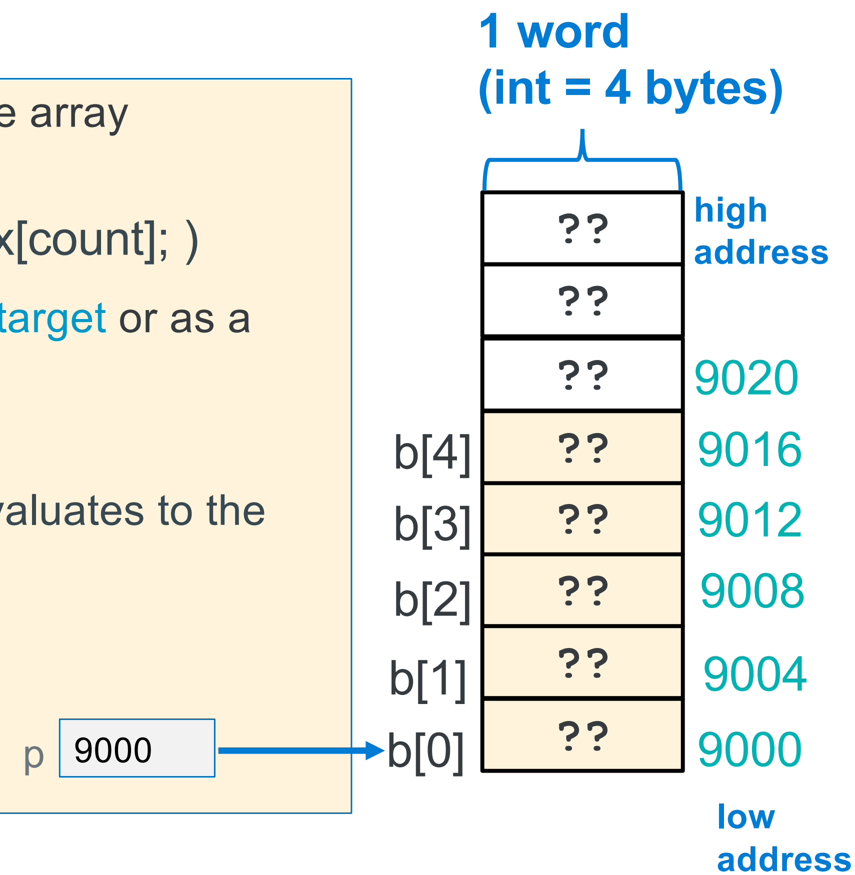
Accessing Arrays Using Indexing

- **`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[5];
int b[5];
```

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

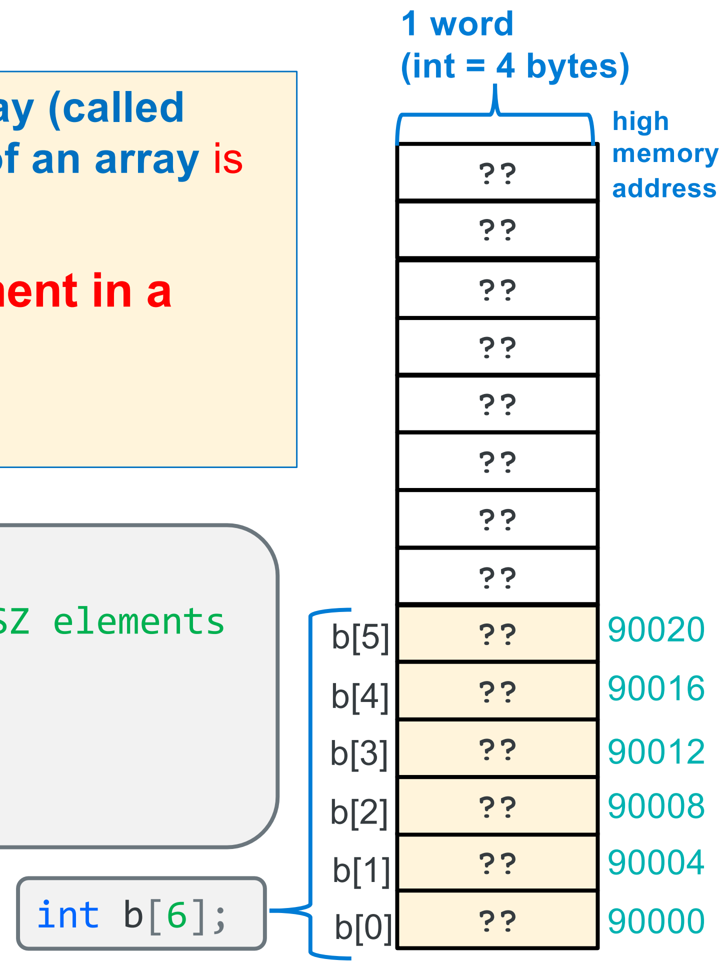
```
int b[5];
int *p = b;
```



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** <u>is 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;
```



Determining Element Count for a compiler calculated array

- Programmatically determining the element count in a compiler calculated array

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

- `sizeof(array)` **<u>only works</u>** when used in the SAME **scope** as where the array variable was defined

```
#include <stddef.h>

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

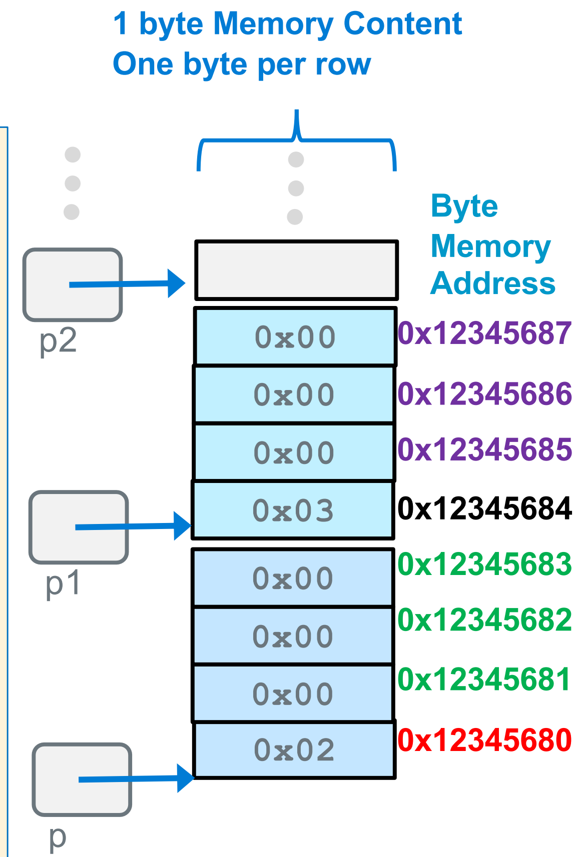
Pointers and Arrays - 1

- 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];

*p = *p + 10;
*p1 = *p1 + 10;        // {12, 13, 5, 6, 11}
```



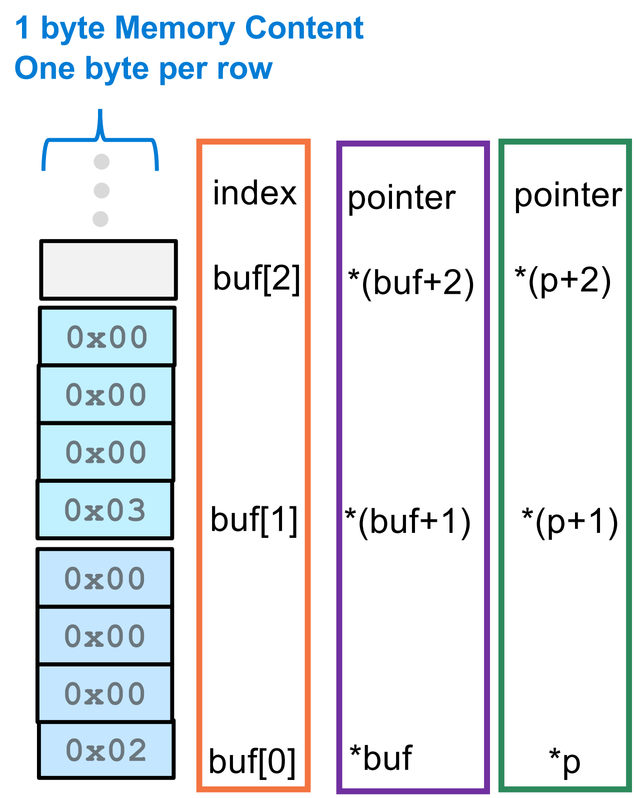
X

Pointers and Arrays - 2

When p is a pointer, the actual value of `(p+1)` **depends on the type** that 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]`

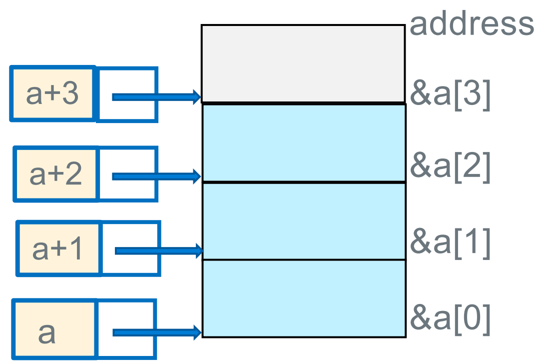
```
int buf[] = {2, 3, 5, 6, 11};
int *p;
p = buf;

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



x

Pointer Arithmetic In Use – C's Performance Focus

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



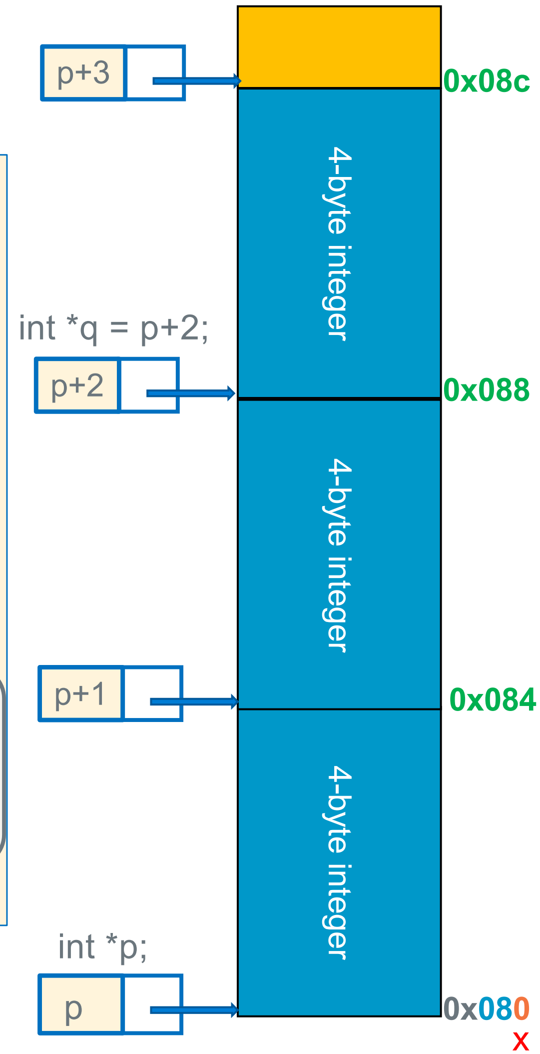
- **Alert!:** C performance focus **does not** perform any array "bounds checking"
- **Performance by Design**: *bound checking **slows down execution** of a properly written program*
- Example: array **a** of length i, C **does not verify** that **a[j] or *(a + j)** is valid (does not check: 0 ≤ j < 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 access to memory (read-only or not 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 cannot add two pointers** *(what is the reason?)*
- A pointer q can be subtracted 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):

```
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:

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

```
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++ */
```

- 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

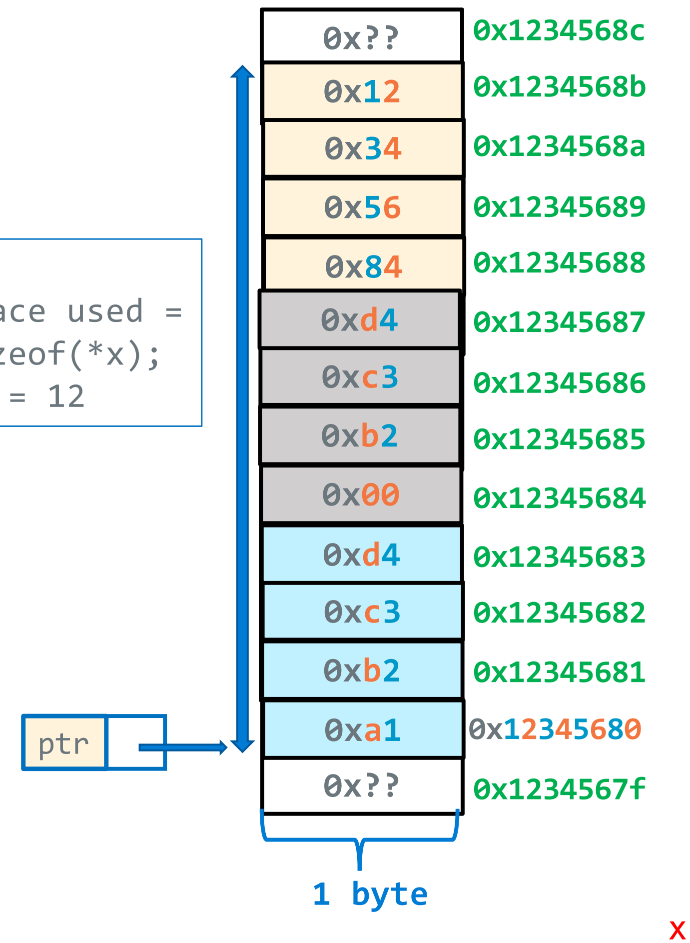
Using Pointers to Traverse an array

```
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

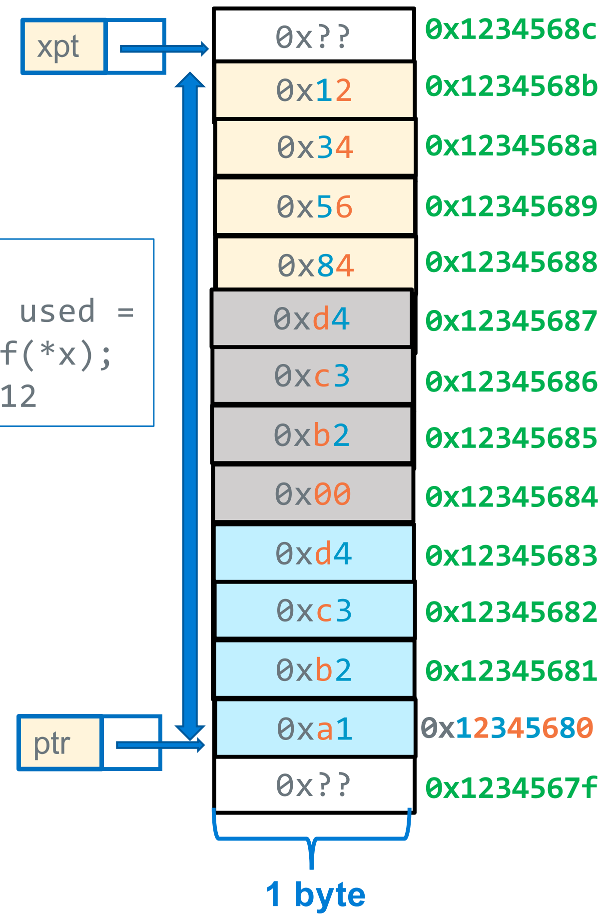
```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));

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++;
}
```

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

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



x