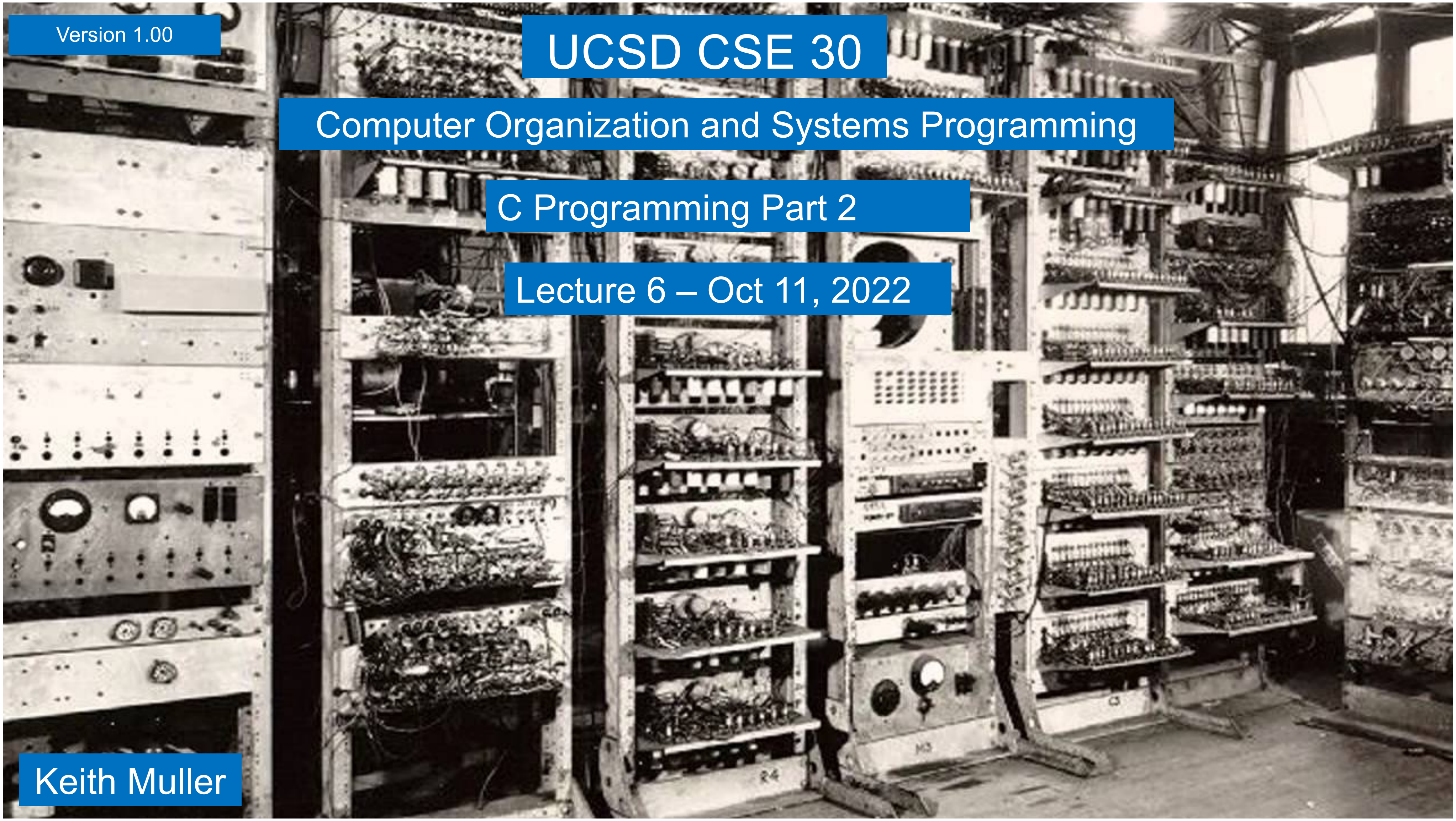


Version 1.00

UCSD CSE 30

Computer Organization and Systems Programming

C Programming Part 2

Lecture 6 – Oct 11, 2022

Keith Muller

Memory Review: Organized in Units of Bytes

- One bit (digit) of storage (in memory) has two possible **states**: 0 or 1
- Memory is organized into a **fixed unit** of 8 bits, called a **byte**

MSB Most Significant Bit → ← LSB (Least Significant Bit)

Bit position in Byte							
7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
Content of each bit							

- Conceptually, memory is a single, **large array** of **bytes**, **where each byte** has a unique *address (byte addressable memory)*
- An address is an **unsigned** (positive #) *fixed-length* n-bit binary value
  - Range (domain) of possible addresses = *address space*
- Each byte in memory can be **individually accessed** and operated on given its **unique address**

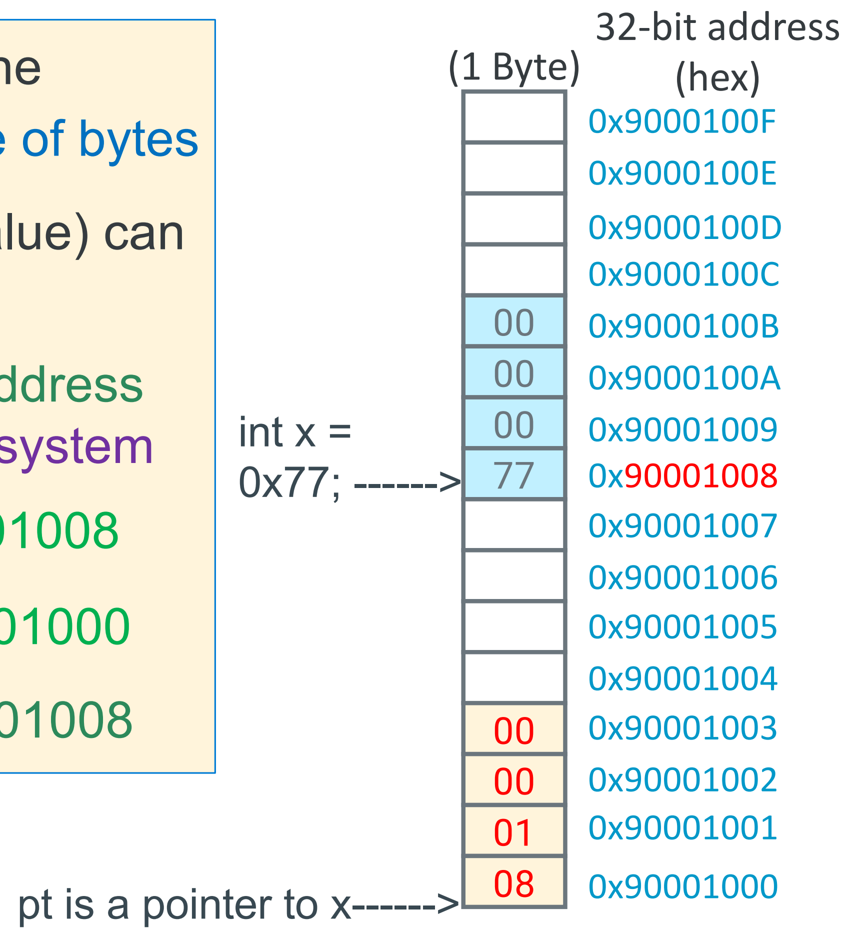
n-bit Memory Address	Memory contents
High address	
..90000111	10101010
..90000110	01010101
..90000101	10101010
..90000100	01010101
..90000011	10101010
..90000010	01010101
..90000001	10101010
..90000000	01010101
Low address	

$\underbrace{\qquad\qquad}_{\text{1 byte (8-bits wide)}}$

x

Address and Pointers

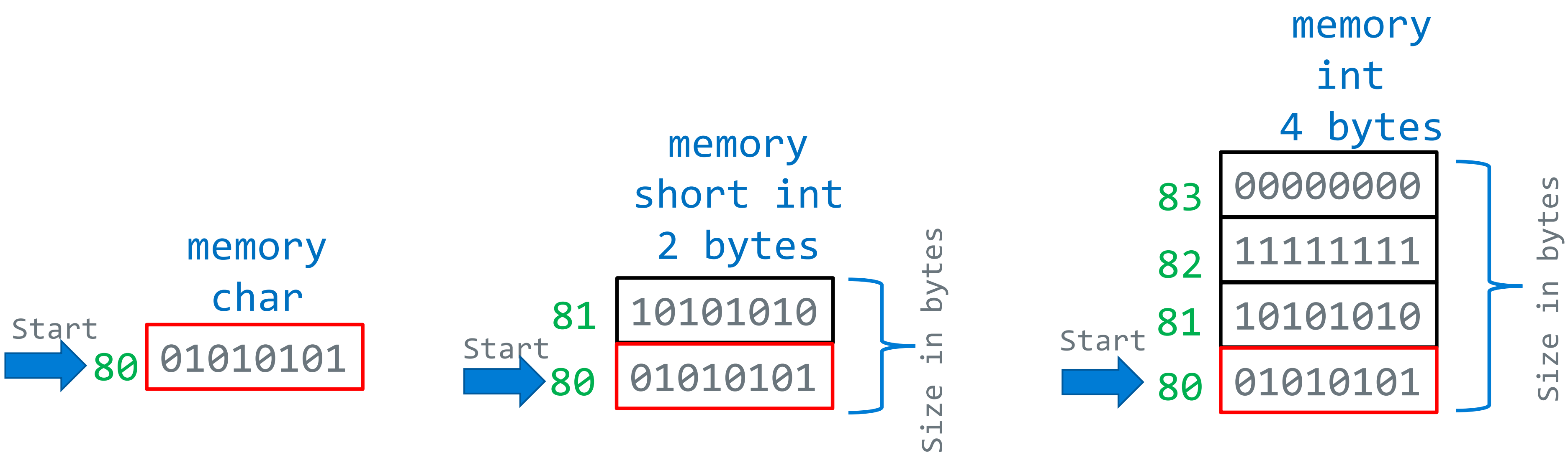
- An address refers to a location in memory, the lowest or first byte in a contiguous sequence of bytes
- A pointer is a variable whose contents (or value) can be properly used as an address
  - The value in a pointer *should* be a valid address allocated to the process by the operating system
- The variable x is at memory address 0x00001008
- The variable pt is at memory location 0x00001000
- The contents of pt is the address of x 0x00001008



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



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:

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

```
x = x;    // Lvalue = Rvalue
```

Variable name in a C statement evaluates to either:

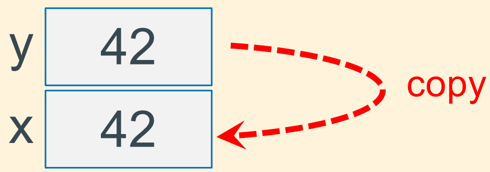
- **Lvalue:** when on the left side (Lside or Left value) of the = sign is the
  - address where it is stored in memory – a constant
  - Address assigned to a variable cannot be changed at runtime
- **Rvalue:** when on the right side (Rside or Right value) of an = sign is the
  - contents or value stored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

x

Memory Addresses & Memory Content

```
y = 42;

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



- **x** on left side (**Lside**) of the assignment operator = evaluates to:
  - The address of the memory assigned to the x – this is x's **Lvalue**
- **y** on right side (**Rside**) of the assignment operator = evaluates to:
  - READ the contents of the memory assigned to the variable y (type determines length) - this is y's **Rvalue**
- 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
- Requirement: **identifier must have a Lvalue**
  - Cannot be used with constants (e.g., 12) or expressions (e.g., x + y)
  - `&12` does not have an *Lvalue*, so &12 is **not** a legal expression
- How can I get an address for use on the **Rside**? Three ways:
  - `&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

Introduction: Address Operator: &

- Unary ***address operator*** (&) produces the **address** of where an identifier is in memory

- Example: this might print:

  value *of g is: 42*

  address *of g is: 0x71a0a0*

  (the address will vary)

```
int g = 42;
int
main(void)
{
    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: Pointer Variables - 1

- In C, there is a *variable* ***type*** for **storing an address**: a ***pointer***
  - **Contents** of a pointer is an **unsigned** (0+ positive numbers) **memory address**

- When the **Rside of a variable** contains a **memory address**, (it **evaluates** to an **address**) the variable is called a **pointer variable**

- A pointer is defined by placing a ***star*** (or ***asterisk***) **(*)** ***before*** the identifier (name)

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

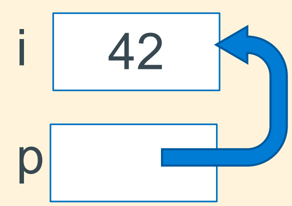

x

Introduction: Pointer Variables - 1

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

- You also must specify the type of variable to which the pointer points

```
int i = 42;
int *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

Introduction: Pointer Variables - 2

- **Pointers are typed**! Why?
  - The compiler needs the size (sizeof()) of the data **you are pointing at** (number of bytes to access)
- A pointer definition:

```
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

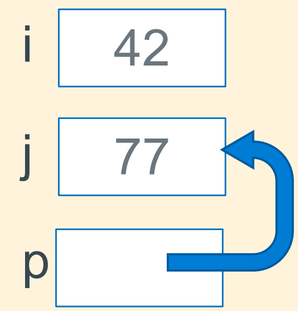
```
int  *p = &i;       /* Style A */
int * p = &i;       /* Style B */
int*  p = &i;       /* Style C */
```


x

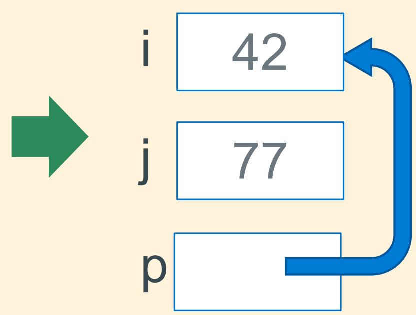
Introduction: Pointer Variables - 3

- As with any variable, its value can be changed

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




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



Introduction: Pointer Variables - 4

- 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

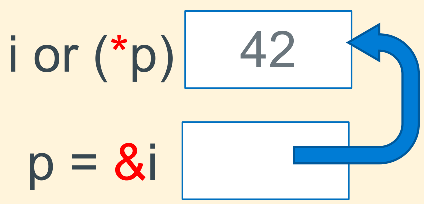
```
iptr(4) cptr(4)
```


x

Introduction: 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"

x

Introduction: Indirection (or dereference) Operator: *

*Contents of **p** is the address of **i** (p points at i)*

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

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

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

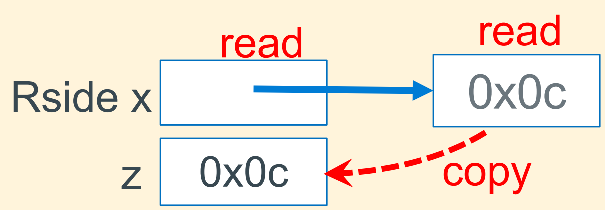

x

Introduction: Indirection Operator Rside

- 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
```



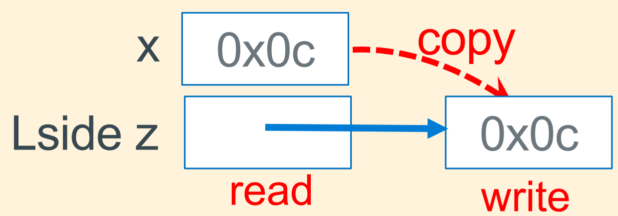
x

Introduction: Indirection Operator Lside

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)

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

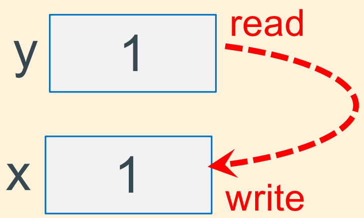


x

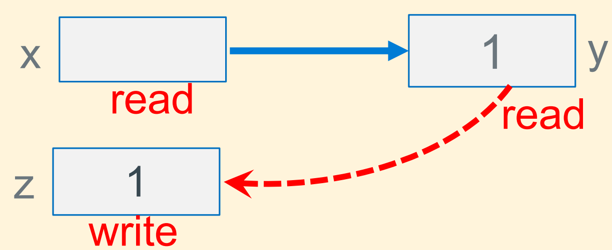
Each use of a * operator results in one additional read -1

Each * when used as a dereference operator in a statement (Lside and Rside) generates an additional read

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




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

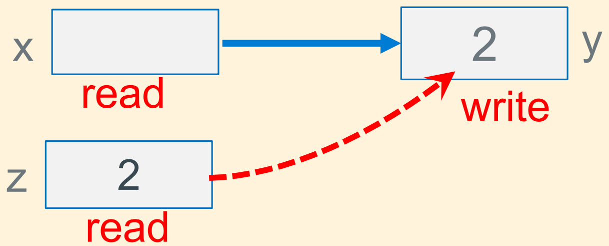


x

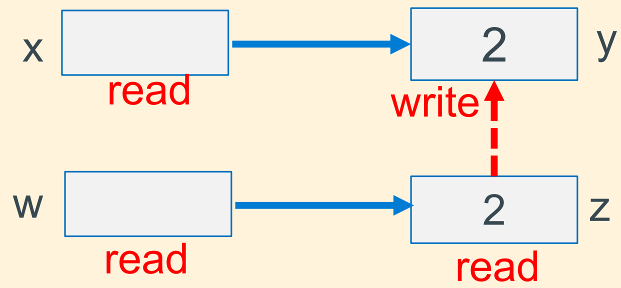
Each use of a * operator results in one additional read -2

- Each * when used as a dereference operator in a statement (Lside and Rside) generates an additional read

```
int z = 2, y = 1;
int *x = &y;
*x = z;
```




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



x

Recap: Lside, Rside, Lvalue, Rvalue

```c
int x = 2, y = 1;
x = y;
```

Constant Var Name	Lvalue address	Rvalue Contents	
y	0x108	0x1	read
x	0x104	0x1	write

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

```
*x on Lside is   0x10c
 w on Rside is   0x100
*w on Rside is   2
```

Constant Var Name	Lvalue address	Rvalue Contents	
x	0x10c	0x108	read
y	0x108	0x2	write
z	0x104	0x2	read
w	0x100	0x104	read

x

Pointer Practice

```
int *ptr;
```

Declares a variable, `ptr`, which is a pointer to (*it* contains the address of) an `int` in memory

```
int x = 5;
int y = 2;
```

Declares two variables, `x` and `y`, that contain `int`s, and *initializes* them to 5 and 2, respectively

```
ptr = &x;
```

Sets `ptr` to contain the address of `x` (“`ptr` points to `x`”)

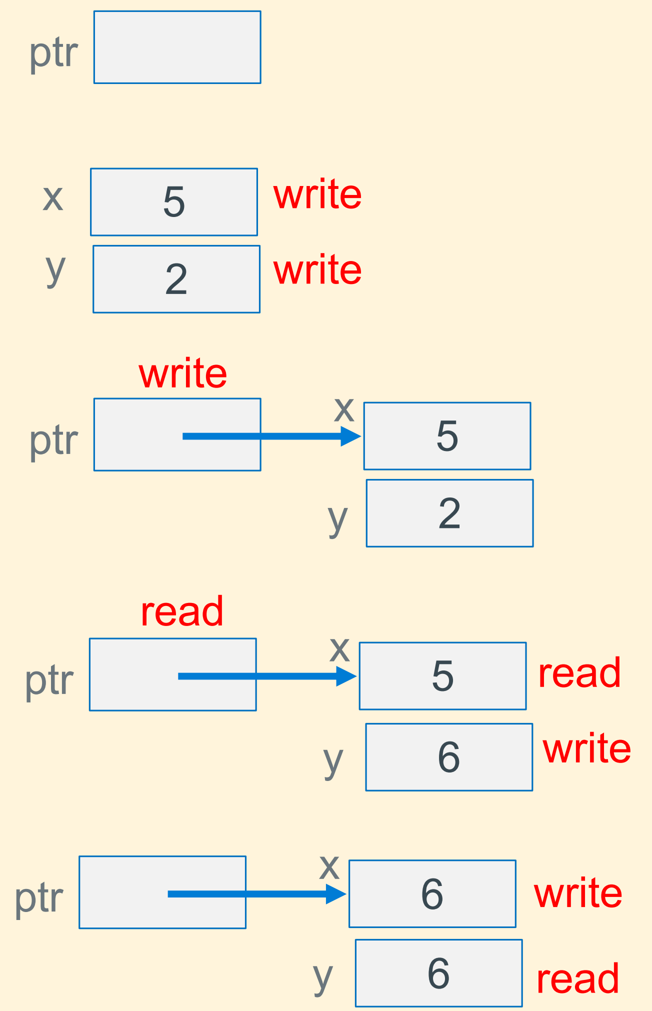
```
y = 1 + *ptr;
```

“Dereference `ptr`”

Sets `y` to “1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y = 1 + x;`

```
x = *(&y);
```

Sets x = y; The * and & cancel each other. get the address of y and then get the contents pointed by that address



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) ...
```

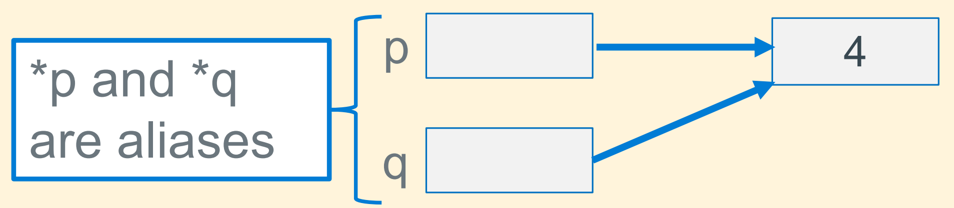

x

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)
- When one pointer is copied to another pointer it *creates an **alias***
- ***Side effect***: Changing one variables cvalue changes the value for another 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 = &i;
int *q;

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



x

Defining Arrays - 1

Definition: `type name[count]`

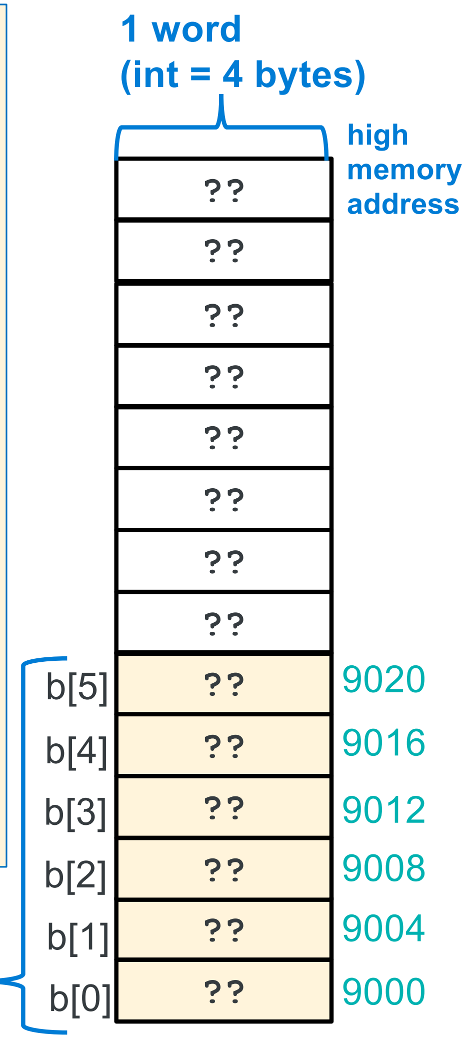
- *"Compound"* data type where each value in an array is an element of type
- Allocates **name** with a *fixed* count array elements of type **type**
- Allocates (`count * sizeof(type)`) bytes of ***contiguous memory***
- Common usage is to specify a compile-time constant for **`count`**

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

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

- Array **names are constants (like all variable names)** and cannot be assigned (cannot appear on the Lside by themself)

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




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

x

Accessing Arrays Using Indexing

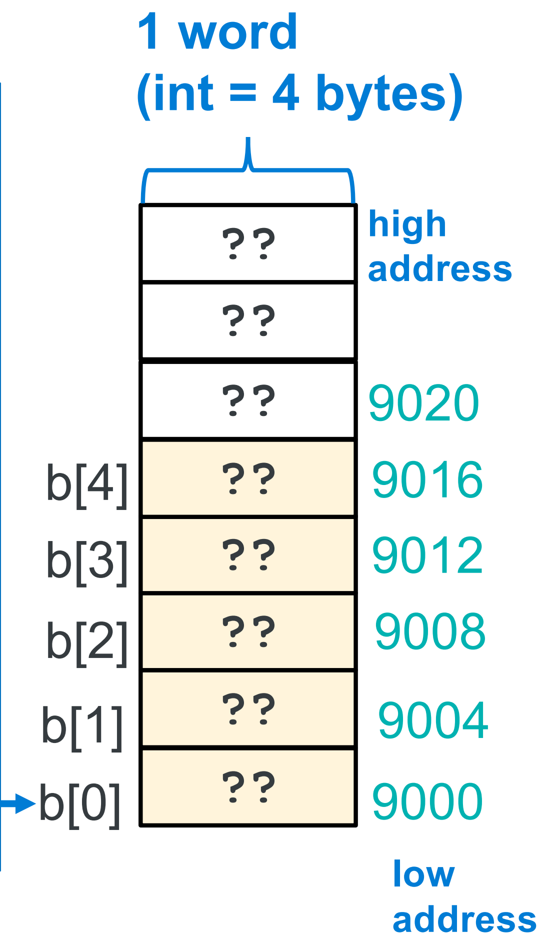
- **`name[index]`** selects the **`index`** element of the array
  - index **should be** unsigned
  - Elements range from: 0 to count – 1 (int x[count];)
- **`name[index]`** can be used as an assignment target or as a value in an expression

```
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;
```

p 9000



x

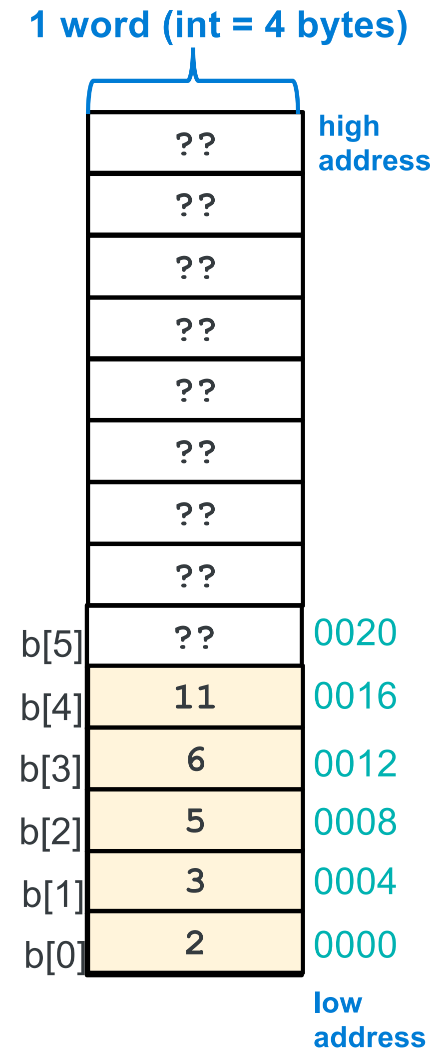
Array Initialization

- 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**



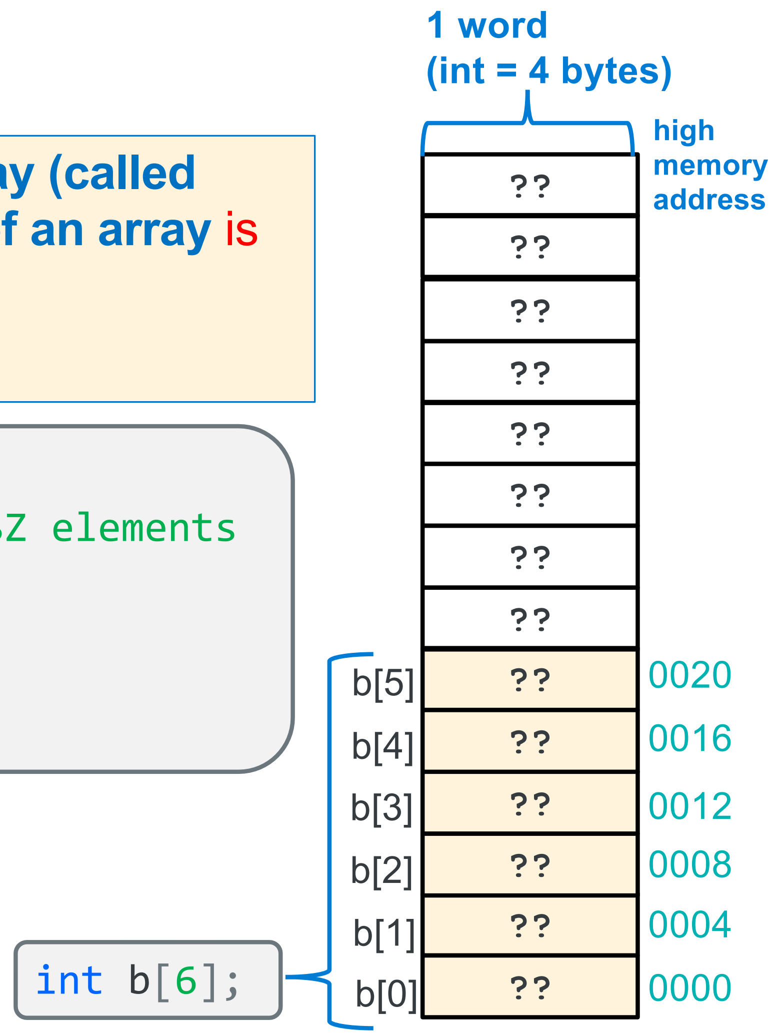
x

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** <u>is not stored anywhere</u>!!!!!!
  - **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;
```



x

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)` **only works** 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

Pointer and Arrays - 1

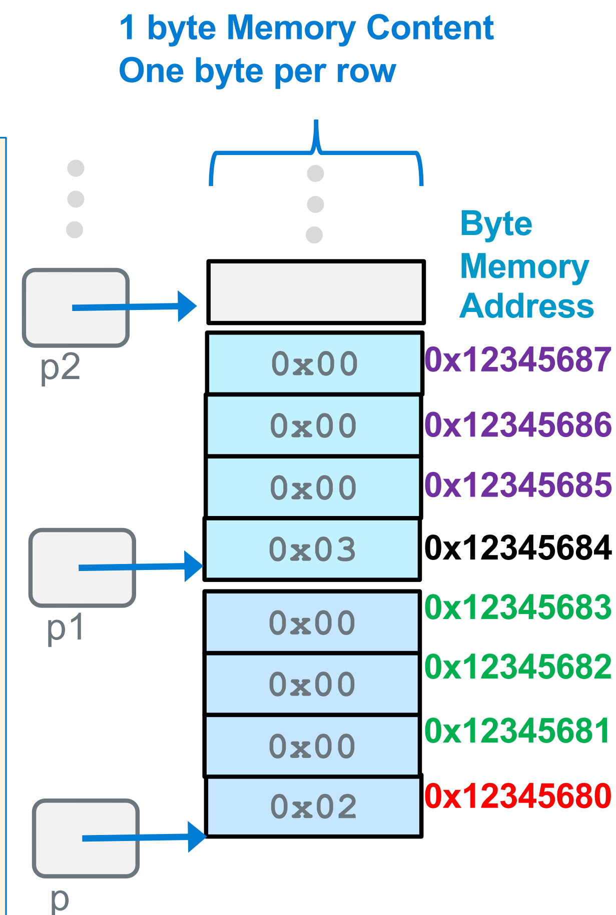
- A few slides back we stated: Array name (by itself) on the Rside evaluates to the address of the first element of the array

```
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 point at 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

Pointer and Arrays - 2

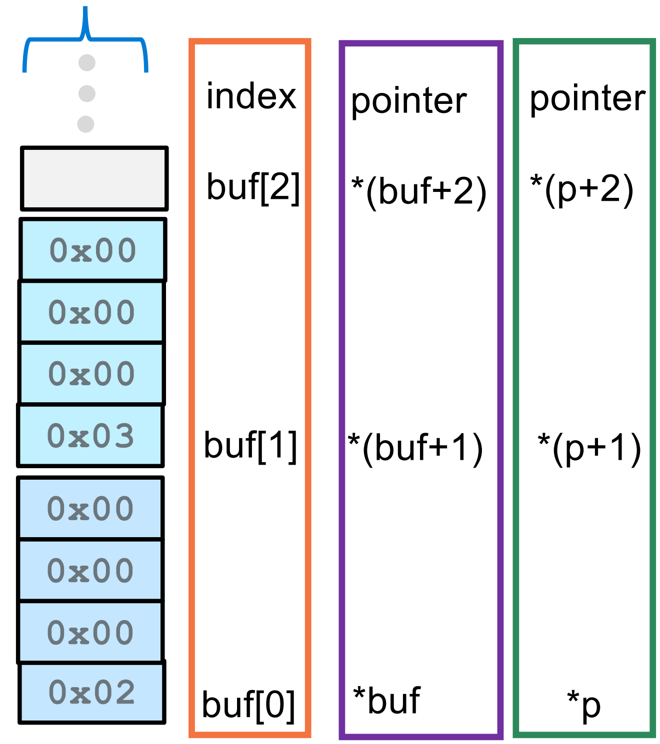
When p is a pointer, the actual value of `(p+1)` **depends on the type** that pointer p points at

- `(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 = buf;

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

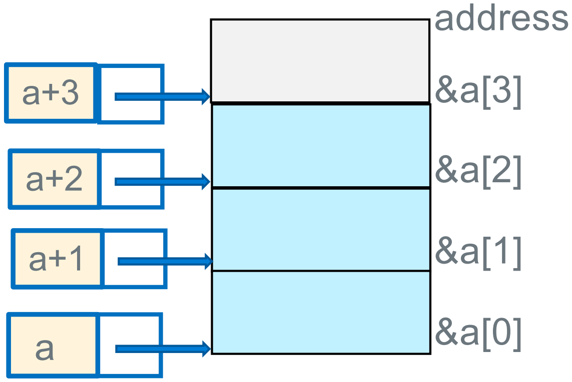
1 byte Memory Content
One byte per row



x

Pointer Arithmetic In Use – C's Performance Focus

```c
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: $0 \le j < i$)
  - 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

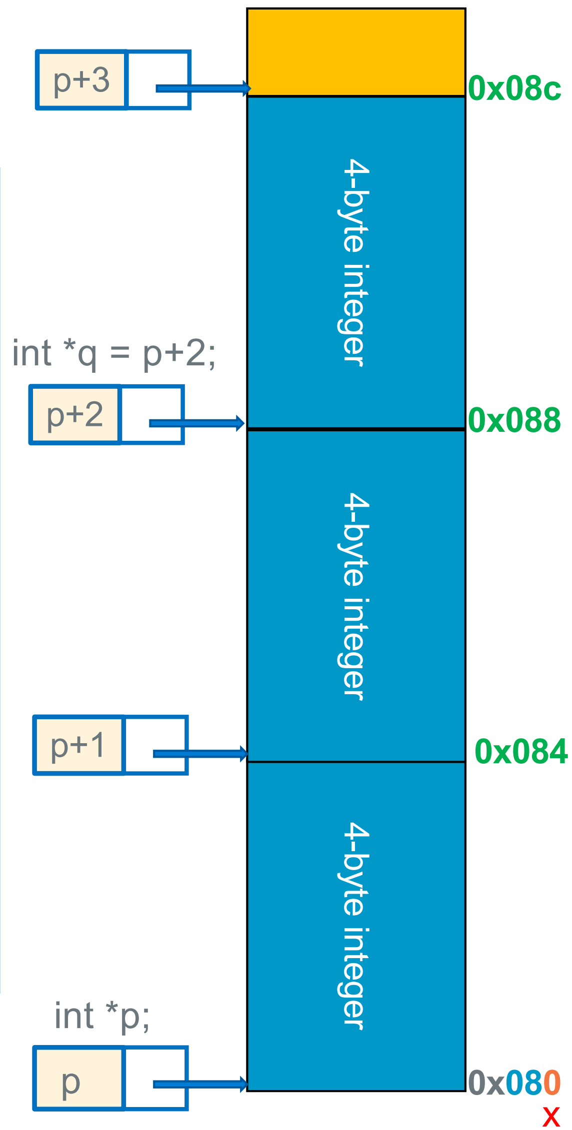
x

Pointer Arithmetic

- **You <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):

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

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



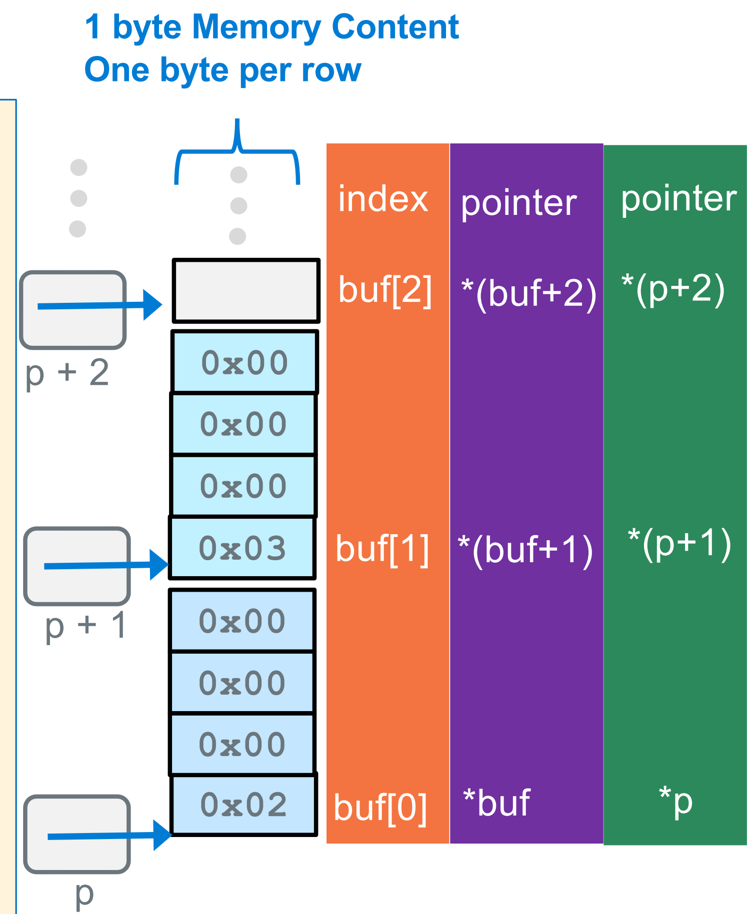
Pointer and Arrays - 2

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`
  - Comment: **`++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 = buf;

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



x

Pointer Comparisons

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

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

```
int numb[] = {9, 8, 1, 9, 5};
int *end = numb + (int) (sizeof(numb)/sizeof(*numb));
int *a = numb;

while (a < end)  // compares two pointers (address)
      /* rest of code */
```

- 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

Fast Ways to "Walk" an Array: Use a Limit Pointer

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


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

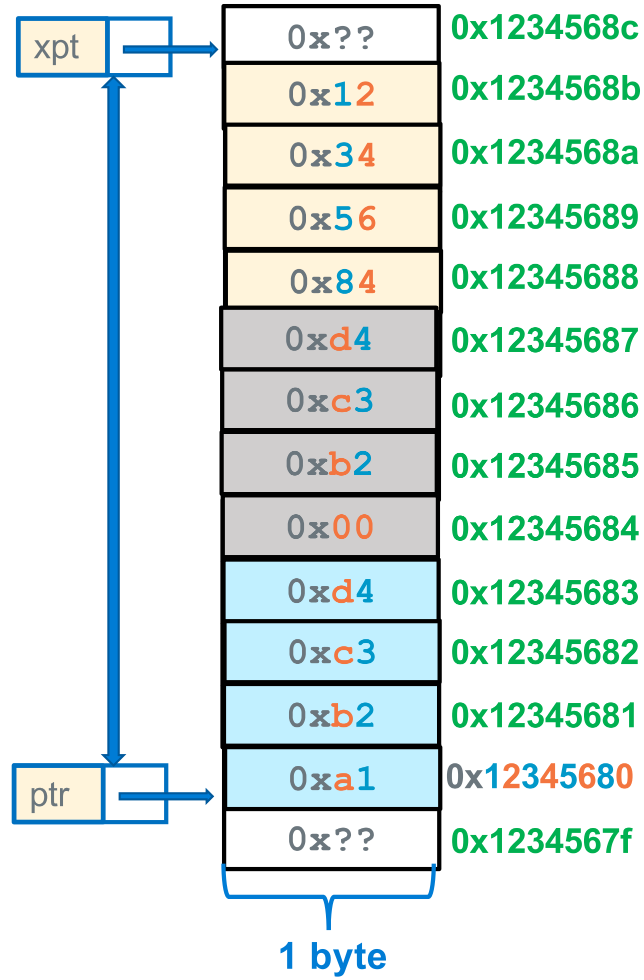
xpt is a loop **limit pointer** points 1 element past the end of the array

```
cnt   = 3;
bytes = cnt * sizeof(*x);
      = 12
```

```
int *xpt = ptr + cnt;

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

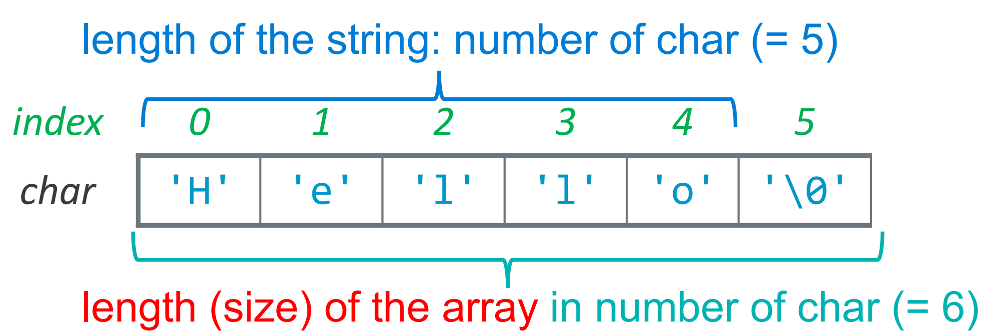
```
 % ./a.out
0xd4c3b2a1
0xd4c3b200
0x12345684
```



x

C Strings - 1

- **C <u>does not</u>** have a **dedicated type** for strings
- **Strings are** an **array of characters terminated by** a sentinel termination **character**
- `'\0'` is the **Null termination character;** has the **value of zero (do not confuse with '0')**
- An **array of chars** contains **a string only <u>when</u>** it is terminated by a '\0'
- **Length of a string** is the number of characters in it, <u>not including</u> the '\0'
- Strings in C are **<u>not</u>** objects
  - No embedded information about them, you just have a name and a memory location
  - You cannot use **+** or **+=** to concatenate strings in C
  - For example, you must **calculate string length** using code at runtime looking for the end



x

C Strings - 2

- **First `'\0'` encountered from the start of the string** always indicates the end of a string
- The **'\0' does not have to be** in the **last element in the space allocated to the array**
  - But, String length is always less than the size of the array it is contained in
- In the example below, the array buf contains two strings
  - One string starts at &(buf[0]) is "cat" with a string length of 3
  - The other string starts at &(b[4]) is "o" with a string length of 1
  - "o" has two bytes: 'o' and '\0'

string length: number of char (= 3)

string length: number of char (= 1)

index	*0*	*1*	*2*	*3*	*4*	*5*
buf	`'c'`	`'a'`	`'t'`	`'\0'`	`'o'`	`'\0'`
	`0x63`	`0x61`	`0x74`	`0x00`	`0x6f`	`0x00`

length (size) of the array in number of char (= 6)

x

Defining Strings: Initialization

- When you combine the automatic length definition for arrays with double quote(") **initialization**
  - Compiler automatically adds the null terminator '\0' for you

```
char a[4] = {'c', 'a', 't', '\0'};
char b[] = "cat";                          // compiler calculates size, adds '\0'
char c[] = {'c', 'a', 't', '\0', 'a, 'b'}; // array size 6, string length 3
char empty[] = "";                         // empty string - contains '\0'
                                           // string length = 0
```


x

Defining Strings: Initialization Equivalents

- Following definitions create **equivalent** 4-character arrays
  - These are all strings as they all include a null ('\0') terminator

```
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};      // missing initial value defaults to 0
char d[4] = { 99, 97, 116, 0};    // 99 = 'c', 97 = 'a', 116 = 't'
char e[4] = "cat";
char f[4] = "cat\0";              // literal has 5 chars; array f string
                                 // length is 3
```

When a double quoted string is used in an expression, it has a different meaning (next slide)

x

String Literals (Read-Only) in Expressions

- When strings in quotations (*e.g.*, "string") are **part of** an **expression** (*i.e.*, *not* part of an *array initialization*) they are called ***string literals***

```
printf("literal\n");
printf("literal %s\n", "another literal");
```

- What is a ***string literal:***
  - Is a null-terminated string in a **`const char array`**
  - Located in the **read-only data** segment of memory
  - Is not assigned a variable name by the compiler, so it is only accessible by the location in memory where it is stored
- **String literals** are a type of ***anonymous variable***
  - Memory containing data without a name bound to them (only the address is known)
- The *string literal* in the printf()'s, are replaced with the starting address of the corresponding array (first or [0] element) when the code is compiled

x

String Literals, Mutable and Immutable arrays

```c
char mess1[] = "Hello World";
char *ptr = mess1;
*(ptr + 5) = '\0'; // shortens string to "Hello"
```

- mess1 is a **mutable** array (type is char [ ]) with enough space to hold the string + '\0'
  - You **can change** array contents

```c
char *mess2 = "Hello World";  // "Hello World" is a string literal
                              // mess2 is a pointer NOT an array!
```

- In the example above, "Hello World" is immutable string literal (array)
  - "Hello World" is not associated with a variable name; anonymous variable
  - "Hello World" has space to hold the string + '\0'
  - "Hello World" is read only (immutable) and cannot be modified at runtime
- mess2 is a **pointer** to an **immutable** array with space to hold the string + '\0'

x

Be Careful with C Strings and Arrays of Chars

mess2 **pointer** to an **immutable** array with space to hold the string + ’\0’

- you **cannot change** array contents, but you can change what mess2 points at

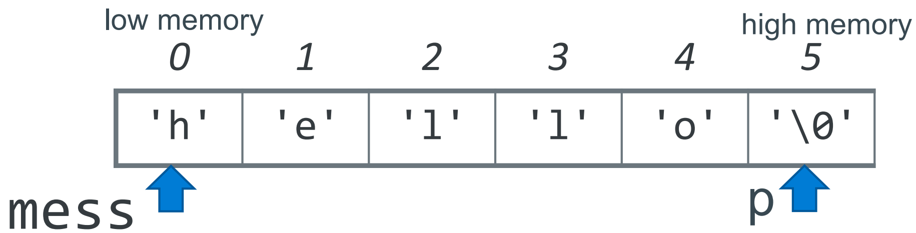
```
char *mess2 = "Hello World";  // "Hello World" is a string literal
                              // mess2 is a pointer NOT an array!
*mess = 'h';                  // undefined in C, linux seg fault
mess2 = mess1;                // where mess2 points can be changed
```

- mess3 is an array but does not contain a ’\0’
  - SO, IT IS **NOT** A VALID STRING

```
char mess3[] = {'H','e','l','l','o',' ','W','o','r','l','d'};
```


x

Finding the Length of a String: By counting the chars




```c
char mess[] = "Hello World";
char *p = mess;

while (*p++ != '\0')
   ;
printf("string length is %d\n", p - mess);
```


x

Background: Different Ways to Pass Parameters

- **Call-by-reference (or pass by reference)**
  - Parameter in the called function is an ***alias*** (references the same memory location) for the supplied argument
  - Modifying the parameter modifies the calling argument

Call-by-value (or pass by value) (C)

- What **Called** Function Does
  - Passed Parameters are used like local variables
  - Modifying the passed parameter in the function is allowed just like a local variable
  - So, writing to the parameter, ***only*** changes the ***copy***
- The return value from a function in C is **by value**

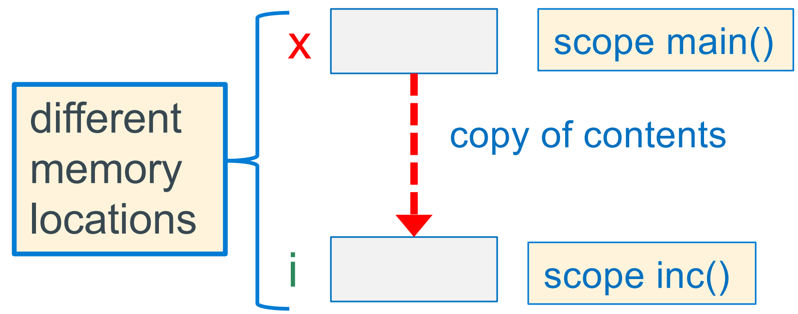
x

Passing Parameters – Call by Value Example

```
int main(void)
{
    int x = 5;
    inc(x);                // makes a copy of x
    printf("%d\n", x); // 5 or 6 ?
}

void inc(int i)        // i is local to inc
{
    ++i;
}
```

if this was an expression like inc(x+1) it evaluates and stores the result in the memory allocated for the copy



- when `inc(x)` is called, a copy of x is made to another memory location
  - `inc()` cannot change the variable x since `inc()` does not have the address of x, it is local to `main()` so, 5 is printed
- The `inc()` function is free to change it's copy of the argument (just like any local variable) remember it does NOT change the parameter in `main()`

x

Function Output Parameters: Passing Pointers

- Passing a pointer parameter with the **intent** that the called function will use the address it to store values for use by the calling function, then pointer parameter is called an **output parameter**
- Enables additional *values to be returned (besides the return)* from a function call

```
void inc(int *p);
int main(void)
{
 int x = 5;
 inc(&x);
```

- With a pointer to x, inc() can change x in main()
  - This is called a *side-effect*
- inc() can also change the *value* of p, the copy, just like any other parameter

- C is still using "*pass by value*"
  - we pass the **value** of the address/pointer in a **parameter copy**
  - **The called routine** uses the address to change a variable in the caller's scope

x

How to Implement Output Parameters

- To pass the address of a variable x use the **address operator** (&x) **or** the contents of a pointer variable that points at x
- To be receive an address in the called function, define the corresponding parameter type to be a pointer
  - It is common to describe this method as: “pass a pointer to x”

```
void inc(int *p);  // inc() is passed an address
…
inc(&x);           // pass the address of a variable to inc()
```

- Be careful when passing and using pointers
  - When you have the address of a memory location you are in effect over-riding (or by-passing) scope protections for accessing variables
  - Remember: Linux does not enforce or even know C scope rules, it will only prevent memory access (either address or write restrictions) on the address space of your executing program

x

Example Using Output Parameters

```c
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);
    printf("%d\n", x);
    return EXIT_SUCCESS;
}

void
inc(int *p)
{
    if (p != NULL)
        *p += 1;          // or (*p)++
}
```

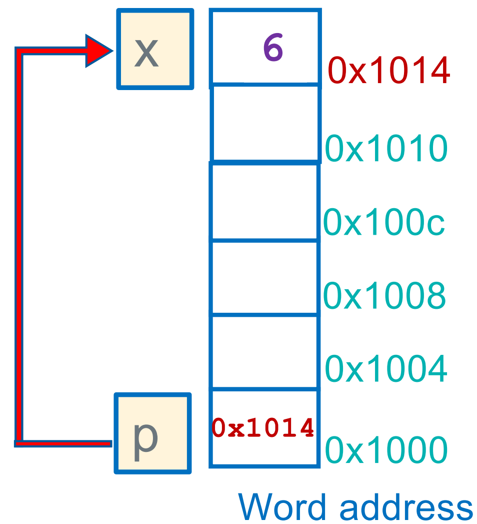
Pass the address of x (&x)

Receive an address copy (int *p)

Write to the output variable (*p)

At the Call to inc() in main()

1. Allocate space for p
2. Copy x's address into p



x

Finding the Length of a String : strlen counts the chars

low memory — high memory

0	1	2	3	4	5
'h'	'e'	'l'	'l'	'o'	'\0'
s					p

- C string library function **`strlen()`** calculates string length **at runtime**
- **Do not overuse strlen(), as it walks the array each time called**

```c
/* Assumes parameter is a terminated string */
int my_strlen(const char *s)
{
   char *p = s;
   if (p == NULL)
       return 0;
   while (*p)
       p++;
    return (p - s);
}
```

```c
int count_e(const char *s)  // o(n^2) !!!
{
   int count = 0;
   if (s == NULL)
       return 0;
   for (int j = 0; j < strlen(s); j++) {
       if (s[j] == 'e')
           count++
   }
   return count ;
}
```

```c
int count_e(const char *s)  // o(n) !!!
{
   int count = 0;
   if (s == NULL)
       return 0;
   while (*s) {
       if (*s++ == 'e')
           count++
   }
   return count ;
}
```


x

To be continued….